Rule-based Text Correction

Bachelor Thesis
Faculty of Science, University of Bern

submitted by

Nils E. Neeb

from Brig, Switzerland

Supervision:

PD Dr. Kaspar Riesen
Corina Masanti
Institute of Computer Science (INF)
University of Bern, Switzerland

Abstract

In the corporate publishing process, maintaining consistency in style and vocabulary is essential for clear communication and a strong brand identity. Companies often rely on proofreaders to achieve this cohesion. This thesis explores the development of a program designed to support proofreaders by automating aspects of the correction process.
The project is rooted in the field of Artificial Intelligence (AI), specifically within the subdomain of Natural Language Processing (NLP). It focuses on text correction tasks, such as style and spelling correction. The primary aim is to partially replicate the work of proofreaders by applying predefined linguistic and stylistic rules. The methodology involved designing rule-based models that enforce uniformity across texts.
The results of the empirical investigation indicate that the program effectively supports style adherence and error correction. It has been statistically identified that the developed program can automate 43% of the corrections made by proofreaders, provided that sufficient rules are defined. This demonstrates that the primary objective of this thesis has been met and that the system can enhance the current correction process.

Acknowledgements

I would like to thank PD Dr. Kaspar Riesen for his guidance during this project and for providing helpful feedback throughout the writing process. I also want to acknowledge PHD student Corina Masanti, who offered valuable inputs and practical advice during all stages of the work.
Finally, I want to thank my family, especially my parents and my sister, for their support and patience.

Contents

Chapter 1

Introduction

In the corporate world, the utilization of proofreaders has become an integral component of the publishing process, particularly in the context of ensuring the uniformity of style and vocabulary across various documents. This practice is employed by numerous companies to maintain the cohesion and clarity of their communication, thereby contributing to the enhancement of their overall brand, corporate identity, and reputation.

The program created in the course of this thesis aims to aid proofreaders and improve the correction process by increasing productivity and decreasing time expenditure. This thesis is situated within Artificial Intelligence, which is a part of Computer Science that focuses on replicating aspects typically associated with human intelligence, such as learning, reasoning, problem-solving, and decision-making. One subdomain of Artificial Intelligence is Natural Language Processing, also called NLP. The relevant structure of Computer Science is shown in Figure 1.1. The present thesis is located within the domain of NLP, known as Text Correction. In this domain, the program is capable of executing tasks across various subfields, including Style Correction and Spelling Correction, if appropriate rules have been defined.

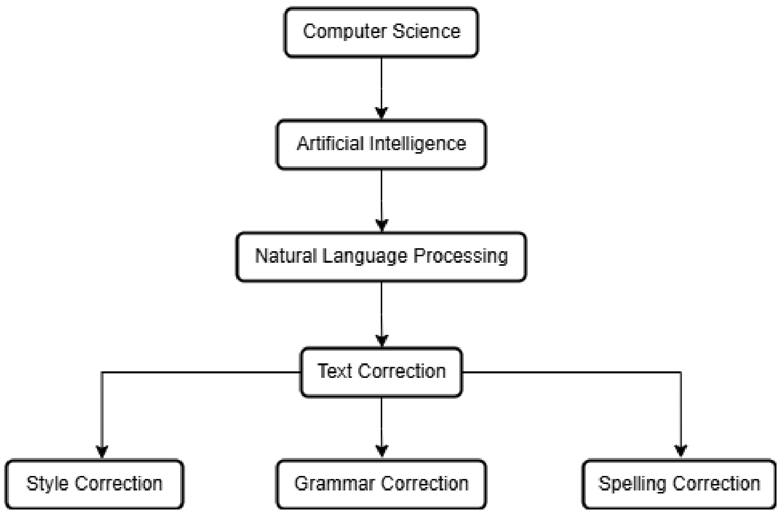


Figure 1.1: Hierarchical Diagram of Computer Science Domains

The primary objective of this thesis is to develop a program with the ability to, at least partially, perform the tasks of the proofreaders. Ensuring the consistent application of the style and wording guidelines provided by companies is a key component of the process.
The remainder of this thesis is structured as follows (see also Figure 1.2). After the introduction, the first chapter is dedicated to the scientific background and previously existing research in this field. Following this, the theoretical foundation of the rules, implemented in the program, is discussed. After the theoretical background is set, the structure as well as the workflow of the correction program are described. After the description of the implementation, a thorough examination of the results ensues. There, the setup of the testing and the results are described and interpreted.

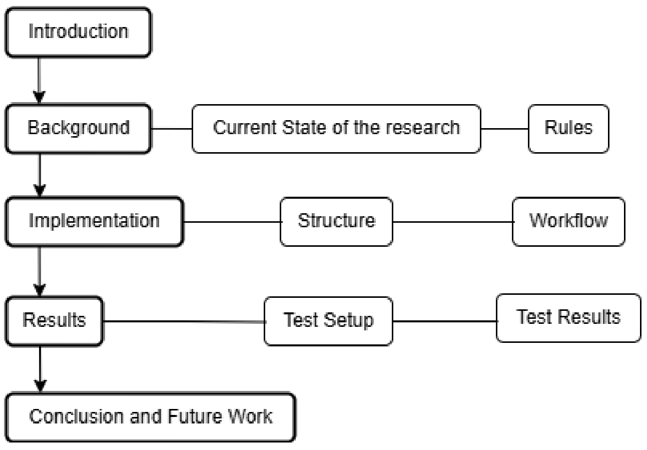


Figure 1.2: Structural Diagram of this thesis

Chapter 2

Background

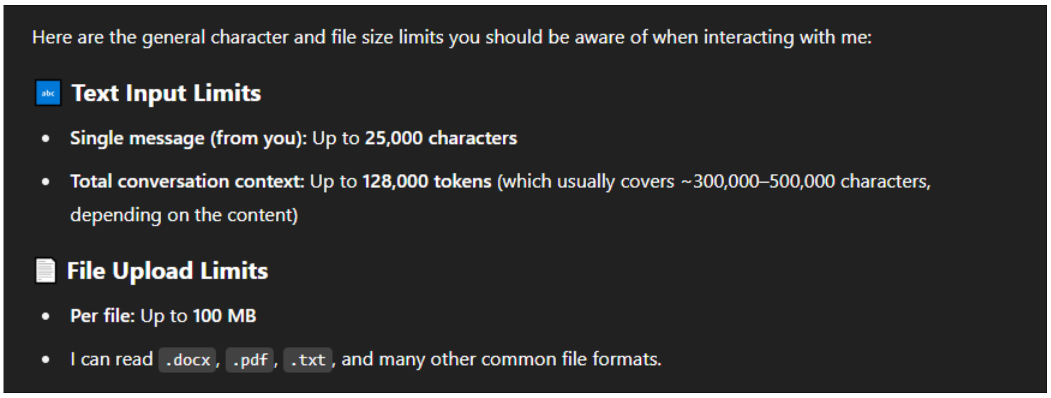
2.1 Current State of the Research

Rule-based text correction approaches have been used before, but primarily for the correction of grammatical and syntactical errors in specific languages. One example of this would be the work of Byun et al. In their research paper, Byun et al. [1] describe a rule-based approach for correcting spelling errors in spoken-style messages in Korean text like short messenger service (SMS). They extracted the rules automatically from the correction corpus and applied the extracted rules to new sentences. With their system, they could reach a precision of 79.04%.
Language-specific correction programs are not only present for languages with comparatively large numbers of speakers, as Adruzi et al. [2] show with their morphological analysis-based method for spelling correction for Basque. It uses a two-level morphological analyzer to decompose words into morphemes and verify their grammatical validity. This process involves mapping canonical forms to written forms through language-specific rules. When a word is not recognized, the system distinguishes between orthographic errors and typographical errors. It then applies appropriate correction strategies. This language-specific approach enables the system to accurately recognize and correct complex word forms that general-purpose spell checkers often misidentify.
With the paper "Automatic Rule Acquisition for Spelling Correction", Mangu et al. [3] focused more on the automatic learning of rules from a larger one-line text corpus. The captured rules are then stored in a small set of rules. They refrained from using opaque features and weights in favor of easily understandable rules, which allowed them to exploit human intuition and thereby enhance the comprehension and refinement of the system's acquired knowledge.
In contrast to Byun et al., Milkowski et al. [4] chose a language-independent tool that uses a rule-based approach in order to detect grammar, style, and consistency

issues. The main focus of their system was on flexibility and extensibility, allowing rule development without a full formal grammar or deep syntactic parser. They used a combination of declarative XML rules and Java-based rules.
In their 2019 study, Alamri et al. [5] developed Sahah, an automatic correction system designed to address spelling errors in Arabic texts written by dyslexic individuals. The system combines rule-based pre- and post-processing with a compression-based language model (Prediction by Partial Matching) and dictionary lookup for error detection and correction. It targets common dyslexic writing issues, such as character repetition, letter confusion, and word segmentation. Evaluation using the Bangor Dyslexic Arabic Corpus showed Sahah achieved high precision (89%) and 81% accuracy, outperforming general-purpose tools like Microsoft Word and Farasa. This work demonstrates the effectiveness of tailored, rule-based NLP solutions for low-resource languages and special user groups. Their work shows an interesting approach to creating more inclusive correction systems.
Another approach to correcting mistakes in text makes use of statistical methods. Golding et al. [6] propose a context-sensitive spelling correction method focused on real-world errors where the incorrect word is a valid dictionary word but contextually inappropriate. Their system, called the Contextual Spelling Corrector (CSC), uses the Winnow algorithm, a linear classifier well-suited for handling high-dimensional feature spaces with many irrelevant attributes. Each confusion set (e.g., to, too, two) is treated separately, and decisions are made based on the contextual words surrounding the target. The approach significantly outperforms traditional methods such as n-gram models and Naive Bayes classifiers. Their work emphasizes the importance of context for text correction programs.
An additional illustration of a rule-based correction system that has been developed specifically for second-language learners is provided by the work of Lee et al. [7]. Their approach is centered on the correction of grammatical errors in English sentences composed by non-native speakers. Rather than attempting to directly parse ill-formed input, their system generates a lattice of possible corrections using a set of rule-based transformations and then selects the most probable corrected sentence using an n-gram language model. A stochastic context-free grammar (SCFG) parser is subsequently used to re-rank the candidates based on grammatical plausibility. The system was evaluated in a task-oriented domain (airline travel) and achieved a correction accuracy of 88.7% on parsable sentences, demonstrating the potential of combining rule-based generation with statistical ranking for robust grammar correction.
A different approach was proposed by Zhou et al. [8]; they developed a grammar correction system that employs a classification-based approach, wherein different

grammatical error types are addressed by dedicated classifiers. The system extracts linguistic features to predict the correct grammatical forms, focusing on errors such as subject-verb agreement and article usage. This targeted method enhances precision while preserving interpretability, thereby presenting a transparent alternative to neural models.
In contrast, Hu et al. [9] developed a misspelling correction system that combines edit distance algorithms with contextual embeddings from pre-trained language models, such as BERT. The system generates candidate corrections via edit distance and employs BERT's masked language modeling to select the best fit in context. This hybrid approach has been demonstrated to be effective in the handling of both obvious typos and subtle context-dependent errors. On benchmark datasets, the model achieved an F1-score of 91.4% on the Twitter misspelling dataset and 88.7% on the Wikipedia misspelling dataset, outperforming traditional spell checkers. This finding underscores the efficacy of integrating symbolic and contextual information for effective misspelling correction.
Another example of a language-specific rule-based correction approach is the work of Óladóttir et al. [10]. With Icelandic being a low-resource language, their work is a key component of the Icelandic government's strategic 5-year Language Technology Program for Icelandic. In their thesis, Óladóttir et al. used a rule-based modular system design consisting of three main modules: the tokenizer, the morphological tagger, and the parser. The tokenizer is responsible for splitting the input text into tokens such as words or punctuation. Next, the morphological tagger is used to assign part-of-speech (POS) tags and morphological features like case, number, gender, or tense to each token. Lastly, the parser performs a syntactic analysis, building a shallow parse tree of the sentences. The tree pattern matching rules are then used to detect grammatical errors.

As the research mentioned before shows, a rule-based approach can yield good results and can increase the understandability of the program. Therefore, a rule-based approach was chosen for this project. This also ensured consistency with applying the covered rules. To illustrate the problems that can arise from the use of generative artificial intelligence, ChatGPT will be used as a representative for this type of non-local artificial intelligence. In the following paragraph, the previously mentioned problems will be discussed.
There are two main problems that can be prevented by using a rule-based approach instead of using AI. The first problem is the character and token limit that many artificial intelligence systems have.
In Figure 2.1, one can see the response of ChatGPT using the model GPT-4-turbo,


Here are the general character and file size limits you should be aware of when interacting with me:

Text Input Limits

- **Single message (from you):** Up to **25,000 characters**
- **Total conversation context:** Up to **128,000 tokens** (which usually covers ~300,000–500,000 characters, depending on the content)

File Upload Limits

- **Per file:** Up to **100 MB**
- I can read `.docx`, `.pdf`, `.txt`, and many other common file formats.

Figure 2.1: GPT-4-turbo describing its own limitations

when asked about the limits of the current model. Some of the texts that companies want to have corrected go beyond the scope of GPT-4-turbo. This means that these texts can not be corrected consistently using such models. This may manifest as semantic shifts or the omission of expected rule applications.
The second major concern from using non-local AI models is security. Some models may use inputs from users to train. If the documents, that need to be corrected, contain sensitive information or private information, the users may not want them to be used in the training of AI models. Another advantage of not using AI models is the simplicity. The rules are much more easily understood by humans than the architecture with weights and other parameters of an artificial intelligence. In the next sections, the rules that the system supports are described.

2.2 Rules

2.2.1 Rule Types

The rules used in the program are separated into three main types with multiple subclasses. These types are based on the underlying mechanism applied to these rules. Rules of Type 1 replace a word, a list of words, or single characters with other words. Type 1 contains two sub-rules. Type 2 rules format specific numbers, such as dates, times, and currency amounts. This type contains three sub-rules. Lastly, Type 3 rules are for replacements in .docx files, where the section style of the location of the word that should be replaced decides whether to replace the word or not. Type 3 has only one sub-rule. The subsequent sections provide a thorough exposition of these rules and the syntax of their storage in the rule files.

Type 1

Rules of Type 1.1 are stored as follows:

```
(list):(replacement):()
```

The first bracket contains a list separated by a comma of at least one word. The second bracket contains a single word. The word in the second bracket is what the words in the first bracket are replaced with. The third bracket is there for consistency reasons. It allows for a uniform handling of all rules of Type 1.

Rule Type 1.2 follows a similar pattern for storing information. This allows for all rules, following this syntax, to be handled uniformly.

```
(character):(replacement):(checklist)
```

The content of bracket one is a single character. It is meant for special characters like % or &. Bracket two contains, similar to Type 1.1, a word that the character in the first bracket is replaced with. The main difference to Type 1.1 can be seen in bracket three. It contains a list following the same syntax as the list in the first bracket, meaning words separated by a comma. The words in the text that contain the character of the first bracket are looked up in the list stored in the third bracket. If the list does not contain the word from the text, then the character is replaced with the content of the second bracket. If the list contains the word found in the text, then the character does not get replaced.

Type 2

Type 2 rules use a different storage syntax than Type 1. All three subtypes use a single square bracket to store all the relevant information. The first three characters are in capital and are used to denote the type of rule. The rest of the content contains the target format, that the corresponding numbers should be formatted to.
The first subtype is Type 2.1. This type is used for currencies. Its notation looks as follows:

```
[NUMformat]
```

The first three characters, as described before, denote the type (in this case, Type 2.1). The second part is a placeholder that will be used for all rules of Type 2. In rule type 2.1, the format is denoted in Format Specification Mini-Language.

Rule Type 2.2 is used to format dates. Its notation is similar to the notation of Type 2.1:

```
[DATformat]
```

The type of rule is defined by the first three characters. The format in Type 2.2 is written as a standard datetime format. This makes it possible that the second part can be used directly when formatting dates with datetime.

The third and last type is Type 2.3. It is used to format times. Here, the format of the rule is similar to the last two rules of Type 2:

```
[TIMformat]
```

The format in Type 2.3 is, as in rule Type 2.2, in datetime notation. This means that the format can be directly used and does not need to be adjusted to be usable.

Type 3

Type 3 contains only one rule type, and it is only applicable in .docx documents. This type of rule shares similarities with rule Type 1.1 and rule Type 1.2. It is stored as follows:

```
{list}:{replacement}:{style}
```

The first bracket contains a list of words, as in rule Type 1.1, that are separated by a comma. These words are replaced with the replacement word in the second bracket. This only occurs if the section of the text is of the style denoted in bracket three. Microsoft Word offers a variety of styles that can be used to quickly format a text section. Examples of a style in Microsoft Word are `title` or `subtitle`. The dependency on style is what differentiates this rule type from any other previously described rule.

2.2.2 Usage and Examples

The rules are stored in a separate .txt file. This allows the user to reuse the same rules without having to manually add them again for each use. Another advantage of having the rules stored in permanent files is that the rules can be standardized for all proofreaders, with the possibility of having a central administration of the rule files. Additionally, rule files can be easily expanded, so that they can evolve and change with the companies for which the rule files are used. Rule files can contain any number of rules. The modular design of the rules enables their representation of a wide range of modifications, as illustrated in Table 2.1. Through the mostly uniform syntax of the rules, the addition of new rules does not require changes in any preparation sequence, which is used to transform the rules into rule objects.

Rule Type	**Generalized**	**Example**
Type 1.1	(list):(replacement):()	(cash,bills):(money):()
Type 1.2	(character):(replacement):(checklist)	(&):(and):(H&M,M&M)
Type 2.1	[NUMformat]	[NUM.2f]
Type 2.2	[DATformat]	[DAT%d.%m.%y]
Type 2.3	[TIMformat]	[TIM%H%M]
Type 3.1	{list}:{replacement}:{style}	{cash,money,bills}:{funds}:{title}

Table 2.1: Generalized syntax and examples for each rule type

In Figure 2.2, the general application process of the rules can be seen. First, candidates that are usually denoted in the first bracket for Type 1 and Type 3 rules are found. For Type 2 rules, these candidates, which are numbers, need to follow a specific syntax. Examples of this syntax are common date or time formats, such as the following examples:

```
01.01.2000
16:00
```

Once an appropriate candidate is found, it is checked if certain rule-dependent conditions are met. If that is the case, then the correction is made according to the rule. This process is described in more detail in Section 3.1.

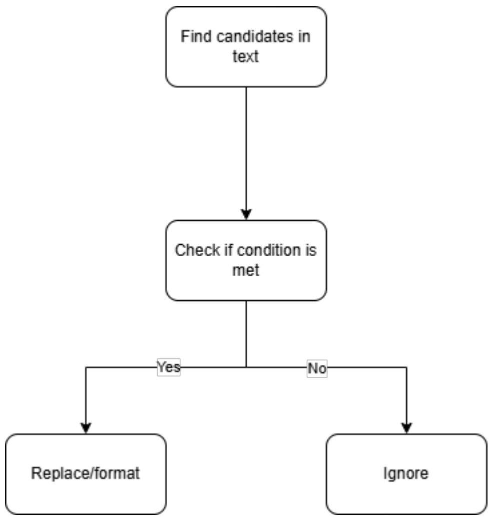


Figure 2.2: Simplified rule application process

Chapter 3

Implementation

3.1 Structure

To run the algorithm that corrects the given text, two things need to be present. Firstly, the rules need to be entered and stored in an object of the storage class, and secondly, the text needs to be stored.
The program consists of five main parts.

- Adapter
- RuleParser
- Applier
- Rule
- Storage

These classes are developed using Python and make use of several standard and third-party libraries. Built-in modules such as `string`[1], `re`[2], `datetime`[3], and `os`[4] were used for core functionality like text processing, regular expressions, time management, and file system operations. The `shutil`[5] module was employed for advanced file handling tasks, such as moving or copying files. For parsing and interpreting natural language dates, the third-party library `dateparser`[6] was integrated. The user interface was developed using `tkinter`[7] and its `ttk`[8] extension to create

[1]https://docs.python.org/3/library/string.html
[2]https://docs.python.org/3/library/re.html
[3]https://docs.python.org/3/library/datetime.html
[4]https://docs.python.org/3/library/os.html
[5]https://docs.python.org/3/library/shutil.html
[6]https://dateparser.readthedocs.io/en/latest/
[7]https://docs.python.org/3/library/tk.html
[8]https://docs.python.org/3/library/tkinter.ttk.html

a basic but functional GUI. Additionally, the `python-.docx`[9] library was used for reading and writing Microsoft Word documents.
All of these parts are needed in order for the program to function efficiently. In the following sections, all parts will be discussed in detail.

Adapter

The adapter is the first part of the program that interacts with the raw text imported from .txt or .docx files. As shown in Figure 3.2, the rules and the text first run through the adapter.

In the following section, the transformations of the text are described.
The first step in the adapter is a transformation of the given text, which is entered as a string, into a raw string. This means that escape characters are now visible and do not have an influence on the formatting, which allows the system to interact with them. As a result, the raw escape characters will exemplarily look as follows:

```
\n
\r
```

Next, the raw escape characters are replaced by placeholders to prohibit them from affecting the string manipulations. These placeholders are unique for each type of escape character. They all follow a pattern that only changes in the last part. The escape character gets replaced with a space followed by:

```
-escape-character-
```

At the end of this string, the escape character and another space are added so the escape characters do not interfere with the correction later on when the text gets separated by a space. Based on the previous examples, this leads to the following results:

```
␣-escape-character-\n␣
```

```
␣-escape-character-\r␣
```

After all the escape characters are replaced with their according placeholder, the text is returned to the main method to be stored in a storage object.
The handling of the rules does not necessitate the presence of escape characters for subsequent transformations. Therefore, after loading the rules as a raw string, the

[9]`https://python-docx.readthedocs.io/en/latest/`

escape characters can simply be deleted and replaced by a semicolon. The resulting text is then split by semicolons and stored in a list.
Subsequent to this, each element in the sequence is examined to ascertain whether it contains one of the three bracket characters employed to signify the rules, which are: `(` for rule Type 1,`[` for rule Type 2,`{` for rule Type 3. If a position contains one of these brackets, a corresponding number (1, 2, or 3) is appended to the string based on the bracket type, in order to facilitate parsing. Then this string is given to the RuleParser as an argument. The RuleParser returns a rule object based on the given argument, which is then stored in a separate list. After every position of the list containing the string given at the start is checked and, if needed, given to the RuleParser, then the second list, containing all the rule objects, is returned to the main method.

RuleParser

The RuleParser is the second and last part of the transformation of the rules from a raw text file into usable rules. The input is the transformed string from the adapter. This string, for example, may look like this:

```
    1(word1,word2):(word3):()
```

Initially, the number at the beginning of the string is removed again and used to determine how the rule needs to be parsed. After this, the string follows the rule format described previously.
For rule Type 1, the string is split first by colon, which results in three separate parts. Then, in each part, the brackets are removed, and they get split again by a comma. This returns a list with usually multiple elements for the first part. The second part is a list with only one element, and the third part of the rule may contain elements. After this split, the third part of the rule is checked. If the result contains any elements, then the rule is of Type 1.2, and otherwise it is of Type 1.1. Lastly, a rule object with the parameters type, first part, second part, and third part is created and returned.

Applier

The applier is the main part of the program. It takes a storage object as an argument. The storage object needs to contain a list of rules and a text, as well as the section type if the text was given as a .docx file.
The applier first analyses the given text and subsequently verifies whether it contains only a single space and no additional characters. This can happen due to the

way runs are extracted from .docx files. The rules are then sorted by rule type in descending order. This sequencing is executed in order to ensure that the more general rules are applied later, once the required placeholders are in place. This is especially important for the formatting of numbers. The current Type 2 rules only edit specific numbers. However, if a rule is later added that formats all numbers, all dates, times, etc., must first be replaced with placeholders before editing the remaining numbers. If the placeholders were not in place, then all the numbers would be formatted. This has the potential to yield improper formatting, which can manifest in undesirable outcomes, such as the following:

```
16:00 h-> 16.00:00.00 h
24.12 -> 24.00:12:00
```

After the rules are sorted, the program iterates over the list containing all the rules. For each rule, the type is checked, and then, depending on the type, the rules are applied to the text. As outlined above, some rules utilize placeholders. Namely, rule Types 1.2, 2.2, and 2.3. These placeholders are replaced with the actual values once all rules from the list have been applied to the text. This ensures that no incorrect number formatting or replacement occurs.
In the following sections, it is discussed how each type of rule is applied in detail.

Type 1.1:
The first step of applying the rules of Type 1.1 is to check the content of the first bracket. If the first bracket of the rule only contains a single character, then this character can simply be replaced in the entire text without having to take any further steps to ensure correctness. The same is valid if the first bracket contains a space. Then the phrase contained in the first bracket can be replaced without any further steps. This approach allows the program to ignore punctuation, meaning that if the phrase or character is at the end of a sentence, it can still be detected and replaced.
However, if the first bracket contains words without any spaces, then the text is split by space. Then, every part of the new list is first cleaned. This means that all punctuation marks are removed. Since `Word` is not equal to `Word.`, this step is needed in order for the words contained in the first bracket to be detected correctly. Once these steps have been taken, it is iterated over all the splits from the text, and the words in the first bracket are replaced with the word in the second bracket. Following this, the text is joined back together and returned.

Type 1.2:

As previously stated, Type 1.2 uses placeholders. The initial step is to replace all words in the text that are contained in the third bracket with placeholders. The placeholders used here are numbered with letters and not with numbers as a means to eliminate the possibility of the placeholder being altered by rules of Type 2. These placeholders can, for example, look as follows:

```
whitelistaA
```

The placeholder consists of three parts. The first part is `whitelist`, which denotes what type of rule this placeholder is used in. That part is followed by a lowercase letter. This letter is changed (following the alphabet) to differentiate between the replaced words. The last part is an uppercase letter. This letter comes from a global variable that counts how often placeholders were used. This is needed because if there are two rules of Type 1.2, then the placeholders can be equal since the counter used in the lower case is not global and resets after Type 1.2 is applied. The replaced words are stored together with the placeholder in a dictionary, so that after all rules have been applied, they can be switched back. After the placeholders are present, the new text is handled similarly to the procedure Type 1.1, and the words in the first bracket are replaced by the word in the second bracket.

Type 2.1:
Type 2.1 replaces all numbers that have any abbreviation of the Swiss franc before or after them. To find the numbers that match this description, a regular expression (regex) is used. This regex currently contains the following variations of notations of the Swiss franc:

- Fr
- fr
- CHF
- Franken

This list can be expanded to other currencies if needed, but it requires the regex to be adjusted.

Type 2.2:
This type handles the formatting of dates. Similarly to Type 2.1, it uses a regex to capture all the dates and format them. The key difference in the handling of Type 2.1 and Type 2.2 is that, after the formatting of Type 2.2, the date is replaced by a placeholder. The placeholders used here have the same structure as the

placeholders for Type 1.2. The main difference lies in the word at the beginning. For this rule type, the word `date` is used to identify the placeholders. This word is followed by a lowercase letter, similar to the placeholders of rule Type 1.2, to differentiate between the replaced words. The uppercase letter comes from the same global placeholder counter to ensure that no two rules of the same type produce the same placeholders. The resulting placeholders can take, for example, the following form:

```
dateaA
```

Type 2.3:
Type 2.3 is handled similarly to Type 2.2. The regex is adjusted to detect times, and depending on whether the time is in a 12-hour format or a 24-hour format, a transformation between the two systems is needed. Type 2.3 is the last rule type that uses placeholders. The placeholders follow the previously discussed structure. The placeholders for this rule type start with the word `time`. The two letters that follow that word are utilized and derived in the same manner as previously stated. Here is an example of a placeholder for Type 2.3:

```
timeaA
```

Type 3.1:
Type 3.1 is specific for .docx files. The .docx files are separated into so-called runs. These runs are then edited separately. Each of these runs has a style type. Common style types for .docx files are:

- title
- subtitle
- heading 1

The information about the current style is stored in the storage object that the adapter receives as an argument. Initially, it is checked if the current style is the one defined in the third bracket. If this is the case, the text is handled exactly as described in Type 1.1. If it is not the case, then the found candidate does not satisfy all the criteria needed for this rule to be applied.

Rule Class

The rule class is needed in order for the applier to effectively apply the defined rules. It also simplifies the general handling of rules. Each rule object can have four attributes:

- first
- second
- third
- type

The first three attributes correspond to the brackets in which the rules are stored. Each rule has its type stored as a numerical value in the attribute `type`. The rules that follow the three-part storage syntax, for example, rule Type 1 and Type 3, transfer the contents of the brackets into the respective attribute. This means that the content of the first bracket is stored in the attribute `first`, etc. The attribute `third` does not contain any information for rules of Type 1.1.
Rules of the Type 2 only utilize the attribute `first` and `type`, where the target format and the type are stored.

Storage Class

A single object of this class is used for all different segments of the code to pass information over to one another. It has 10 attributes, which are:

- rules
- text
- doc_type
- file_path
- name
- style
- placeholder_counter
- dict12
- dict22
- dict23

The rules and text are stored in an object of this class at the beginning of the program, once the user selects the two files, which are the rule file and the text file. The attributes `doc_type`, `file_path`, and `name` are also stored while the user selects the files. The value of the attribute `style` is stored during the text extraction of the .docx file. `placeholder_counter` is the global variable that is used to ensure the uniqueness of each placeholder. The remaining three attributes are used to store dictionaries for the respective rules to enable an accurate replacement of the placeholders with the edited values.

3.2 Graphical User Interface

The Graphical User Interface (GUI) of the developed system consists of a vertical navigation bar on the left. The main content of the program is displayed on the right of the navigation bar. The document view can be seen in Figure 3.1. This is shown by the title in the center labeled "Document". The button in the center is used to upload a document. On the right side, the text marks the space where any outputs of the system are shown. This feedback is essential for users to understand whether the system is currently processing, what actions it is performing, and whether any input from the user is required. It also serves as feedback for the loaded documents. In the document view, a snippet of the document is displayed after uploading. The rules view is structured the same way. The navigation bar on the left stays there while the title and the text of the button change to "Rules" and "Load Rules". The output space on the left is used to display a list of all the rules contained in any uploaded rule file. The run view consists of a single button in the center and an output field on the right side, similar to the document and rules view. The output of the run view, after pressing the run button, is a snippet of the corrected version of the text previously uploaded.
The vertical navigation bar on the left consists of three buttons: Document, Rules, and Run. The first button, Document, switches to the document view, where a .docx or a .txt file can be uploaded. The button Rules switches to the rules view, where complete rule files in .txt format can be uploaded. The last button on the left, Run, can be pressed to switch to the run view.
The design of the GUI is simple in order to keep the amount of visual clutter low and not to distract from the core functionality of the System. Another advantage of the simplicity is that the usage of the system does not require any explanation. The program's interface is straightforward, requiring only a few steps to complete tasks, making it very efficient. The labeling, color schemes, and button placement are consistent throughout the GUI, making it easier for first-time users to use the system.

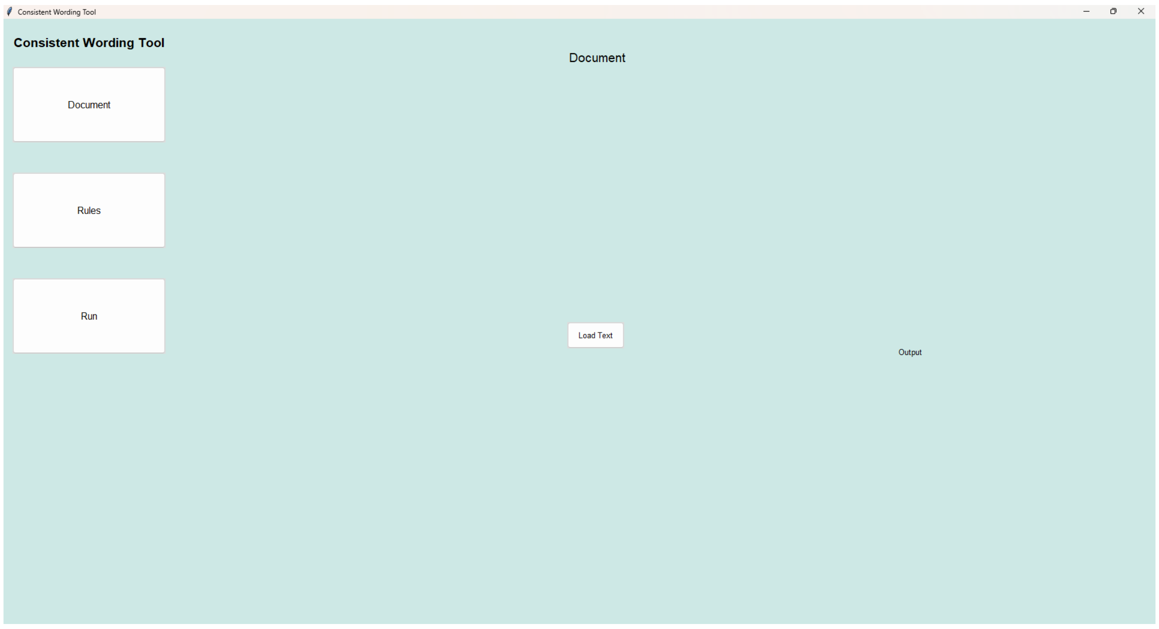


Figure 3.1: Screenshot of the current GUI

3.3 Workflow

Once the program has started, a storage object is created. The user then needs to input two files. It does not matter in which order these files are provided. There are two separate buttons in the GUI to input the different files. One of these files is the rules file. Once the rule file is entered, the rules automatically run through the adapter and the RuleParser. After that, the rule objects get stored in a list in a storage object. The text, on the other hand, only runs through the adapter before being stored in the storage object.
Once both files have been stored in the storage, the user can press a separate button to apply the rules to the text. How the rules are applied to the text is described in detail in Section 3.1. This process is illustrated in Figure 3.2.
The sequential structure of the workflow ensures that each rule type is applied in an order that prevents interference between dependent transformations. Rules that rely on placeholder substitution, such as those for dates and times, are executed early to isolate their formatting targets. This prevents subsequent rules from altering critical substrings that have already been processed. Once all the rules have been applied, the system iterates through the stored placeholders and replaces them with their corrected representations. This two-phase approach guarantees that formatting integrity is preserved throughout the process. Rules are applied iteratively and deterministically to ensure the reproducibility of results with identical inputs. The workflow incorporates rule-type-specific logic to enable tailored handling of simple replacements, context-based conditions, and style-dependent changes in .docx files.

All transformations are applied directly to an in-memory representation of the text, minimizing I/O overhead and facilitating efficient processing. The design allows for extensibility, enabling new rule types to be integrated without altering the workflow sequence.

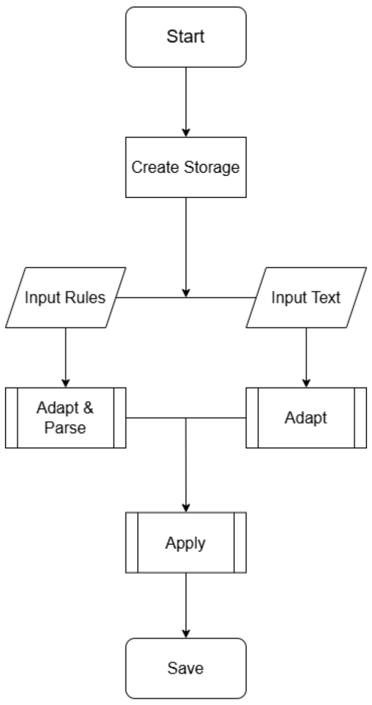


Figure 3.2: Workflow of the program

Chapter 4

Results

4.1 Test Setup

To evaluate the program, which was developed in the course of this work, real-world data provided by different companies was used. The data that was made available encompassed two types of files. Firstly, the companies provided guidelines on how they want their documents to be written. The guidelines contained directives about the formatting of numbers, dates, and times. In addition, the use of the respective company name as well as product names was regulated. This program was intended only to be used with German, therefore, files used for the evaluation are only in German. In the German language, inclusive language assumes a variety of forms. The majority of companies have established guidelines and rules regarding the implementation of inclusive language in their documents. Consequently, the guidelines contained directives on inclusive language. A significant component of the guidelines pertained to the general use of specific words, which were subject to regulation. In Table 4.1, examples of the previously mentioned types of directives are provided. In this table, "X" is used as a pseudonym for a company name. The examples shown there are of the same form as the rules in the guidelines.

Guideline type	**Example**
Formatting of numbers	For currency: Two digits after decimal point
Use of company or product name	Always write X AG and never only X
Inclusive language	Always use male form followed by *in
Use of specific words	Never use customers, instead use patients

Table 4.1: Examples of company guideline types and their formulation

The second type of document made available by the regarded companies consists of both corrected and uncorrected documents, which are corrected by the proofreaders. The provided documents range from one to four pages in length. The distribution of the document length is illustrated in Figure 4.1. All tested documents combined contain 42 pages of content.

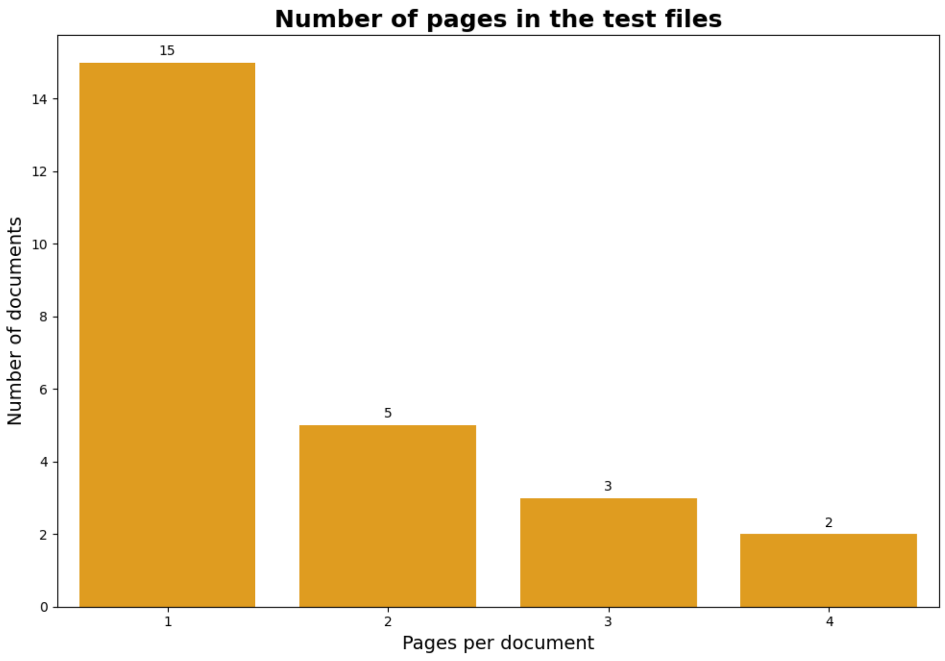


Figure 4.1: Distribution of documents according to their length

Before the testing process on the basis of the selected documents can be started, the guidelines of the companies need to be transformed into usable rule files. This ensures that the same rules are applied to all documents from the same company. In Table 4.2, the previous examples of guideline types from Table 4.1 are transformed into usable rules in the syntax defined previously in Section 2.2. The rule files created with this process consist of one rule in the defined syntax per line. The number of rules was dependent on the quality and the level of detail in the guidelines.

Guideline example	**Example rule**
For currency: Two digits after decimal point	`[NUM.2f]`
Always write X AG and never only X	`(X):(X AG):()`
Always use male form followed by *in	`(Mitarbeiter):(Mitarbeiter*in):()`
Never use customers, instead use patients	`(customers):(patients):()`

Table 4.2: Examples of guidelines and their respective rules

The documents used in the evaluation of the system were selected based on three criteria. The first criterion is language. A majority of the guidelines only contain corrections for German, therefore, the documents need to be in German too to ensure a sensible evaluation. The second criterion, corrections, refers to the presence

of corrections. The documents need to contain corrections made by the proofreaders in order to compare them to the corrections made by the system. Lastly, the system's capabilities were considered. This means that documents containing only changes that, e.g., alter the text's meaning, were not used for evaluation. After creating the rule files and selecting the documents used for testing, the first round of testing is done. This is done by running the program with all the files from each company and the respective rule files. After the files have been edited by the program, the number of corrections is counted. For each file used this way, the number of corrections the proofreaders made is also counted and used for comparisons.
Following the first test run, the rule files for each company are extended with additional rules. The rules that are added this way are not described in the guidelines of the different companies. The additional rules are derived from the documents used in the first test run. This process consists of analyzing the documents for corrections that the proofreaders made, which are easily transformed into rules, while being generalizable so that the new rules can be applied to every document from the respective company.

4.2 Test Results

4.2.1 Results without additional Rules

Figure 4.2 shows the number of corrections made by the system in relation to the number of corrections the proofreaders made. Without any additional rules, the program corrected 26 errors, denoted by the orange bar labeled "True Positive 1", that were present in the test files. The proofreaders corrected 246 errors, represented by the green bar labeled Proofreader, and zero false positives are present, meaning no wrong corrections were made. This means that the program corrected 10.57% of the errors the proofreaders detected. The baseline for the rules extracted from the guidelines is 28 rules. The low rate of corrections is due to the poor quality and the low level of detail in which the guidelines are written. It is also important to note that a significant number of the corrections in the tested documents were stylistic and semantic, which is not possible to replicate with the current rule set. However, all the rules that were applied in the first test run were consistently applied throughout all the documents of the respective company. This means that even at this low rate of correction, the rules that are applied are applied more consistently than if done by hand.
It is also noteworthy that, without extended rules, there were zero false positives. The guidelines, provided by the companies, mostly did not include a lot of generally

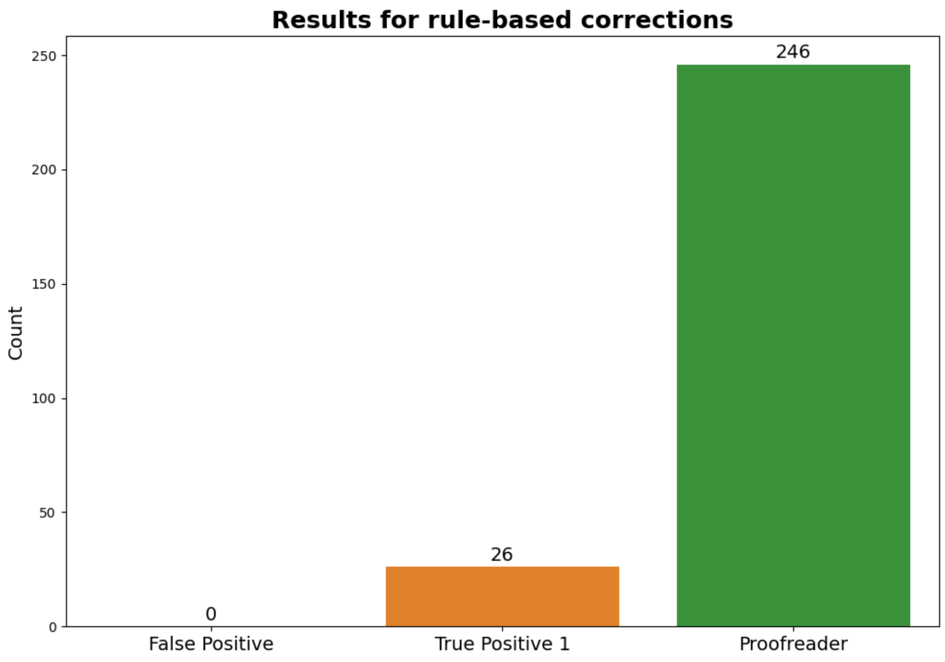


Figure 4.2: Results without additional rules

applicable rules. This is reflected in the number of corrections made.

4.2.2 Results with additional Rules

If the rule files are adapted and expanded as described in Section 4.1, the number of corrections made by the program could be more than quadrupled. This expansion of the rule files consisted of adding 17 rules, for a total of 45 rules.
As seen in Figure 4.3, with the extended rules, the system could correct 106 errors from the tested files as shown by the orange bar labeled "True Positive 2". The number of corrections the proofreaders made did not change and is denoted by the green bar labeled "Proofreader". With the extended rule files, three false positives were produced, which are shown by the blue bar labeled "False Positive". These false positives resulted purely from inclusive language in German, also known as "gendering". With the extended rules, 43% of all corrections made by the proofreaders can be done automatically. It is important to note that the rules need to be adjusted, which requires human interaction, but it can decrease the workload for proofreaders.

4.2.3 Interpretation

The tool was tested on 25 files with varying numbers of pages, ranging from one to four. The number of pages the tested documents had can be seen in Figure 4.1.

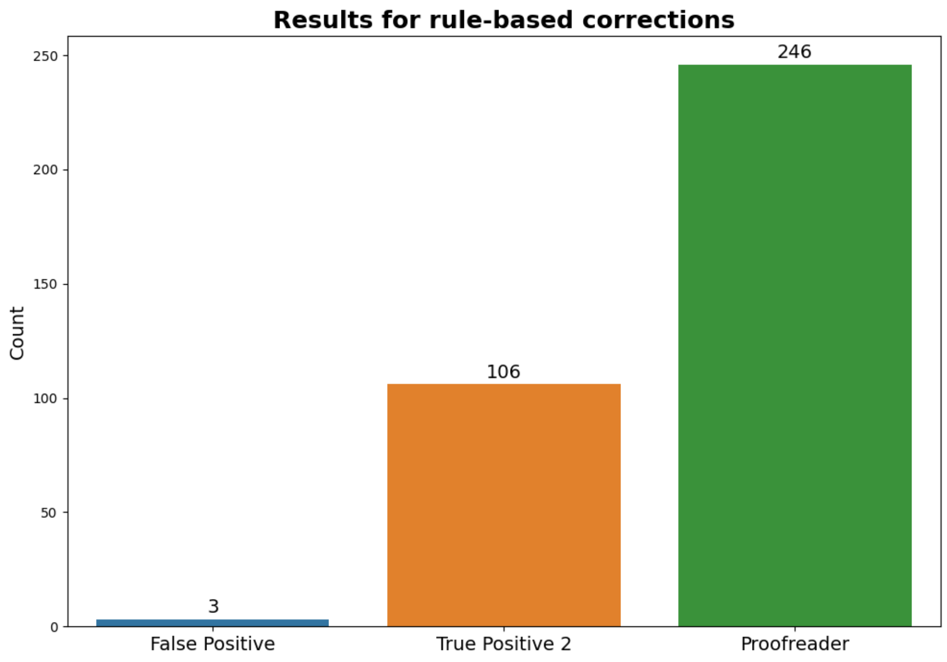


Figure 4.3: Results with additional rules

Since there was no noticeable change in the time difference between the beginning and the end of the corrections for the documents of different lengths, speed was not specifically tested. The varying percentage of corrections that could be made from company to company is the result of the varying quality and level of detail of the different guidelines. The percentage of corrections that can be made is proportional to the quality and level of detail in the guidelines. Companies with higher correction percentages tend to possess more extensive and robust guidelines. The correlation between guideline quality and correction percentage is also shown in the extension of the existing guidelines. With the additional rules not specified by the companies, more of the corrections could be automated.
It is, however, important to note that the distribution of true and false positives was not consistent over the tested companies, as shown in Figure 4.4. This comes from the non-uniform standards that companies use regarding inclusive language. Figure 4.4 uses a different color scheme as described in Subsections 4.2.1 and 4.2.2. In this figure, the blue bars labeled "True Positive 1" represent the percentage of true positives without additional rules, the orange bars labeled "True Positives 2" represent the percentage of true positives with extended rule files and the green bars labeled "False Positives" represent the percentage of false positives with extended rules. It is important to note that 100% for any given bar references 100% of the corrections from that company.
The results demonstrate that, when the corresponding rules are well-defined, the

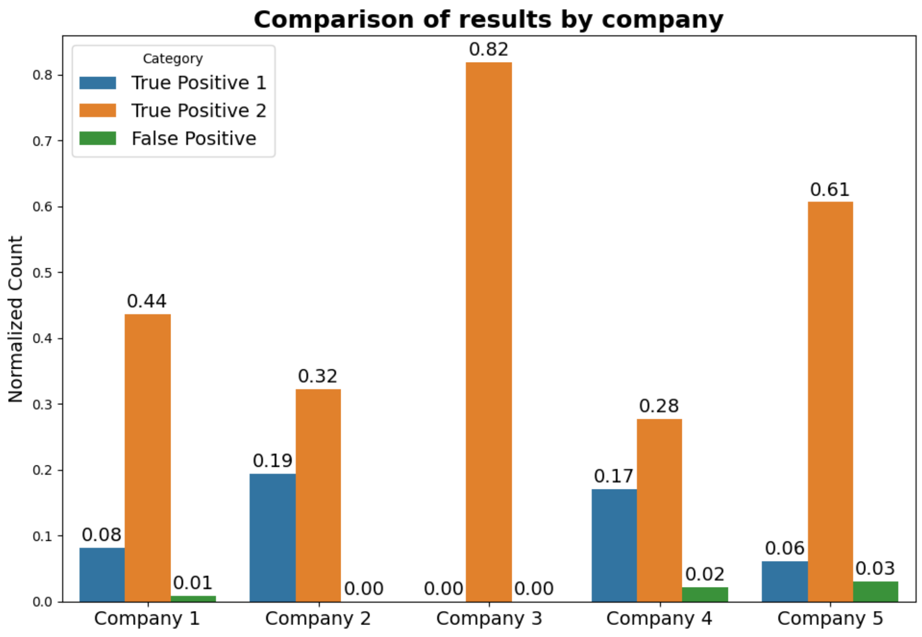


Figure 4.4: Results with additional rules by company

rule-based correction system can reliably address a substantial portion of the inconsistencies. The observed automation rate of about 43% indicates a significant reduction in manual proofreading effort. However, the system's effectiveness is limited by the scope and specificity of the predefined rule set. Corrections requiring nuanced interpretation of linguistic context or semantic intent, for example, remain beyond the capabilities of the current implementation. This limitation highlights the trade-off between transparency and flexibility in deterministic approaches. Furthermore, reliance on static rules requires ongoing maintenance and expansion as language usage and domain-specific conventions evolve. Despite these constraints, the system offers a valuable, interpretable tool for environments that prioritize consistency, reproducibility, and data confidentiality. Overall, the results support the viability of rule-based correction systems as a practical complement to human proofreading in structured publishing workflows.

Chapter 5

Conclusions and Future Work

5.1 Conclusion

The main objective of this thesis was to develop a system that supports proofreaders in their work by automating a portion of their work. The primary problem in the current correction process is that a significant number of rules defined in the guidelines are not applied consistently. A further issue is the time required to rectify the identified errors and implement the established guidelines on the documents.
To achieve this, a rule-based approach was chosen. This facilitated the application of the defined rules. The developed solution consists of five classes, which are needed in order to complete the defined tasks. These five classes are:

- Adapter
- RuleParser
- Rule
- Applier
- Storage

The Adapter is the first class that comes into contact with the rule files as well as the documents after the user imports them. Its objective is to remove any unnecessary characters and either remove the escape characters for the rules or replace them with appropriate placeholders for the text. These placeholders for the escape characters allow for the text to be edited without information loss in terms of formatting.
After the rules are cleaned by the adapter, they are given to the RuleParser. There, the clean strings, which represent the rules, are transformed into rule objects, which allows them to be applied to the text. The rule objects are then stored in a list.

The Storage is a class used to store all relevant information during the entire correction process. An object of this class is created at the beginning of the process. At that point, it does not have any information stored. The 10 attributes that this object has are filled with information throughout the entire process.
The core part of the program is the Applier. It is used to apply the rules to the text. A storage object is given to the Applier in order to apply the rules. This is done by iterating over the list of rules that is stored in the Storage object. Then the rules are applied to the text. The exact process of the application is described in Section 3.1.
To evaluate the developed solution, 25 files from five different companies were selected. The guidelines provided by the companies were then transformed into rule objects. Then, as a first test run, the files were corrected with the respective guidelines. With no additional rules, 10.57% of the corrections, the proofreaders made could be done. In a second step, the rule files were extended with rules that are not defined in the guidelines. To obtain these additional rules, the documents were analyzed, and consistent corrections the proofreaders made were used as a base for the new rules. The newly added rules are applicable to all documents from the respective companies. With the additional rules, 43% of the corrections the proofreaders made can be done. The proofreaders will likely do the same in order to increase the coverage of the system and relieve them of more work. The statistical evaluation of the performance of the created system shows that the correction process is improved through the system. Mainly, the consistency in applying the guidelines and the speed of correction have increased. As the main goal of this thesis is to develop a system to aid the proofreaders in the correction of corporate documents based on guidelines and not to automate their entire body of work, this goal was achieved with the capabilities of the developed solution.
The implementation of a rule-based correction system provides a structured, interpretable approach to automating parts of the proofreading tasks. However, several limitations emerged during development and testing that warrant critical examination. First, the system's effectiveness depends heavily on the comprehensiveness and quality of the rule set. While the system successfully automated 43% of proofreader interventions, this percentage depends on the existence of rules for relevant cases. Although adding new rules is straightforward due to the system's modular design, it creates a dependency on human expertise to formulate, test, and maintain the rules. In fast-changing environments or domains with evolving language usage, this dependency can create bottlenecks. Second, the program lacks the ability to handle context-sensitive errors or nuanced phrasing. For instance, a word that is appropriate in one instance may require replacement in another, which is difficult

to address without more complex logic or probabilistic models. Additionally, there is no mechanism for learning from corrections over time, which limits the system's ability to adapt or improve autonomously.
Despite these drawbacks, the system excels in areas where interpretability, privacy, and rule enforcement are paramount. In industries where sensitive documents must not be exposed to external AI services, this rule-based solution offers a viable, controllable alternative. However, to increase robustness and scalability, future versions could integrate statistical components or machine learning techniques. This hybrid approach would allow the system to maintain transparency while becoming more adaptable—a necessary evolution for wider adoption. As described in Section 5.2, there are possibilities to expand the capabilities of the developed system in order to increase the coverage of corrections and possibly to make it more widely applicable. Some other systems developed for correcting texts, which use generally applicable systems in a language-independent setting, could be used to enhance the capabilities of the system.

5.2 Further Work

The main objective of this thesis was achieved with the developed solution. The program supports the proofreaders and can perform part of the proofreaders' work. With extended and adjusted rules, 43% of the work of the proofreaders can be automated, which can speed up the correction process and ensure consistency in the defined rules. The program developed in the course of this thesis can be extended in a multitude of ways. One example of this would be to implement a spelling correction process for specific languages. With Switzerland in mind, a spelling correction process for the four official languages, German, French, Italian, and Rhaeto-Romance, would be sensible.
Another way that the capabilities of the program could be expanded would be the usage of an artificial intelligence to analyze the documents in a context-sensitive manner, to find and correct any errors that the previous correction steps may have caused. This would also allow for the addition of context-sensitive rules, which could open up a wide range of new possibilities.
The system of Óladóttir et al. [10] has some promising approaches used, which could be implemented in the correction program developed with this thesis. Especially the part-of-speech (POS) and the tree matching to detect and possibly correct grammatical errors.
Another addition that could be made to the developed system would be the addition of language-specific rules, tailored to common dyslexia mistakes, as in the work of

Alamri et al. [5]. This could enhance the user experience for people with dyslexia and make the program more accessible and inclusive. This could also eliminate common spelling mistakes. It is also a possibility to create a basic rule set for the four official languages of Switzerland in order to make the tool more usable for the general correction of texts, and not only for standardizing and enforcing guidelines onto a given text.

Bibliography

[1] Jeunghyun Byun, Hae-Chang Rim, and So-Young Park. Automatic spelling correction rule extraction and application for spoken-style korean text. In *Proceedings of The Sixth International Conference on Advanced Language Processing and Web Information Technology, ALPIT 2007, Luoyang, Henan, China, 22-24 August 2007*, pages 195–199. IEEE Computer Society, 2007.

[2] Itziar Aduriz, Eneko Agirre, Iñaki Alegria, Xabier Arregi, Jose Maria Arriola, Xabier Artola, Arantza Díaz de Ilarraza Sánchez, Nerea Ezeiza, Montse Maritxalar, Kepa Sarasola, and Miriam Urkia. A morphological analysis based method for spelling correction. In Steven Krauwer, Michael Moortgat, and Louis des Tombe, editors, *Sixth Conference of the European Chapter of the Association for Computational Linguistics, Proceedings of the Conference, 21-23 April 1993, Utrecht, The Netherlands*. The Association for Computer Linguistics, 1993.

[3] Lidia Mangu and Eric Brill. Automatic rule acquisition for spelling correction. In Douglas H. Fisher, editor, *Proceedings of the Fourteenth International Conference on Machine Learning (ICML 1997), Nashville, Tennessee, USA, July 8-12, 1997*, pages 187–194. Morgan Kaufmann, 1997.

[4] Marcin Milkowski. Developing an open-source, rule-based proofreading tool. *Softw. Pract. Exp.*, 40(7):543–566, 2010.

[5] Maha Alamri and William John Teahan. Automatic correction of arabic dyslexic text. *Comput.*, 8(1):19, 2019.

[6] Andrew R. Golding and Dan Roth. A winnow-based approach to context-sensitive spelling correction. *Mach. Learn.*, 34(1-3):107–130, 1999.

[7] John Lee and Stephanie Seneff. Automatic grammar correction for second-language learners. In *Ninth International Conference on Spoken Language Processing, INTERSPEECH-ICSLP 2006, Pittsburgh, PA, USA, September 17-21, 2006*. ISCA, 2006.

[8] Shanchun Zhou and Wei Liu. English grammar error correction algorithm based on classification model. *Complex.*, 2021:6687337:1–6687337:11, 2021.

[9] Yifei Hu, Xiaonan Jing, Youlim Ko, and Julia Taylor Rayz. Misspelling correction with pre-trained contextual language model. *CoRR*, abs/2101.03204, 2021.

[10] Hulda Óladóttir, Thórunn Arnardóttir, Anton Karl Ingason, and Vilhjalmur Thorsteinsson. Developing a spell and grammar checker for icelandic using an error corpus. In Nicoletta Calzolari, Frédéric Béchet, Philippe Blache, Khalid Choukri, Christopher Cieri, Thierry Declerck, Sara Goggi, Hitoshi Isahara, Bente Maegaard, Joseph Mariani, Hélène Mazo, Jan Odijk, and Stelios Piperidis, editors, *Proceedings of the Thirteenth Language Resources and Evaluation Conference, LREC 2022, Marseille, France, 20-25 June 2022*, pages 4644–4653. European Language Resources Association, 2022.

Erklärung

gemäss Art. 30 RSL Phil.-nat.18

Name/Vorname:	Neeb Nils Erik
Matrikelnummer:	21-106-299
Studiengang:	Bachelor of Science in Computer Science
	[✓ Bachelor] Master Dissertation
Titel der Arbeit:	Rule-based Text Correction
LeiterIn der Arbeit:	PD Dr. Kaspar Riesen Corina Masanti

Ich erkläre hiermit, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.
Für die Zwecke der Begutachtung und der Überprüfung der Einhaltung der Selbständigkeitserklärung bzw. der Reglemente betreffend Plagiate erteile ich der Universität Bern das Recht, die dazu erforderlichen Personendaten zu bearbeiten und Nutzungshandlungen vorzunehmen, insbesondere die schriftliche Arbeit zu vervielfältigen und dauerhaft in einer Datenbank zu speichern sowie diese zur Überprüfung von Arbeiten Dritter zu verwenden oder hierzu zur Verfügung zu stellen.

Bern, 29.05.2025

Ort/Datum

Nils E Neeb

Unterschrift