OCR post-processing for Dutch literature

DATA SCIENCE PROJECT – FINAL REPORT

Floris Cos (*s1027889*)
Tamara Mauro (*s1027617*)
Boudewijn Schiermeier (*s1010135*)
Annemijn van Klink (*s1022396*)

18th of June 2021

Course coordinator: Stefan Frank
Supervisor/TA: Iskaj Janssen

Abstract

This paper discusses an algorithm for post-OCR correction. The algorithm contains functions that find and repair common character-based mistakes; it uses the context-based language model roBERTa, as well as the word-similarity based Levenshtein Distance measure in order to correct wrongly recognized words, based on a Dutch lexicon. After construction, the algorithm was applied to 154 Dutch literary books from the years 1961 to 1965, to fix mistakes originating from the digitization of the texts. A character and word error rates evaluation shows that application of this algorithm improves the quality of the texts in comparison to the original scans, using digitally published books as a reference. It is important to realize that the algorithm is constructed specifically for the data at hand and may therefore be too specific to be used in general. Since different OCR methods make different mistakes, it is advisable to construct an algorithm that is tailored to the mistakes found in the text that is to be processed. However, our method could be an inspiration for others and parts of our program could certainly be reused."

Table of contents

1. Introduction

Optical Character Recognition, also known as OCR, is a method to convert a picture of text into actual, readable text for a computer (Govindan & Shivaprasad, 1990). It recognizes characters in a picture and constructs a text file accordingly. The purpose of OCR is to make text searchable or to be able to analyse it. The recognition of character features is done automatically. All computers and computer programs are fallible however, and thus OCR is not always performed flawlessly. Oftentimes, success rates of around 90% are reported (Govindan & Shivaprasad, 1990), which indicates that 90% of the characters are accurately recognized.

In this report, we will explain how we have tried to correct the OCR-output of a corpus of 154 Dutch books from the years 1961 until 1965. This corpus will be used by our client to compare social networks in real life to those in books. This will be done in combination with an already digitized corpus of Dutch literature from the year 2012. The research will be similar to Volkert & Smeets (2020). Therefore, there are a few specifics to this assignment. An example is that the names of the protagonist and other persons in the story stay the same, even if they are in archaic or incorrect spelling. Because the books are in Dutch, compounds exist that are not necessarily in a dictionary. These and others are all aspects which we tried to take into account. Another important aspect is that our code shouldn't make the text worse than it was. This sounds obvious, but with an algorithm that replaces words, this could be a pitfall.

There were different kinds of mistakes in the text. The most common ones had to do with spaces, in different kinds of ways. First of all, a single letter could be separated from the rest, see 1. The spaces could also be throughout the whole sentence, or even more sentences (see 2). Other mistakes mostly had to do with the program confusing several characters or letters (see 3 and 4).

(1) l ike this.
('like this'; one letter separated from the rest)

(2) T h a t l o o k e d l i k e t h i s.
('That looked like this'; all letters separate)

(3) I can/t think of / more example
('I can't think of 1 more example'; The / is often confused with a '1' or with ' ' ')

(4) Üg jij in de zon?

('Lig jij in de zon?', *Are you lying in the sun?*; the combination of letters 'Li' was confused with 'Ü')

In the past, much research has been done on OCR-correction. From this, several types of correction programs can be defined. Ma (2018) describes four: lexicon-based, context-based, Statistical Machine Translation (SMT) and Featured Based Word Classification (FBWC). Lexicon-based, most often applied with the Levenshtein distance, corrects a word based on its distance or similarity to an existing word. Context-based approaches operate on a calculation of how likely a word is used in a certain context. SMT and FBWC are more complicated and less relevant for this report. Other approaches are statistical (Lasko & Hauser, 2000; Mei et al., 2018) or use Machine Learning (Salimzaleh, 2019). In this report, we present an algorithm based on a combination of a lexicon-based and a context-based approach.

Research on OCR-correction, and specifically our code, is scientifically and societally relevant because it applies to a specific niche; it is relevant to Dutch literature from the 1960's. Most previous OCR-correction software is either not suitable to be used on Dutch texts or on literature[1], or it is specific to one period in time. From the beginning of OCR development, it has also been used by people with a visual impairment (Govindan & Shivaprasad, 1990). For those people, the mistakes in the OCR-layer can be limiting in how well they understand a text. A better version of the text can offer a better understanding and a more pleasant way of taking in the text.

In section 2, we will first explain how we approached this problem, how the final product of code works and how we applied it on the data. Then, in the section after that, we will discuss the evaluation of our code based on three full books, as well as the results. Lastly, in the Discussion section, we will reflect on the process and evaluate the execution of the task we were charged with. We will also suggest ideas and improvements for future research.

[1] https://github.com/KBNLresearch/ochre

2. Methodology

In this section, we will describe how we went about creating this post-OCR correction tool, what each of its components does and how we set it to work on the text files that had to be corrected.

2.1 Data

As stated in the introduction, the goal of this project was to write a post-OCR correction algorithm, to improve the quality of text of 154 classic Dutch Novels from the 1960's. These novels had all been run through a pre-existing OCR program which digitalized the printed novels. The output of this original program were flat texts. These texts contained many errors, like characters that were misrecognized and the insertion of too many spaces.

We started with taking a look at most of the text files and compiling a non-exhaustive list of the errors that we frequently encountered in the books. This was done partially manually, and partially by automatically checking the words of the books in a Dutch lexicon[2] to see what words deviated from correct spelling forms. After that, we were able to group these frequent errors into categories, each of which could be dealt with in a programming setting. These categories included the following list:

- Problems with page numbers (e.g. '10' recognized as 'io')
- Many problems with spaces (e.g. 'h em' instead of 'hem' *him* or 'h a l l o' instead of 'hallo' *hello*)
- General problems with characters that were often recognized incorrectly (e.g. a quotation mark recognized as '5' or '/' and strikingly 'ü' as 'li').

While automatically checking the words in the lexicon, we encountered many words that should have been treated as being correct, but were not. Here, too, some categories were identified, and we planned to handle these issues as well.

- Words cut in two by a line break hyphen; either part was often marked as incorrect.
- Words from other languages. We encountered English, German, French and Spanish.

[2] https://github.com/OpenTaal/opentaal-wordlist

- Compound words; In Dutch, new words can be formed by concatenating two smaller words. This results in an almost infinite amount of possible words that cannot all sensibly be recorded in a lexicon.
- All kinds of names

2.2 Methods

We decided to make an algorithm and take a step-based approach with this project. The programming was done in Google Colab (Bisong, 2019), so that all code could be accessed by all members of the project and the algorithm had a clear structure from the beginning. In this section, a very brief overview of the inner process of the algorithm is provided. For a pipeline visualization of the algorithm, see *Appendix I.*

Our program processes all lines of a book one by one. First, some general clean up functions are executed on a line. Second, we check for every word in the line if it is an existing word by running it through the Dutch lexicon. If the word is in the lexicon, the program moves onto the next word, until it finds a word that does not appear in a lexicon or until the sentence reaches its end. In case of a word that was not present in the lexicon, the word is run through two different functions in parallel: a pre-trained language transformer model called roBERTa (Delobelle et al., 2020), and a Levenshtein Distance Matrix. These will be explained in more detail in sections 2.2.2.2 and 2.2.2.3, respectively. We proceed with comparing the output of these two measures and if there is a match, we replace the non-existing word with the correct word, after which the program moves onto the next word. Afterwards, the line is returned to its original structure' and is written to a new file with the same name.

During construction, we tested each of the components making up the algorithm with a test text. We composed this text ourselves, and it contained many different errors, all of which we encountered while scouting in the texts. The test text can be found in *Appendix II: Test Text*. We also used this text to test what configuration of components yielded the greatest improvement in correction.

Our program is could be divided in three overarching parts: correction on line level, correction on word level and post-correction. Each of these parts will now be described in further detail, also highlighting decisions that were made during development.

2.2.1 Correction on line level

Every step of the algorithm, until the step of roBERTa and the Levenshtein Distance Matrix and further, is explained in the list below. Sections 2.2.2 and 2.2.3 connect directly and linearly to this list. For the code and the technical explanation of it, please take a look at the GitHub repository[3].

2.2.1.1 Installing Transformers Package

The pre-trained ML-algorithm we use, roBERTa, originates from a Python Transformers module. This contains the roBERT model, which is able to suggest words based on its calculations. The function loading in this library needs to be executed once before running the program. It is therefore executed on a higher level than the rest of the algorithm, because it is not necessary to load in the module for every single line.

2.2.1.2 Adding names and numbers to the wordlist

The research of our client is based on the names of characters in the book. It is therefore of paramount importance that the names are not meddled with by the program. Thus, we added a pre-existing list of all name variants to the lexicon. This way, we could be certain that they would always be recognized as being correct and hence left alone. We also added numbers from 0 to 300 to the lexicon, so that the algorithm would not waste time on these.

This function is also executed only once per runtime of the algorithm, because the operation does not have to do with each line individually, or even with each book.

2.2.1.3 Correcting and deleting page-numbering

As mentioned before, poorly recognized page numbering consistently plagues the OCR-output. This function contains a close to exhaustive list of things that often go wrong in page numbering, and changes these characters to their respective numbers – and only in lines that are so short they are likely to be page numbers.

The client expressed that he would like to receive the results without page numbers, so this function also removes these. It was still relevant to correct the numbers, because that made the removal a lot easier. Roman numerals are also removed in this function.

---

[3] https://github.com/FlorisCos/DSP_OCR_improvement_algorithm.git

Until here, functions were executed once or once per book. From now, operations will be performed per line.

2.2.1.4 Correcting the "space problem"

The "space problem", as we named it, was one of the more challenging conundrums to fix. It was the most common error in the OCR files, most likely caused by scanning pages while they were curved or because the text was justified without breaking words. Some examples are as follows, in addition to those mentioned in the introduction.

(5) h em
('hem', *him*; one single letter disconnected, usually the first letter)

(6) H a l l o
('hallo', *hell*; one single word in a line in single letters)

(7) ' J a , d i e o u d - l e e r l i n g v a n h e m.
('Ja, die oud-leerling van hem.', *Yes, that ex-student of his*; several words in a row to an entire line in single letters)

Some books contained entire paragraphs consisting only of single letters. It's almost impossible to let an algorithm determine how many words there should be in lines like that, especially since many words consist of several smaller words.

The solution we used is to concatenate all single letters in a sequence. This happens only if there are multiple single letters in a row (so the word 'u', *you (polite)* on its own remains intact), and if the single character is not a punctuation mark. This fixes single words, like example (6).

If more than 37% of the characters in the source line were spaces, an additional measure was taken. At this proportion and up, most of the characters in the line were isolated letters. In the previous step, these were pasted together to one single continuous string. These strings are likely to consist of several words, however. This part of the function separates these continuous strings into actual words that were as long as possible. The threshold of 37% was chosen because we could be certain that at that threshold the line would consist (almost) entirely of isolated letters. This approach appeared to be effective.

Finally, if the program encountered a single letter, the word following this letter was not in the lexicon and the combination of the single letter and the word did appear in the lexicon, this

letter and the loose word were replaced by the combination of the two. This fixes cases like example (5).

2.2.1.5 Word-wrap correction

Sometimes, words are cut in two by end-of-line hyphenation. The two parts of the split up word are often not in the lexicon. Also, if a name was broken up, it would not be recognized by the name list our client wished to use for his analysis. Therefore, we decided to take all word parts at the end of lines ending on a hyphen, remove them from the current sentence, and paste them without the hyphen at the start of the next line. This program was made robust against page number lines and blank lines.

2.2.1.6 Correcting and pre-processing of punctuation

A word with a period, comma or quotation mark attached to it would not be recognized as a word in the word list, because it is technically a different string. Therefore, we inserted a space before and after all interpunction marks, to be certain that all punctuation would be disconnected from words. In this step, we also replaced all possible types of quotation marks to a simple, single one, for easier processing down the line.

Finally, some characters that should have been recognized as quotation marks, but were not, are fixed here. This includes for example the ‘5’ and the ‘/’, both closing quotation marks.

2.2.1.7 Storing space configurations

It was our intention to remove the spaces added in the previous step after the words would be checked and possibly corrected, for a neater end result. Because we wanted to check the words of the sentence one by one, we converted the line string to a list. All spaces in the line were omitted in this process. There was no way however to assess for every single punctuation mark, quotation marks especially, on what side it would need a space when we turned the list into a string again after the correction (i.e. whether it was a quote opening or ending).

That is why in this function, the number of spaces leading before each word is stored within a dictionary. The words can be used as a list to be corrected, and their positions function as keys in the dictionary. After correction, the position of the words can be used to index the dictionary and build up the sentence just like it was – only with corrected words. In post-processing, any redundant spaces around interpunction can be removed without problems.

We tried an approach with storing the character positions of spaces within the line. We abandoned this plan however, because sometimes the length of the word is changed in correction. In such cases, spaces would appear in the middle of words when inserted afterwards.

In this function also, a very small number of very frequent OCR errors were fixed. These errors included:

- 'TK' or 'Tk' instead of 'Ik' (*I*)
- 'Cura.ao' or 'cura.ao', in which the period could be any character and was replaced by 'ç'.

2.2.1.8 Approving compound words

In Dutch, it is possible to combine any combination of nouns and adjectives into a single compound word. Literary writers quite like to use this option to create very original and never-before-used words to bring about certain feelings – or just for the sake of it. There is also no upper limit to the number of words that are used to create the compound.

Languages with this characteristic cannot possibly include all of these potential words in a lexicon, for it would contain billions and billions of words.

In principle, these words are correct, however, it would both be a waste of time and cause a decrease of the quality of the texts to have our language model come up with a word that would be 'correct'.

In this function, we check for words that are not in the lexicon whether two parts of the word that are present in the dictionary together make up the current word. We included several binding interfixes in this analysis. Words that are filtered out this way are approved. Moreover, these words are also added to the lexicon, so that this analysis only has to be done once. The added words remain also when the algorithm starts working on a new book.

2.2.2. Correction on word level

As previously mentioned, the program we built consists of different approaches in order to replace non-existing words with correct, existing words. After pre-processing is done, the input for the correction functions is a list, containing all words of a line in the book. This part of the algorithm consists of four steps that will be discussed below.

2.2.2.1 Running separate words through the lexicon

In this step, every word in the list of the line is separately run through our Dutch lexicon including the added words from pre-processing. If the word matches a word from the wordlist, it means that it is an existing word. In that case, there is no correction needed and our program moves onto the next word. Interpunction is also approved. When the program finds a word which does not exist in the wordlist, it is in need of correction. The program sends the single word to step (2), and the entire list of words on that line of the book to step (3).

We experimented with using dictionaries from English, German, French and Spanish, each with a length of 20,000 tokens, as these languages all made appearances in the books. This did not really slow down the processing, but with our test text, we found that performance was better without these additional dictionaries. This was mainly due to false positives in the recognition; 'wrong' words that would falsely be recognized as correct because they occurred in any of the other dictionaries or because they were inaccurately recognized as a compound word of two words in foreign dictionaries. Therefore, in the final algorithm, these additional dictionaries were not used. Including them was not worth the trade-off, as indicated by the evaluation tool. Results of this test can be found in *Appendix III*. In the end words from foreign languages generally would not be changed for the worse because of the interplay between the Levenshtein Distance matrix and the roBERTa language model.

Before sending a word to the language model, two additional checks were performed. Firstly, in archaic Dutch spelling, the letter K is often used where modern spelling uses the letter C, like 'kontakt' ('contact', *contact*). Our lexicon used modern spelling, so we included a conditional that if a word that was not in the lexicon contained a K, that K was replaced with a C and rechecked. If it was present this time around, the word in the list was replaced by this new word with a C, and the program moved on to the next word in the list.

Secondly, when a word is processed by the language models, the output is saved in a dictionary. Before being sent to the language models, it is checked if the word has previously been processed by the language models (i.e. if it is in the dictionary). If so the word is replaced by the previous solution, that was saved in the dictionary. If not, the word is sent to the language models.

2.2.2.2 Levenshtein Distance Matrix

The input for this step is a single word from the last step. Say, in this case that input word is the word ‘pape’; a non-existing word which needs correction. The Levenshtein Distance metric measures the distance between two words, through a matrix. The optimal Levenshtein Distance is the minimal number of single-character edits (deletions, insertions or substitutions) required to change one word into another. In our example of “pape”, words with a low Levenshtein Distance could for example be “page” or “paper”.

Our program inputs a word like “pape” and outputs a list of 25 words with the lowest Levenshtein Distance, thus the 25 “closest words”. We chose the hyper parameter of 25 after testing and pruning.

2.2.2.3 roBERTa Language Model

This step takes as input all words of the line surrounding the incorrect word as context, including the incorrect word itself. The roBERTa model is a pre-trained machine learning algorithm that is trained with a language masking objective function using a large corpus of texts (Delobelle et al., 2020). The model is available in the Python module package Transformers. It takes the sentence as input, with one masked word, the incorrect word. It reads the context and decides, based on probability calculations, which word is the most likely to replace the incorrect word, given the context of the sentence. The output of this model is, equivalent to the Levenshtein output, a list of 25 suggested words; in this case these are the words with the highest probability of being the correct word, given the context of the sentence.

2.2.2.4 Matching & replacing

The last step is the most obvious, though very important step. If there is a match in the suggestive wordlists from Levenshtein and roBERTa, the incorrect word will be replaced with the word that appears in both lists. We tested this and this resulted in lower error scores every time. Though, we also tried replacing it when there was no match, taking the first suggestion of either the Levenshtein Matrix or the roBERTa model. This resulted in a higher error rate, so we chose not to do this. Indeed, the lowest error rate was scored when we replaced the word when there was a match, but left the old word when there was no match. For an overview of the results of these tests, see *Appendix III*.

Match or no match, the output of the model is saved in a dictionary so that the same word does not need to be processed all over again when it is encountered multiple times, as described in section 2.2.2.1.

2.2.3 Post-correction

After checking and correcting the words, we first converted the python list back to a string, while inserting all the spaces in the positions corresponding with the dictionary described in section 2.2.1.7.

The last step entails removing the spaces around punctuation marks, and writing each of these lines beneath each other in a new text file, that has the same name as the title of the book followed by '_verbeterd.txt'. All these steps result in a better version of the previous OCR file of the book.

2.3 Applying the algorithm to the books

After constructing and extensively testing the algorithm, we released it on the corpus of 154 books that needed to be checked. We used two computing nodes of the csedu partition of the Science Faculty of the Radboud University, each consisting of two Intel Xeon Silver 4214 processors, running with Slurm (Yoo et al., 2003) on a Linux/Unix based system.

A function was added on a higher level, so that multiple books could be processed in succession, based on a file with the titles of these books. The output was put in separate files for each of the books, as mentioned in section 2.2.3.

Because of constraints on speed and memory per user, the algorithm was run in triple parallel. The 154 books were divided into three groups. The books were processed in a Python virtual environment, of which we had three running simultaneously. A short manual for effectively and appropriately setting up such a virtual environment can be found in *Appendix IV*. The output can be found in a repository on the website of the Open Science Foundation[4].

There are a few important notes that could halt the processing. First of all, when the roBERTa language model is prompted to calculate the most probable word, it receives the whole line as context. The upper limit to the amount of tokens appears to be 512, but numbers over 100 or

[4] https://osf.io/uhkce/

so may already cause problems. Therefore, it must be checked that the length of the lines is reasonable (say, at most one or two sentences, 20 to 30 words). Secondly, the algorithm may get stuck on unrecognizable code symbols in for example ISO-coded files. Files that are coded in ISO-8859-1 must be recoded to preferably UTF-8.

3. Results

In this section, we will present the results of our correction program. We decided to use the ocrevalUAtion tool (Carrasco, 2014) to test the extent of improvement provided by our algorithm. First, the usage of the tool will briefly be described. After that, we'll give a brief overview of the testing procedure and the results of our tests.

3.1. OcrevalUAtion tool

This is an evaluation program designed specifically for comparing two texts and expressing where and how much they differ. For an in-depth description of the inner workings, see Carrasco (2014). The procedure of using the tool is quite straightforward; there are two text windows. In one of them, a so-called ground-truth file can be entered, and in the other one the starting or resulting file (i.e. before and after the correction). The program highlights differences between the two within the text, and expresses the dissimilarity between the texts in two different measures, the Character Error Rate (CER) and the Word Error Rate (WER). We will discuss these now.

3.1.1. Character Error Rate

The CER is computed by comparing the words on the same position in the ground-truth and the other file. This computation considers not only how many characters are in the wrong place, but also how many actions are needed to correct it. This makes the CER a thorough and relatively robust measure.

Carrasco (2014) mentions the example of 'nester' and 'ernest'. If only the positions of the letters are considered, this word has an error rate of 100%. Large parts of the words, the 'nest' part in either word, are technically in accordance, however. It is only the 'er' part that is in the wrong spot. A Dutch example would be 'kaas' (*cheese*) in the ground-truth text and 'aska' in the text to be compared; at first the word looks plainly wrongly recognized, but in fact, the 'ka' and 'as' parts are correct on their own.

To compute this dynamic CER, the program makes use of the Levenshtein distance, the amount of operations needed to turn the tested word into the true word. To calculate the measure, the following formula is used.

(1) $CER = (i + s + d) / n$

In this formula, *i* is the number of inserted characters. *s* is the number of substituted characters, *d* is the number of deleted characters and *n* is the total number of characters in a word.

3.1.2. Word Error Rate

The computation of the WER value largely follows the same logic as the calculation of the CER value. The difference is that the WER operates on a word level. That means that it is not about characters being compared within a word, but words within a line. The rest of the logic is very similar to that of the CER calculation and thus the formulas look a lot alike.

(2) $WER = (i_w + s_w + d_w) / n_w$

In this formula, the *i*, *s*, *d* and *n* stand for the number of inserted, substituted, deleted and total words respectively, the subscript *w* indicating that these variables are about words.

It is important to realize that in the WER analysis, the content of words is not taken into account; it is about the words as symbols (i.e. as a whole). For example, having ‘nester’ in the text where ‘ernest’ should be is considered just as wrong as having ‘emast’ or even ‘apple’ in that same place, even though these options differ in their CER values.

3.2. Evaluation

After searching extensively, we found free ebooks for three of the processed books. As these were professionally distributed, we assumed a certain level of quality control by the publisher. Together with a brief scan of the ebooks, we decided that indeed the quality of the ebooks was quite high and that we could use these as our ground-truth files in the evaluation tool. The following three ebooks were used in the evaluation.

1. Een schot in de lucht (Koolhaas, 1962)
2. Bericht uit het hiernamaals (Vestdijk, 1964)
3. De weerspannige naaktschrijver (Geel, 1965)

Then, we let the ocrevalUAtion tool generate a CER and WER value for both a comparison between the OCR source file and the ebook and a comparison between the OCR file processed by our algorithm and the ebook. These values can be found in *Table 1*. Note that a higher value of the CER or WER measures indicates a larger difference between the two compared texts. Therefore, lower values after the correction denote an improvement.

All error rates are reduced after the correction algorithm has been applied. This means that for each of the books that we tested, at least some words are changed for the better, reducing the value of the CER measure. These corrected words are then recognized by the WER calculation as being the correct word in the correct place, partially explaining the reduction in this measure as well.

Table 1: CER and WER before and after application of the correction algorithm for each of the ebooks.

ebook	CER			WER		
	Before correction	After correction	Difference	Before correction	After correction	Difference
Een schot in de lucht	0.34	0.24	-0.10	1.29	1.09	-0.20
Bericht uit het hiernamaals	0.72	0.67	-0.05	2.32	1.51	-0.81
De weerspannige naaktschrijver	0.34	0.12	-0.22	0.59	0.57	-0.02

Improvements in the CER rates range from 6.1% to 64.7% as far as the test set goes. The differences in the WER are relatively smaller but still considerable, ranging from 3.4% to 35.0%. Because of the relative unavailability of free and relevant ebooks, more extensive testing was not an option. It stands to reason however that the results could change drastically per book, because the quality of the OCR source texts varied quite a bit between books as well.

4. Discussion

The objective of this project was writing an algorithm that finds and repairs mistakes in a text generated with OCR. We used a combination of a lexicon-based model and a context-based model, together with some hard coding. We found that application of our code improved both the word error rate and character error rate in all three tested texts.

4.1 Critical view on the used evaluation

For the evaluation, 'correct' digital versions of three texts were used that we found on the internet. We found that these texts differed from the original text that the OCR was applied to. These differences mostly consisted of different kinds of quotation marks and different use of hyphens. Therefore, we had to slightly alter the 'correct' texts as well, to be able to perform an accurate test.

There are other evaluation methods available and more commonly used. However, these were difficult to use for this specific project because we didn't have a labeled text or even a perfect ground truth text, as discussed before. An example of alternative evaluation methods is the F-score, as used in Reynaert (2004). The F-score is a formula consisting of the recall and the precision. Other measures include calculating the AUC (area under the curve) for an ROC-curve consisting of the true and false positives (Reynaert, 2008), or the weighted sum of the Levenshtein distance (Chiron et al., 2017). Using the edit distance, as described by Kanai et al. (1993), is similar to the evaluation measure we used in the end. Considering the niche this algorithm is applied to and the limitations the project imposed, the use of the evaluation tool is justified. With no restrictions to time and resources, the most accurate test results could have been attained by using texts annotated by human(s).

4.2 Critical view on the process

We've come up with the optimal code by testing several versions of our code, each with different parts of code turned on or off. However, we didn't test all possible combinations. Perhaps this could have led to overseeing an even more suitable possibility.

Secondly, the text with which we did the testing was composed by us, using several kinds of frequent mistakes we spotted in the books. We composed our own text instead of using an actual book or a part of an actual book because we wanted to have a shorter running time, diverse mistakes and a high error density, since we had to run this program each time we

changed our code and we needed a clear overview of which mistakes were fixed by our code and which were not. Our text contained a higher error rate than the books and since we manually added mistakes, it is not a perfect representation of the books we applied our code to. This might have led to overseeing certain types of mistakes that we did not get to fix.

In addition to our main solutions, we also hardcoded some common mistakes. The decision of which common mistakes to fix was based on what mistakes stood out to us and could be easily fixed in just a few lines of code. We did not do a count of which mistakes were most common. This may be a good idea to do in future attempts at OCR correction.

Lastly, since we did not have enough correct texts at our disposal, we have not statistically tested if the error rates improve significantly, though this is advisable to any future attempts at OCR correction.

4.3 Critical view on the used code

Our code is far from perfect. When an incorrect word is still a Dutch word in our wordlist, our program lets it slide. And vice versa, when a correct word is not in our word list and is not recognized as a two-word-compound, our code still tries to fix it. Among others, this happens to three-word-compounds, foreign words, old fashioned words, rare words and some conjugations.

A more serious flaw of our current program is that we tried to reduce running time by letting all words with a capital letter slide. We did this because a lot of names, like names of cities or historic figure, were not in our lexicon nor in the list of names and therefore being processed for correction, even though they were either correct or irrelevant. Ignoring all words with a capital letter means that they never enter the roBERTa and Levenshtein part of our program and may often stay unfixed. These parts of the program may not offer a benefit anyway, however. In short, we currently do not know what the exact impact is on the performance and further testing is needed.

Next, our code fixes only certain types of mistakes regarding too little or too much white spaces. More time or perhaps a trained model is needed to fix more (or all) of the white space problems.

Finally, our algorithm is far too slow. It took on average about three hours to run an entire book. Most of the time was spent on computing related words in the Levenshtein Distance

Matrix. Due to the scope of our project and our client's deadline, it was not possible for us to dive into the issue of fixing this. However, with future attempts at OCR correction we recommend giving this issue some more thought. One of the ways our algorithm could be sped up is further reducing the amount of 'correct' words unnecessarily entering the language model part of our program.

4.4 Other OCR correction methods

More options to do post-OCR correction are possible, like using deep learning methods[5] or probabilistic OCR, using pretrained models (i.a. Salimzadeh, 2019). The reason we chose the method used is because it suited our abilities and expertises and few Dutch pre-trained models and Dutch databases were available.

4.5 Uses of our code

Our code is written specifically to be applied to Dutch novels from the 1960's. It is not possible to apply it to other languages, since we are using a Dutch pretrained model, and there are some language specific solutions to language specific problems. It might however be possible to use our code for other Dutch text genres.

The only part of our code that is really specific to the 1960's is the part that identifies words with K's instead of C's as 'correct' (since this is old spelling). The parts that fix problems with page numbers and end-of-line characters are novel specific. These parts can easily be turned off, however. Then, our code might be applicable to other Dutch texts than just Dutch novels from the 1960's.

Essentially, it is important to realize that our code is based on common OCR mistakes we identified by hand in the novels that were to be corrected. Other OCR methods may lead to other kinds of mistakes. Therefore, we conclude that it is inadvisable to use our code blindly, without checking the kind of errors your OCR has made. The method we used could still be an inspiration and parts of our code could be copied, but it would be wise to use a tailor-made version of our code, specific to the kinds of mistakes the used OCR method makes.

[5] https://github.com/hs105/Deep-Learning-for-OCR

5. References

Bisong, E. (2019). Google colaboratory. In Building Machine Learning and Deep Learning Models on *Google Cloud Platform* (pp. 59-64). Apress, Berkeley, CA.

Carrasco, R. C. (2014, May). An open-source OCR evaluation tool. In *Proceedings of the First* International Conference on Digital Access to Textual Cultural Heritage (pp. 179-184).

Chiron, G., Doucet, A., Coustaty, M., & Moreux, J. P. (2017, November). ICDAR2017 competition on post-OCR text correction. In 2017 14th IAPR International Conference on Document Analysis and Recognition (ICDAR) (Vol. 1, pp. 1423-1428). IEEE.

Delobelle, P., Winters, T., & Berendt, B. (2020). Robbert: a dutch roberta-based language model. arXiv preprint arXiv:2001.06286.

Geel, R. (1965). *De weerspannige naaktschrijver.* Amsterdam: De Bezige Bij. Retrieved from: https://www.dbnl.org/tekst/geel005weer01_01/

Govindan, V. K., & Shivaprasad, A. P. (1990). Character recognition—a review. *Pattern recognition*, *23*(7), 671-683.

Kanai, J., Nartker, T. A., Rice, S., & Nagy, G. (1993, October). Performance metrics for document understanding systems. In Proceedings of 2nd International Conference on Document Analysis and Recognition (ICDAR'93) (pp. 424-427). IEEE.

Koolhaas, A. (1962). *Een schot in de lucht*. Amsterdam: Stichting Collectieve Propaganda van Het Nederlandse Boek. Retrieved from: https://1lib.nl/book/4065413/d954bb?id=4065413&secret=d954bb

Lasko, T. A., & Hauser, S. E. (2000, December). Approximate string matching algorithms for limited-vocabulary OCR output correction. In *Document Recognition and Retrieval VIII* (Vol. 4307, pp. 232-240). International Society for Optics and Photonics.

Ma, E. (2018, november 24). Correcting text input by machine translation and classification. Retrieved from https://towardsdatascience.com/correcting-text-input-by-machine-translation-and-classification-fa9d82087de1

Mei, J., Islam, A., Moh'd, A., Wu, Y., & Milios, E. (2018). Statistical learning for OCR error correction. Information Processing & Management, 54(6), 874-887.

Reynaert, M. (2004). Text induced spelling correction. In *COLING 2004: Proceedings of the 20th* International Conference on Computational Linguistics (pp. 834-840).

Reynaert, M. (2008, May). All, and only, the Errors: more Complete and Consistent Spelling and OCR-Error Correction Evaluation. In *LREC*.

Salimzadeh, S. (2019). *Improving OCR Quality by Post-Correction* (Doctoral dissertation, Universiteit van Amsterdam).

Vestdijk, S. (1964). *Bericht uit het hiernamaals.* Amsterdam: De Bezige Bij. Retrieved from: https://www.dbnl.org/tekst/vest002beri01_01/

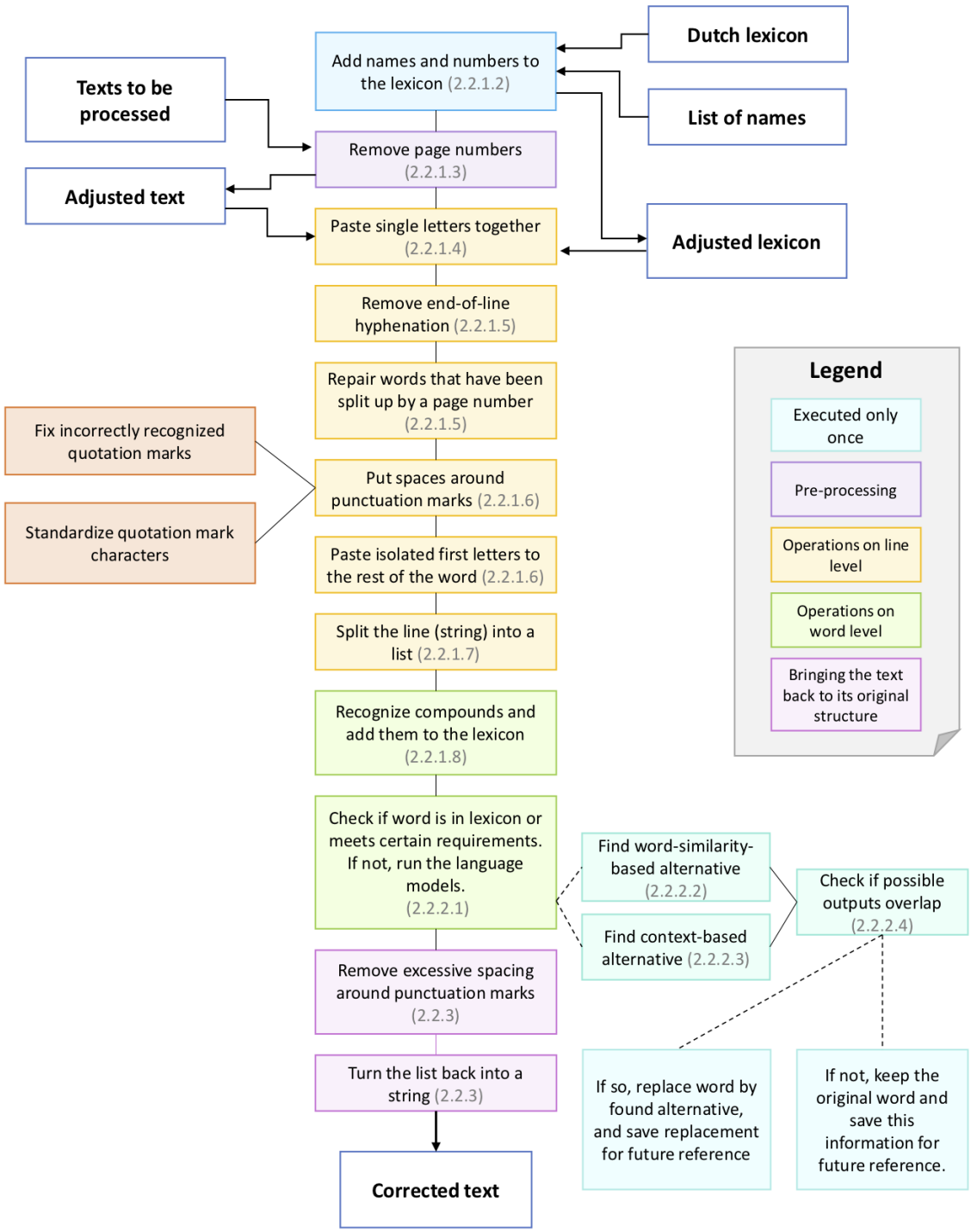
Volker, B., & Smeets, R. (2020). Imagined social structures: Mirrors or alternatives? A comparison between networks of characters in contemporary Dutch literature and networks of the population in the Netherlands. *Poetics*, *79*, 101379.

Yoo, A. B., Jette, M. A., & Grondona, M. (2003). Slurm: Simple linux utility for resource management. In Workshop on job scheduling strategies for parallel processing (pp. 44-60). Springer, Berlin, Heidelberg.

6. Appendices

Appendix I: Pipeline of the algorithm

Below, you find a visualization of the algorithm. Technically, output from each function is sent to the next function via a central, overarching function list. For the sake of clarity we opted to visualize the process like this, however. Finally, the numbers at the bottom of some of the cells correspond to the section numbers of this report, where these functions are explained.



Appendix II: Test text

Explanation

The text below was used to test certain configurations of functions in the algorithm and to find out which of these configurations yielded the best result (i.e. the greatest improvement). It must be noted that this text has an error rate that is considerably higher than averagely found in the books. This is because the text is composed of as many errors as we could fit in.

Text

H i j stond op en liep naar de huiskamer. Zijn vader, in de stoel bij de schoorsteen, het hoofd bijna op de knieen, schrok op.
‘Mm?’
‘We zijn nog niet weg/ fei Akijn*
‘Moeten jullie nog njet weg?’ vroeg zijn vader.
‘Moe is nog niet klaar4
‘Mm?’
COp de knieën van zijn vader lag het rode Engelsegrammaticaboek,
een slap beduimeld deel dat h em
troostte onder de schemerlamp en hielp bij het inslapen.
Na zijn kandidaatsexamen had zijn vader hem
steeds meer lastig gevallen met grammaticale vragen,
waarbij hijhadgemerkt dat zijn rüiterlijk bekend ‘dat
weet Tk niet’ zijn vader groot plezier deed. Het was
een soort band geworden waar-
van zijn vader zich zelfs
tot diep in een visite bewust was. CDan stak hij guitig
de v i n g er op en riep: ‘My sister Ann is a pretty girl’,
zijn mond in de meest gummieachtige bochten wringend
omdat hij dacht dat dit moest bij een vreemde
taal. ‘This lake contains all sorts of fish,’ riep Akijn danioterug i n h e t glunderende gezicht.
Men was er gewend
aan geraakt, te meer omdat h ij meedeed; als hij zijn
mond maar open-deed zei iedereen al Amen.
Boe h eet d e hoofdpersoon?Akun*
I k Ücht d at even íoe.

Appendix III: Choosing the best methods

As described in section 2, we tested different methods on our test text before choosing a definite method. Firstly, we had to decide what we wanted our program to do when there was no match between the suggestions of the Bert model and the Levenshtein matrix. The options were: choose the word the Levenshtein part of the program suggests, use the word the Bert part of the program suggests, or leave the old word as it was. As can be seen in Table 2a, the best option was to leave the old word as it is, instead of changing it into something that might be wrong. This is especially the best option for the character error rate.

Table 2a: CER and WER on our test text giving the different methods priority.

	CER	WER
Test text before processing	4,07	22,89
Levenshtein priority	3,71	14,93
Bert priority	9,28	16,42
Leave wrong words alone	2,92	15,92

Then, we tested if it was best to add additional dictionaries of foreign languages. This can be seen in Table 2b. Because the results were better without adding additional languages to the lexicon, we decided to not include those.

Table 2b: CER and WER on our test text with and without adding foreign languages to the lexicon

	CER	WER
Test text before processing	4,07	22,89
Foreign languages	2,92	13,93
No foreign languages	2,83	13,43

Appendix IV: Manual for setting up a suitable virtual environment

Below, you find a stepwise guide to setting up a virtual environment in which the algorithm can be run. Once logged in, execute the following steps:

1. Create the virtual environment itself, with the command `python3 -m venv /home/username/path/to/environment`.

2. Install the following libraries or modules with the command `pip3 install package_name`.
    a. torch
    b. Flax
    c. TensorFlow (version 2.0 or higher)
    d. transformers

3. Update the following package with the command `pip3 install --upgrade package_name`.
    a. scipy

4. Make sure that the following files are uploaded to the virtual environment folder:
    a. All text files (`.txt` format) that need to be processed
    b. A text file (`.txt` format) containing the file names of all files that need to be processed (called `boeknamen.txt`)
    c. The Dutch lexicon (called `wordlist.txt`)
    d. A list of name variants (called `naamsvarianten.txt`, as explained in section 2.2.1.2)
    e. The python script of the algorithm
    f. A shell script executing the python script (not required, but strongly advised because of long running times)