Wikipedia Learner’s Corpus

Bachelor’s Thesis

Jiří Kletečka

Brno, Spring 2017
This is where a copy of the official signed thesis assignment and a copy of the Statement of an Author is located in the printed version of the document.
Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Jiří Kletečka

Advisor: Mgr. et Mgr. Vít Baisa, Ph.D.
Acknowledgement

At this place I would like to thank my advisor for all his support, consultations, valuable advice and answers to my questions and last but not least for all his time he devoted to me. All of these things were very helpful and greatly contributed to content of this thesis.
Abstract

This bachelor’s thesis deals with an automated creation of error-annotated corpus from Wikipedia history of articles. Such corpus contains the newest versions of articles with marked errors obtained from their editing history. For that reason, a new tool was designed and implemented. After implementation, it was used in the process of corpus creation using Czech Wikipedia database dump and this corpus was uploaded to the faculty server for public use through interface of Sketch Engine.
Keywords

Learner’s corpus, Error corpus, Wikipedia, Wikipedia revision history, Sketch Engine
List of Tables

4.1 Sample of each error type 23
4.2 Sample output of 4 sentences from 4 randomly chosen articles in good articles category 25
4.3 Example of tokenized and vertical file 26
5.1 Comparison with the Chyby corpus 30
A.1 Command line options 35
A.2 Samples of error types I 36
A.3 Samples of error types II 37
List of Figures

2.1 Output of Wikiedits tool 5
4.1 Page representation in ErrCorp 14
4.2 Input overview 15
4.3 Page processing overview 16
4.4 Nested errors 17
4.5 Error extraction overview 18
4.7 Example of wrapping errors into hiding string 19
4.6 Oldest vs. latest mode comparison (red text is hidden) 19
4.8 Example 20
1 Introduction

This bachelor’s thesis deals with an automated creation of error-annotated corpus (learner’s corpus) from Wikipedia history of articles. The main goal of this work is to create corpus which could be afterward used by linguists and another language users through an interface of corpus manager Sketch Engine [1].

Learners’ corpora, in general, have many diverse use cases in computer and linguistic sciences. As they contain common mistakes of language learners these annotations could be used programmatically in proofreading tools (e.g. for spelling correction, grammar correction, etc.). These corpora can be also used in linguistic research or teaching for example for analysis of common language mistakes. All of these areas are data-demanding so there is a permanent need for larger corpora. The size of common corpora varies between ten millions and billions of tokens but this size is hardly reachable for corpora whose errors are annotated manually.

The learners’ corpora in a classic way are created from students’ works. These works are corrected by teachers and rewritten into electronic form. For many languages large and quality error corpora do not exist just because rewriting into electronic form is very costly and time-consuming.

For example for English speaking researchers, there is NUCLE corpus which consists of 1414 hand-corrected essays written by foreigners [2]. There is also Czech learner’s corpus developed by the Faculty of Informatics at Masaryk University called Chyby. This corpus is manually annotated and as of 2009 it contained approx. 500,000 word forms and almost 18,000 errors. It was made from university students’ works [3]. Another Czech corpus named the AKCES/CZESL was created at Charles University in Prague and Technical University in Liberec containing about 2 million of words [4]. This corpus is made from collected documents made by second-language learners. Many creators and learners’ corpora all over the world are associated in the Learner Corpus Association1.

1. Introduction

If we want to create corpora larger in the order of ten or hundred times we have to modify gathering process somehow. The possible approaches are outlined in Chapter 2.1.

The quality of manual corrections also often depends on corrector’s skills or even his momentary attention at the time of proofreading. These disadvantages are minimized in computer systems like Wikipedia where many users contribute to them and improve texts together. The limitation can be an encyclopedic style of articles, which tends to be too formal or uncommon selection of topics. Though computer systems have also some limitations, the corpora created this way are noticeably larger than the manual one and what’s more important, the corpora creation can be fully automated. Therefore, the creation of learners’ corpora from Wikipedia is only beneficial. Other advantages and limitations are more discussed in Chapter 3.

There are already some tools for Wikipedia error corpora creation. The first one is Wikiedits tool which was developed by Faculty of Mathematics and Computer Science at Adam Mickiewicz University in 2014. It has been used to create WikEd Error Corpus [5]. However, its resulting corpora are intended to be used rather programmatically in proofreading tools and not in linguistic research. For example, the Wikiedits tool does not classify errors and resulting corpora contain only extracted and isolated sentences with annotated mistakes, whereas linguists often need to know wider context of errors. The corpora also cannot be used by the Sketch Engine directly.

Another implementation is called Wiki Error Corpus. It is developed by the Institute of Formal and Applied Linguistics at Charles University. This tool is even more specialized as it focuses only on extracting misspelled words and their corrections.

Due to these circumstances, I decided to design and implement tool named ErrCorp which is partially inspired by Wikiedits tool and adds extensions and other functionalities for linguists. The script can be accessed through GitHub. It is further described in Chapter 4. There is also shown how the ErrCorp processes input using the mwclient library or Wikipedia database dump in this chapter. All stages of processing are then explained including the page processing, the error extraction, the post processing, the error classification and the export of corpora.

2. https://www.github.com/jirkle/errcorp
ErrCorp is designed universally and after a few minor edits of configuration files, it can be used with any Wikipedia language version. Language migration and adjustment of configuration files are described further in Chapter 4.2.1.

With this tool, the Czech Wikipedia Learner’s Corpus was created and its specifications can be found in Chapter 5. Whereas, the resulting corpus is placed and available at the installation of Sketch Engine hosted by the Faculty of Informatics at Masaryk University³.

2 Related work

Wikipedia can serve not only as an Internet encyclopedia but also as a rich and interesting source of content for many programmers, linguists or various linguistic projects at all. For example, the content could be downloaded and processed into simple corpus but the usage is wider.

Adéla Štromajerová dealt in her bachelor’s thesis with the creation of the parallel corpus from Wikipedia articles. The parallel corpus is a corpus which contains pairs of translated fragments of texts in two different languages. Her tool can extract the English-Czech pairs from articles which were translated from one language to other. It uses the fact that the articles which are fully translated from another language are marked and refer to the original articles. She got these marked pages along with their translations, then aligned text fragments via the hunalign library and stored them into the corpus [6].

At the Faculty of Mathematics and Computer Science at Adam Mickiewicz University Roman Grundkiewicz and Marcin Junczys-Dowmunt created a tool for Wikipedia error corpora creation called Wikiedits. It has been used to create the English WikEd Error Corpus [5].

The algorithm itself at first removes vandalized revisions. Then it renders Wiki markup into plaintext and splits it into sentences via NLTK toolkit. Edited sentences are extracted using Longest Common Subsequence algorithm (LCS). After extraction, they are matched using similarity function and collected if they meet several surface condition [5, p. 4]. Then every sentence and its corrective edit are compared together token by token and shrank into one sentence with annotated errors as shown in Figure 2.1.

However, resulting corpora are not suitable for linguistic research as the algorithm produces only isolated sentences with no context provided by the article. The errors produced by this tool also have no classification at all. But this types of corpora are great for use in proofreading tools. Actually, the creators used WikEd Error Corpus in an automated ESL (English as a Second Language) error correction scenario. In composition with other error corpora like NUCLE (manually annotated), L8-NAIST (corpora from lang-8) they achieved 35.79%
correction rate upon ST-2013 test set which was significantly better (1.64\%) than without the use of WikEd Error Corpus \cite[ch.4]{5}.

The Institute of Formal and Applied Linguistics at Charles University developed another error extracting tool. It is called Wiki Error Corpus\footnote{https://github.com/ufal/wiki-error-corpus} and this set of scripts focuses on extracting spelling errors and their corrections from Wikipedia dump.

It consists of six mutually independent scripts which are executed on a computer cluster using Oracle Grid Engine. In first four steps, the input dump is split into files which each of them contains one article. Then every article is rendered, cleaned and revisions are extracted.

The fifth step collects spelling errors. It splits old and new revision into sentences and afterward, pairs them as they go. Pairs are then compared using the Levenshtein distance and threshold. Though this approach has one big disadvantage—if someone, for example, adds one sentence at the beginning of the article and then corrects many spelling errors after, this script will not collect them because all sentences after sentence addition are mutually displaced and Levenshtein distance is naturally greater than the threshold.

Finally, the last step lies in writing collected misspellings into four files. One file contains old sentence versions, one corrected sentence versions and last pair consist of the extracted errors themselves while one is ordered by error frequency.

From what has been said, it follows that the resulting corpora contain only spelling errors and therefore it cannot be considered as fully-fledged corpora for linguists. So everything points to the creation
of a new tool which would be able to eliminate disadvantages of existing approaches and create Wikipedia learners’ corpora friendlier to linguists.

The Sketch Engine itself already hosts some learners’ corpora. For example, it processes learners’ corpora Šolar³ and Lektor⁴. They consist of text collections written by pupils and students of Slovene primary and secondary schools [7].

After a brief overview of existing corpora and existing solutions, it is time to explore more precisely another approaches of the error corpora creation.

2.1 Creation of the error corpora

Gathering the corpus containing annotated errors in the traditional way is based on collecting language learners’ or students’ works. These corpora are also known as learners’ corpora and the creation of this type of corpora is obviously very costly and time-consuming as mentioned above because they are proofread manually, works are usually not stored in digital form and thus the rewriting is manual or just slightly automated (e.g. using OCR⁵).

Another disadvantage is the sentiment of the corrector. Errors are often annotated by only one or a few people and therefore they represent corrector’s concept of correct language usage. So if the corpus creator wants to reduce sentiment he needs to spend some additional work.

These difficulties led to some modifications of the gathering process which speed up error corpora creation. Miłkowski [8, ch. 2] and Grundkiewicz [5, ch. 2.2–2.4] proposed these four possible approaches:

- **Using existing resources**—In some industries, proofread texts are collected systematically. One example of this industry could be large translation companies which need to ensure output quality. Proofread texts then could help them to create their own

---

5. Optical character recognition
2. Related work

autocorrect tools or to do some error analysis. These corpora are unfortunately very difficult to obtain because their content is often considered as secret or unshareable as it contains for example books, paperbacks, program GUIs, etc. and therefore they are subject of copyright.

- **Using artificial errors**—Artificial errors are helpful when there is a lack of larger error corpora. This technique uses seed error-annotated corpus to learn rules based on frequency distribution of errors and then applies learned rules to larger corpora and fills errors in them. E.g. the GenERRate tool could be used for this [9].

- **Using social networks for language learners**—Maybe the best source of errors is the social networks for language learners—lang-8, duolingo, etc. At lang-8 native speakers correct texts from language learners while at duolingo each user can correct other’s texts but there is used a system of approving by other users. However, the copyrights for data are often with either the user or the copyright holder and in most cases, an explicit agreement for particular usage has to be established [10, 11].

- **Using content management systems (CMS)**—Nowadays there are many computer systems or environments where users collaborate and improve texts. Google docs, Wikipedia, even Github and other version control systems could be considered as such environments. It is crucial to store all contributor edits to allow restoring of old versions and assigning responsibility for a change. And this history of content is a great source of error corrections.

As this bachelor’s thesis aims at a Wikipedia which belongs to CMS we need to explore these systems more deeply.

Getting errors from CMS generally stands on the postulate that majority of editors make an effort to improve contents of documents over time. This is true in general but Wikipedia is also a target of vandals who ruin the quality of articles. This results in a requirement for a tool which should be able to identify this vandalized revisions and to remove them.
The crucial idea of error extraction itself is in taking every two consecutive versions of one document from history and doing a comparison of them. The differences are located and annotated as errors. There are several approaches how to do the comparison. Individual revisions can be compared line by line, sentence by sentence or character by character. Simple line by line comparison is often not enough because it produces coarse granularity.

For that reason, the script implements sentence by sentence approach for extracting pairs of error and corrected sentences. Then it compares every pair with character by character comparison for getting only the changed character sequences. These minimal changes are post-processed afterward, expanded to whole words and classified and marked as errors. With this approach, it was possible to achieve finer granularity of error annotation.
3 Error corpora from Wikipedia

The third chapter discusses error corpora from Wikipedia—their pros and cons. It also contains a preparation part where the main requirements are established and the classification system used in the Čhýby corpus is introduced. This information is then important in the design process of ErrCorp tool and it also characterizes all resulting corpora.

3.1 Advantages and disadvantages of Wikipedia

Firstly, the Wikipedia and its direct derivatives are copyleft licensed. That means that their users have the right to freely distribute copies and modified versions of them under the same or any compatible license.

Another advantage is its size and diversity. It consists of hundred thousands (Czech - 378,007/10.4.2017[12]) or even millions (English 5,380,561/10.4.2017[13]) of pages from many areas of human research. Besides that, the content is generally considered as reliable.

Individual revisions of articles are also easy obtainable through API. Moreover, the full content of Wikipedia can be downloaded as database dump and processed offline (with some limitations connected with template expansion).

Another advantage is that majority of Wikipedia articles reflect language in its up-to-date version. This is possible thanks to its contributors who renew article contents. Finally, the contributing community is really wide. So the diversity of correctors prevents any sentiment in articles to the maximum extent possible.

There are also some disadvantages. Miłkowski [8] has pointed out several limitations for which Wikipedia as such cannot be considered as a source of an average speaker’s language. Firstly, he addresses the digital divide. It means that people from poorer countries and poorer people generally cannot afford computers and therefore they use them less frequently.

The editors are also more educated—they must have at least some basic IT skills and also some knowledge about the topic they want to contribute to otherwise the editors would revert their contribution.
The limitation is also in the encyclopedic style of articles, which tends to be too formal and the selection of topics is uncommon. As an example, Miłkowski gives us the fact that historical data about churches is not as common in everyday speech as in encyclopedias. Another disadvantage is also vandalism which was discussed more in the previous text.

3.2 Corpora requirements

At this point, it is also needed to establish the main requirements of corpora. These properties will afterward mostly affect the tool functionality.

- **Sketch Engine compatibility**—The input of Sketch Engine is vertical (word-per-line, WPL) or tokenized plain text file. Words, numbers and punctuation marks are written one per line, without any formatting. The inline XML tags and attributes are used for structural differentiation (document, sentence or paragraph boundaries, headlines, metadata, etc.). Sketch Engine allows one own XML schema and the used one is described in the chapter dedicated to the tool output implementation 4.1.5.

- **Linguistic research**—The resulting corpora will be used by linguists through Sketch Engine. They should allow the examination the error itself and also its surroundings and therefore, the desired output is the whole article with inline error annotation.

- **Granularity**—One of the goals is also finer granularity of errors. This means in the context of learner’s corpora to minimize the presence of words in error annotation which are not actually its part.

- **Meta information**—The more metadata corpus contains the better it is for linguists. Metadata is stored in structures. This structures can be, for example, sentence (paragraph, article, error, ...) boundaries and they can hold article name, timestamps, error classification, bot edit flag, good and featured Wikipedia articles distinction, etc.
• **False positives**—This requirement addresses errors caused by vandalized revisions. These types of errors are not desired in the resulting corpora.

### 3.3 Error classification

After error extraction, the errors have to be sorted into several types (classes) of errors such as typographical, grammatical, stylistic and so on. This classification can be used later by linguists who could search corpora only for one particular type of error. The classification itself is not codified and after some research, the classification system inspired by the Czech *Chyby* corpus was used.

The main reason is that the class rules can be written (with some limitations) programmatically. Secondly, this classification system was developed in Czech environment and there is a premise that it reflects and fits Czech language more than other systems.

In *Chyby* corpus errors are divided into five groups [3]:

• **Punctuation**—The class consists of usage of a comma, colon, semicolon, dot, triple dot and other punctuation marks which affect semantic of the sentence.

• **Typographical errors**—These errors are closely connected with the visual form of documents. They contain usage of spaces, hyphens, inverted commas, brackets, one character consonant prepositions or incorrect characters. Overall document layout, document reorganization and formatting changes are also present in this group.

• **Spelling errors**—This group contains various typing errors (e.g. *i/y, s/z*), inflectional noun endings, syntactic errors (valencies), use of capital letters or lower cases, compounds and so on.

• **Lexico-semantic errors**—This group covers incorrect usages of language such as multiword expressions (MWEs), missing words, incorrect use of possessives or bad choice of lexical items.

• **Stylistic errors**—Stylistic errors are the most dependent on the corrector’s concept of language. This group can contain errors
3. Error corpora from Wikipedia

based on the incorrect register (archaic, slang, etc.), repeated expressions, the cumulation of the nouns ending with -ní, use of passive and reflexive passive, incorrect word order, clumsy expressions, too long sentences, etc.

Concrete implementation of ErrCorp error classifier and classification rules are described in the chapter dedicated to ErrCorp tool 4.1.4.
4 Implementation

After previous research, it is clear what kind of tool is needed, what are the requirements and also what are the other approaches used in error corpora gathering. In this chapter, it is described implementation details of ErrCorp, its possibilities and language-specific configuration.

4.1 Algorithm description

From multiple considered languages the Python was chosen. It is used widely in the natural language processing (NLP) area and also easy to use. The whole code was written according to the Python 3 syntax and it is placed at GitHub repository\(^1\). The code of script is licensed under the 3-Clause BSD License.

For better understanding, the script is divided into several files. Input is solved in main.py and WikiDownload.py files, page processing in PageProcessor.py, error extraction in ErrorExtractor.py, post processing in PostProcessor.py and ErrorClassifier.py and finally output addresses Export.py.

The script itself operates in situ, no additional files are created during processing (except the situation when the dump is located online and needs to be downloaded first). It is also unpretentious to memory as it processes input page by page.

4.1.1 Input

The first thing which has to be considered is the way that the articles will be obtained. They could be accessed using MediaWiki action API or by processing the database dump which is provided by Wikipedia\(^2\). Both of these possibilities are implemented.

The MediaWiki action API provides a direct access to the Wikipedia databases using HTTP requests. The endpoint is located at /w/api.php destination while the action parameter denotes the desired action [14]. The most used action is action=query. Using this, it is possible to

---

4. Implementation

query Wikipedia database and acquire data which matches certain criteria. For example, http://en.wikipedia.org/w/api.php?action=query&prop=revisions&titles=Main%20Page&rvprop=content request gets the metadata and the latest revision of the Main page article zipped in a JSON object [15].

For a simpler use of the API, the mwclient library was used. This library provides Site object which is used to access specific Wikipedia site. The Site object then allows getting Page object which represents certain article and after that, it is possible to iterate through individual revisions and collect them for later use.

The second option is to use Wikipedia database dumps with complete page edit history. Wikipedia dump files use unified Wikipedia XML export schema and due to this fact, it is possible to run the script with any input based on this schema. Wikipedia dumps are usually stored in archives so firstly they are opened using Python native libraries and then processed incrementally using event-based XML processor.

For each page, the script gets its name and list of its revisions. For each revision then script gets its content, name or IP of the user who made it, revision comment, timestamp and flag whether the edition was anonymous.

This entire information is stored in dictionary data structure as shown in Figure 4.1. This structure was selected because it reflects an object obtained through MediaWiki action API.

There are also two formats in which anyone can get Wikipedia pages. Firstly, it is possible to retrieve a revision in Wiki markup language. This is the default format for dumps and HTTP responses and also in this format, the revisions are stored in Wikipedia databases.

Figure 4.1: Page representation in ErrCorp

3. https://www.mediawiki.org/xml/export-0.10.xsd
4. Implementation

The second option is to get an article in HTML representation. Unfortunately, the dumps with pre-rendered content do not exist. Also, the MediaWiki action API allows retrieval of only one rendered revision for one request (meanwhile up to 5,000 revisions could be served at once using Wiki markup language) and therefore it is very slow. The Python library called WikiExtractor can do this rendering locally and also directly into plaintext and for that reason, all the revisions are obtained in Wiki markup language.

WikiExtractor is also capable of performing template expansion to some extent—according to its specification, it does not fully support Lua modules. But these templates are used rarely and therefore the usage of this processor is sufficient for our purpose.

4.1.2 Page processing

Revisions obtained through API or from dump need to be unified from Wiki markup language into plaintext. This transformation is done in Page processor. Firstly, the vandalized revisions are removed so they do not have to be processed further. For distinction, there are used the comments of revisions. They are searched for predefined words such as revert, rvrt, vrácen, zrušen and so on. If any of these words are found within the comment then the revision and its direct ancestor are removed.

There are also removed the robot revisions if the robots flag is not provided from the command line. The same principle is applied, com-
4. Implementation

Contents are searched for predefined words and the revision is preserved or discarded according to the result.

After this step, the content of revisions is rendered into plaintext which is done using WikiExtractor library. This rendering is costly transformation and therefore the pool of processes was used for acceleration.

The ideal state is to have the text of the revision without any markup. However, WikiExtractor leaves some inline markup and this has to be removed. For example WikiExtractor does not remove headings markup (=== Heading ===), some HTML tags, bullets or Wiki interlinks in the text.

After removal, the content of each revision is split into sentences and stored as a list. The splitting is done on sentence ending chars (,!?). Before splitting, there are marked non-sentence endings. As non-sentence endings are considered full stops in abbreviations, ordinals, websites, dates and roman numbers.

After splitting, the sentences are cleaned further. The bullets are removed from the start of the sentence, the rest square brackets and braces are also discarded as they are components of Wikipedia links and templates. After all of these steps, the error extraction itself begins.

Figure 4.3: Page processing overview

4. Implementation

4.1.3 Error extraction

The error extraction starts with comparing every two adjacent revisions. It extracts every sentence from old revision which is not present in new revision and vice versa. It’s done via Python function for computing deltas called *unified diff*. This function uses the gestalt pattern matching for finding differences.

Different sentences are stored in two lists—one for the old version of an article and one for the new—and they are complemented by the comment of revision. The sentences longer than 100 words are discarded and then for each old sentence the script tries to find the best matching new sentence. For this purpose, there is used sentence similarity function which returns number from the interval $\langle 0, 1 \rangle$.

As the similarity function has been chosen this equation:

$$
sim(a, b) = 2 \cdot \frac{|\text{bag}(a) \cap \text{bag}(b)|}{|\text{bag}(a)| + |\text{bag}(b)|},
$$

where the \text{bag} is \textit{bag of words} function which returns set of unique words.

This similarity function is calculated for each combination of the old and new sentences. If the result is bellowed the threshold the pair is discarded and the most similar one is selected from all pair combinations and stored. The threshold value 0.7 has been chosen experientially and can be edited changing context key \textit{sentence threshold} (see Chapter 4.2.1).

One sentence could have been edited multiple times over multiple revisions so evolution resolution is started after extracting all sentence pairs.

This resolution produces continuous sequences of one sentence versions (they are called \textit{evolution lists} in next text). The script itself iterates through all extracted pairs and looks if some newest sentence of the pair from older revisions is equal to some oldest sentence of the pair from newer revisions. If so, they are

Z gravitačního zákona vychází
<err>
<err>epiptický</err>
</err>
<corr>ekliptický</corr></corr>
</err>
<corr>eliptický</corr> pohyb planet.

Figure 4.4: Nested errors
joined together and the script continues until all sentence pairs are processed.

Usage of *evolution list* could lead to nested errors as shown in Figure 4.4 where one error is corrected twice. If the nesting needs to be disabled the easiest way is to preserve only the oldest and newest versions of the sentence to next processing. And this approach is used when the nesting flag is not provided as the command line argument.

**Figure 4.5: Error extraction overview**

4.1.4 Post processing and error classification

From previous part, the script has *evolution lists* with multiple versions of sentences sorted from oldest to newest. It could simply flush them to corpora at this time but errors based on whole sentences are not actually what was expected. There is a need for errors which are classified and also more granularized and this is exactly the task of post processor.

Hiding string data structure

Firstly, the data structure, which will help the script to shrink *evolution lists* into one sentence, is needed. For this purpose, there was designed and implemented data structure named *hiding string*. It does exactly what its name means—hides inner error or correction annotation from the outer observer. It also allows the observer to annotate inner content as an error using *wrap* function.
Hiding string works in two modes—the latest mode hides all errors and their content and shows only the content of corrections meanwhile the oldest mode shows only the content of oldest errors and hides all corrections. It is more clear in Figure 4.6.

It also provides a function called wrap. This function allows inserting an annotation of error. Firstly, it checks if wrap conditions are fulfilled (i.e. if the string which is to be wrapped contains whole error annotation and not only its part). Then (if the conditions are satisfied) it wraps inner string with error annotation (see Figure 4.7).

Granularization

As was said in previous part, the script has continuous evolution lists with multiple versions of sentences. At the beginning, post processor stores the newest version of each sentence into hiding string structure and set its mode to oldest. The post processor now sees only the oldest versions of sentences.

Then post processor goes through all sentence versions to the oldest one. In a nutshell, the script chronologically compares them
against hiding string, extracts errors for each sentence version and tries to wrap extracted errors into hiding string. This is done this way:

In each step, it compares sentence along hiding string and gets all differences locations. They are stored in a quaternion \((s_1,e_1,s_2,e_2)\), where \(s_1\) and \(e_1\) are start and end boundaries within hiding string and \(s_2\) and \(e_2\) are the boundaries in the current version of a sentence. The comparison is done using Python library called Sequence Matcher. Sequence Matcher provides only the retrieval of matching block locations and therefore the next step lies in inverting them and getting differences.

The Sequence Matcher also does the comparison upon single characters but this adds too much granularity as it can produce something like this: `<err>S</err><corr>Z</corr>`tišit. So in the next step, post processor expands errors to whole words. The expansion is done simply by widening all four boundaries apart until the punctuation mark or space is reached.

This approach could lead to the state when the differences are overlapping. For example, it could happen when two differences are marked inside one word. So the script concatenates overlapping intervals. The last two steps are shown visually in Figure 4.8.
The error expansion is not done when an error or its correction contain only punctuation marks because expansion would cause coarser granularity.

At this point, ErrCorp has the quaternion with final locations of differences within hiding string structure and the current version of a sentence. It also has the comment of the revision and all this information are sent to Error Classifier.

Error classification

At the beginning, the comment of revision had been used for estimation which type of error is corrected within the text. For example, if revision comment contained predefined word typo, the error class was considered as typo too.

This approach turned out to be insufficient for two main reasons. The first one is obvious—the revision could contain more than one edit and thus all errors corrected in it was considered as one error type meanwhile they do not need to necessary be. This could be resolved by additional comparison of error and it’s correction. I.e. if their char based distance (typically Levenshtein’s one) is lower than some threshold.

But this approach does not help with the second problem which lies in error type diversity. The amount of information contained in the comment is just not enough to resolve the type of one error. Especially when the script wants to use more sophisticated classification system such as in the Chyby corpus. For example, we have no possibility to resolve punctuation error from the stylistic one using only the comment of revision.

For this reasons, the error classification has been implemented differently and wrapped into error classifier which is also better in having a unified interface. The classifier gets comment of revision and pair of sentences at the input along with intervals which specify error and its correction locations within the sentences. Based on this meta information about the error, it guesses and returns one error type.

Implementation of the classifier uses following heuristics for error type estimation:

- **Punctuation** type is resolved when error and correction differ only in punctuation characters which have some semantic meaning. It could
be the addition of comma, removal of dot and so on. Programmatically it’s done this way: all of the following chars are removed from both error and correction,.;...?!¿& and then they are simply compared. If it appears that they are the same then they are classified as this type of error.

**Typographical** errors are these where error and correction vary in use of spaces and other punctuation marks. This is reflected by classifier—if the error is not resolved as punctuation then all other punctuation and space characters are removed. After removal, the same principle as above is applied and strings are compared.

The typographical errors are resolved once more right after spelling errors. The resolution is done with one difference—the error and correction are lowercased. This allows catching errors which are both typographical and spelling. (e.g. `<err>- nepenthes</err><corr>Nepenthes</corr>)

**Spelling** errors contain typos and other misspellings. It also contains the improper use of capital letters.

Firstly, the misused capital letters are resolved—the error and correction are lowercased and compared.

Then the misspellings are resolved. This rule has one initial condition that the bags of words of the error and the correction have the same size. This condition is here for the reason that spelling errors are characterized by the same count of words and addition or removal of words are more characteristic for lexico-semantic and stylistic groups. Membership in spelling class of errors is then verified by the equation

\[
\text{dist}(\text{err}, \text{corr}) < \text{TypoThreshold},
\]

where the \( \text{dist}(\text{err}, \text{corr}) \) is Levenshtein’s distance of error and correction and the \( \text{TypoThreshold} \) is chosen value. By default, it is set to 2.

**Lexico-semantic** along with stylistic mistakes are the most ambiguous of error types. It contains omitted or added word, edits which change semantic of a sentence or multi-word expressions.

The classifier classifies only omitted or added words because other errors are formally difficult to resolve. Whole resolution is done using the symmetric difference

\[
\text{len}(\text{bag}(\text{err}) \triangle \text{bag}(\text{corr})) < \text{WordThreshold},
\]
where the $bag$ is the bag of words function and the $WordThreshold$ denotes how many words can be unique for error and its correction to be still considered as a lexico-semantic error.

**Stylistic** errors are resolved at the end as they are most general. They contain e.g. change of word order, incorrect register, clumsy expressions etc. The goal of the classifier is only to resolve word order changes as other changes are difficult to classify programmatically.

Firstly, the classifier looks if words of correction are located somewhere in the old sentence. This happens when its words were relocated within the sentence. The bag of words is used for resolution:

$$\text{len}(\text{bag}(\text{corr}) - \text{bag}(\text{old})) == 0$$

The second used approach is this: As the stylistic errors are recognizable when old and new sentences have many common words, the classifier looks on the symmetric difference of their word sets. If their symmetric difference differs up to certain amount of words (by default $2 \times WordThreshold$) then errors are considered as the stylistic ones.

**Unclassified** errors are in the last group and this group contains errors which had not been classified into one of the previous classes.

<table>
<thead>
<tr>
<th>Error type</th>
<th>Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>Punctuation</td>
<td>takový&lt;err/&gt;&lt;corr&gt;, &lt;/corr&gt; který</td>
</tr>
<tr>
<td>Typographical</td>
<td>ztvárnění filmu &lt;err/&gt;&lt;corr&gt;&quot;&lt;/corr&gt; Kouzelník&lt;err/&gt;&lt;corr&gt;&quot;&lt;/corr&gt;</td>
</tr>
<tr>
<td>Spelling</td>
<td>&lt;err&gt;mikrovná&lt;/err&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;corr&gt;mikrovlnná&lt;/corr&gt; trouba</td>
</tr>
<tr>
<td>Lexico-semantic</td>
<td>&lt;err&gt;Mezi&lt;/err&gt;&lt;corr&gt;Mezi antické&lt;/corr&gt; filosofy se...</td>
</tr>
<tr>
<td>Stylistic</td>
<td>&lt;err&gt;nevkládají&lt;/err&gt;&lt;corr&gt;zpravidla nemohou vkládat&lt;/corr&gt;</td>
</tr>
<tr>
<td>Unclassified</td>
<td>&lt;err&gt;ve druhé polovině&lt;/err&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;corr&gt;během&lt;/corr&gt; 30. let</td>
</tr>
</tbody>
</table>

Table 4.1: Sample of each error type
Grafting

When the classification of error is done the script assembles error annotation and tries to wrap it within the *hiding string*. Then it continues in processing remaining sentences of evolution list.

After processing all *evolution lists* of all sentences, the modes of the *hiding string* structures are switched to *latest* mode so now the latest version of sentences are observable. Then the newest revision of article with split sentences is obtained and each *hiding string* (its full content with annotations) is tried to be placed in proper place. It is done by comparing the observable content of *hiding string* with each sentence of the newest revision.

Using this approach some sentences remain ungrafted. This could happen, for example, when some sentence was edited in history and then removed. These orphan sentences are collected separately and stored to own file as they can be used as a source of errors as well.

The sample results are shown in Table 4.2. It is obvious that the close errors are often connected and could be concatenated. For example <err>byly</err><corr>byla</corr> <err>určeny</err><corr>určena</corr> can be concatenated into one error <err>byly určeny</err><corr>byla určena</corr>. But this is not a rule in general and it could be the topic for further research.

### 4.1.5 Output

After the post processing part, the latest revision and sentence orphans have to be stored in the format accepted by *Sketch Engine*. The *Sketch Engine* accepts the corpus files in the format of tokenized text or in full vertical text (word-per-line vertical). Their examples are in Figure 4.3.

This kind of plaintext files consists of positions, one position per line. Each position then has attributes associated with it and separated by the tabulator. The most common is a triple—*word*, *lemma* and *tag*. The *word* is one token of the article (it could be the whole word, punctuation mark, etc.), the *lemma* is the basic form of a word and the *tag* contains information of token lexical categories.

The tokenized file has only the *word* attribute and others are added by *Sketch Engine* automatically.
Muniční sklad Hajniště
Ty se využívaly na ukládání munice s prošlou záruční dobou, jež byly určeny k delaboraci (zničení).

Země Koruny české za vlády Ferdinanda I.
Za jeho vlády přišli do českých zemí jezuité, znovu uvedeny některé starší řády a konečně roku 1561 po více než stu letech znovu obsazen stolec pražského arcibiskupa.

Eliška Junková
Protože její touhou bylo cestovat, učila se jazyky, jejich učení ji šlo poměrně snadno.

Program Apollo
Snažili se přitom minimalizovat riziko pro astronauty, i náklady technickou náročnost letů.

Table 4.2: Sample output of 4 sentences from 4 randomly chosen articles in good articles category
Table 4.3: Example of tokenized and vertical file

<table>
<thead>
<tr>
<th>Tokenized file</th>
<th>Vertical file</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;err type=&quot;ls&quot;&gt;</code></td>
<td><code>&lt;err type=&quot;ls&quot;&gt;</code></td>
</tr>
<tr>
<td>Nebeská</td>
<td>Nebeská nebeský k2eAgFnScld1</td>
</tr>
<tr>
<td>mechanika</td>
<td>mechanika mechanika k1gFnSc1</td>
</tr>
<tr>
<td>se</td>
<td>se se k3xPyFc4</td>
</tr>
<tr>
<td>zabývá</td>
<td>zabývá zabývat k5eAaImIp3nS</td>
</tr>
<tr>
<td>&lt;/err&gt;</td>
<td>&lt;/err&gt;</td>
</tr>
<tr>
<td>&lt;g/&gt;</td>
<td>&lt;g/&gt;</td>
</tr>
<tr>
<td><code>&lt;corr type=&quot;ls&quot;&gt;</code></td>
<td><code>&lt;corr type=&quot;ls&quot;&gt;</code></td>
</tr>
<tr>
<td>Zabývá</td>
<td>Zabývá zabývat k5eAaImIp3nS</td>
</tr>
<tr>
<td>se</td>
<td>se s k7c7</td>
</tr>
<tr>
<td>&lt;/corr&gt;</td>
<td>&lt;/corr&gt;</td>
</tr>
</tbody>
</table>

Sketch Engine uses majka [16] (a Czech morphology analyzer) for finding tag and lemma to every word. This tag describes all grammatical categories of the given word such as part of speech, number or gender. For example, the output for the Czech word “těles” can look as follows:

těleso k1gNnPc2

where the “k1” means the part of speech – noun, “gN” stands for the gender – neutrum, “nP” for the plural and “c2” for the genitive case.

It also has to do morphological disambiguation which deals with the problem of having more tags for one word. For example, the word “information” can be both singular and plural. The tool for the disambiguation goes through the possible tags and decides which tag is correct based on predefined rules and collocation statistics. At Sketch Engine this is done using Desamb [17].

The tokenized or vertical file also contains XML tags which determine document structures. The ErrCorp uses the `<doc>` tag for annotation of article boundaries. This tag has two attributes—n for a name
of article and \( t \) for a date of the last editing. Then there are \(<s>\) tag used for enclosing sentences and pair of \(<err><corr>\) tags for error annotation. Currently, the \(<err><corr>\) tags have only type attribute which contains one of error classes classified by the classifier (punctuation, spelling, etc.). This could be improved in future, for example, with timestamp or comment attribute.

The last tag used in corpora vertical files is the glue tag \(<g/>\). It indicates that following and precedent positions have not been separated by space.

The ErrCorp concatenates all sentences of one article from previous processing and fills in the rest of tags (doc, s). Then the tokenization of the text starts using the unitok tool [18]. It splits the text into tokens and preserves inline XML. The unitok also adds the glue tag \(<g/>\) where it is needed.

Then the output is stored in two files (one for orphan sentences and one for the whole documents) and manually uploaded to Sketch Engine using its web interface. Here the vertical file is constructed and after that, the whole corpus is compiled. The compilation of corpora lies in creating word sketches, thesaurus, terms and subcorpora (if any was defined) [19]. After all of these steps, the corpus is ready to use.

4.2 ErrCorp usage and doc overview

The possible input of ErrCorp is the path to any bz2 or XML file (remote or local) which uses Wikipedia XML export schema [5]. The second option is calling ErrCorp with names of articles separated by separator char (default is [;]). They are then downloaded through MediaWiki action API using the mwclient library.

From the command line, it is possible to provide robots flag which states if the revisions made by robots are included in the processing. Another flag is the nesting flag which states that the evolution of errors through revisions will be enabled. All options are listed in Appendix A.1.

5. https://www.mediawiki.org/xml/export-0.10.xsd
4. Implementation

4.2.1 Configuration

There are several options how to configure the behavior of the script in the `main.py`. The configuration is done by the editing of the `context` object. The most remarkable are these options:

- **Pool Processes**—It denotes count of processes which are used at rendering stage of page processing.

- **Typo Threshold**—Threshold which is used by the classifier when classifying spelling errors.

- **Word Threshold**—Threshold which is used by the classifier when classifying lexico-semantic and style errors.

- **Sentence Threshold**—Threshold which is used by the error extractor to resolve if two sentences could be considered as predecessor and successor.

- **Supported Langs**—This list contains tuples of name and abbreviation for supported languages. Name denotes name of language file in `confs` directory, whereas abbreviation denotes language for `mwclient` (*MediaWiki* action API endpoint).

The behavior can be edited also by rewriting base functions which are in `Utils.py`. It is possible to edit sentence similarity function used by error extractor from there and it also contains the bag of words function.

4.2.2 Language migration

Language migration lies in creating language-specific configuration files located in `confs` directory and then adding language into supported languages in `main.py`. Conf directory contains two types of files. First one is the configuration files for the *unitok* tokenizer. There are also files used by *ErrCorp* itself and their name ends with `-err-corp.py`.

These files contain language-specific regular expressions for text splitting and for revision filtering (bots, reverted, and other types). The most important are:
4. Implementation

- **Abbrs**—Regex contains abbreviations which are used during sentence splitting. The Czech abbreviations were obtained from Wikipedia⁶.

- **Websites**—This group consists of commonly used web domains.

- **DigitSentenceEndings**—As sentence endings, there are not generally considered the dots after numbers. This group allows setting specific combinations of digits and words which actually are the sentence endings.

- **ClassifierPunctuation**—This is the group of all punctuation marks. They are used for resolution of punctuation error type by the classifier.

- **DecimalPoint**—Regex for catching up the decimal points.

- **ExcludeFilter**—Filter for excluding whole pages from processing (e.g. Main Page, Help pages, Discussion pages, etc.).

- **RevertFilter**—Filter for excluding vandalized revisions.

- **BotFilter**—Filter for resolving revisions made by bots.

---

⁶. https://cs.wiktionary.org/wiki/Kategorie:%C4%8Cesk%C3%A9_zkratky
5 Corpus description

The resulting corpus was made from Czech Wikipedia dump from 6th May 2017. It was done in approximately 48 hours on the common desktop processor while the script processed one million of all Wikipedia pages (including discussions, help pages, etc.). From the total amount of one million pages were about 380,000 articles while at least one error was extracted from 285,776 articles. The revisions made by bots were also included in the processing and evolution of errors disabled. Overall, it was annotated 3,793,553 errors which give 13.27 errors per article.

The structure of errors is shown in next table along with the comparison against the Chyby corpus. Data about the Chyby corpus was obtained from the article [3]. There was the spelling group divided into morpho-syntactic, simple and other spelling types in the Chyby corpus and for purpose of comparison, these sub-groups were shrunk into one.

<table>
<thead>
<tr>
<th>Type of error</th>
<th>The ErrCorp corpus</th>
<th>The Chyby corpus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Punctuation</td>
<td>114,173 (3.01%)</td>
<td>3,837 (21.32%)</td>
</tr>
<tr>
<td>Typographical</td>
<td>823,778 (21.71%)</td>
<td>2,165 (12.03%)</td>
</tr>
<tr>
<td>Spelling</td>
<td>666,876 (17.58%)</td>
<td>4,903 (27.25%)</td>
</tr>
<tr>
<td>Lexico-semantic</td>
<td>1,754,504 (46.25%)</td>
<td>2,536 (14.09%)</td>
</tr>
<tr>
<td>Style</td>
<td>228,667 (6.03%)</td>
<td>4,184 (23.25%)</td>
</tr>
<tr>
<td>Unclassified</td>
<td>205,555 (5.42%)</td>
<td>371 (2.06%)</td>
</tr>
<tr>
<td>Total</td>
<td>3,793,553 (100.00%)</td>
<td>17,996 (100.00%)</td>
</tr>
</tbody>
</table>

Table 5.1: Comparison with the Chyby corpus

Some examples of errors for each group are present in Appendices A.2 and A.3 and also in Table 4.1.

The ErrCorp and Chyby corpora are most diverse in the lexico-semantic group. The classifier distinguishes these errors when they consist of additions or omissions of words. The diversity, in this case, could be caused by the enlarging character of Wikipedia which stands
on the principle of collaborating users who add facts and improve articles incrementally.

The corpus was made public and it is available at the Sketch Engine site of the Faculty of Informatics of Masaryk University under the name "Czech Wikipedia Learner’s Corpus". This corpus is published under the CC-BY-SA 4.0 license.
6 Conclusion

In this thesis, the process of creating learners’ corpora from Wikipedia edition history was discussed and described. In the first part, the reader obtained overview upon existing and related tools and description of their approaches. The Wikipedia is not the only possible source of learners’ corpora and therefore this chapter also contained the section dealing with general techniques of gathering learners’ corpora.

The second chapter discussed advantages and disadvantages of Wikipedia as a source of texts for natural language processing (NLP) research. There was also a preparation part where the main requirements of corpora were established along with classification system used in the Chyby corpus. These requirements were then important in the design process of ErrCorp tool.

In the third chapter, the algorithm of ErrCorp was described from start to end. The algorithm was split into five stages for better understandability: input (getting content of revisions using Media Wiki API or Wikipedia dumps), page processing (rendering revisions from Wikipedia markup into plaintext, cleaning up the text), error extraction (extracting pairs of edited sentences, concatenating them into evolution lists), post processing (shrinking evolution lists into one sentence using hiding string data structure, classifying, grafting sentence into newest revision) and output (exporting grafted revision into tokenized file, creation of vertical file). There were also described configuration possibilities of ErrCorp and language-specific settings needed when porting ErrCorp to another language.

The last part then described resulting corpus which was made from Czech language mutation of Wikipedia and uploaded to Sketch Engine hosted by the Faculty of Informatics. The corpus was also compared against the manually annotated Chyby corpus.

Finally, the resulting corpus can be now used in linguistic research as well as a source of errors for another NLP tools.
Bibliography

1. Sketch Engine | language corpus management and query system [online] [visited on 2017-04-20]. Available from: https://www.sketchengine.co.uk/.


## A An appendix

<table>
<thead>
<tr>
<th>Command options</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>General options:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--help, -h</td>
<td></td>
<td>Print help</td>
</tr>
<tr>
<td>--lang, -l</td>
<td>[czech</td>
<td>english]</td>
</tr>
<tr>
<td>--separator, -s</td>
<td>[s]</td>
<td>Separator char, default [;]</td>
</tr>
<tr>
<td>--robots, -r</td>
<td></td>
<td>Flag: Include revisions made by bots</td>
</tr>
<tr>
<td>--nesting, -n</td>
<td></td>
<td>Flag: If present, nesting of errors are allowed, yet experimental</td>
</tr>
<tr>
<td>--mute, -m</td>
<td></td>
<td>Flag: Script informs only about current processed page, estimated time and corp stats</td>
</tr>
<tr>
<td><strong>Input</strong> (all combinations are allowed):</td>
<td></td>
<td></td>
</tr>
<tr>
<td>--paths, -p</td>
<td>path([s] path)*</td>
<td>Local paths to database dumps</td>
</tr>
<tr>
<td>--dumpUrls, -d</td>
<td>url([s] url)*</td>
<td>Remote paths to database dumps</td>
</tr>
<tr>
<td>--articleName, -a</td>
<td>name([s] name)*</td>
<td>Article names to be downloaded through wiki api</td>
</tr>
<tr>
<td><strong>Output:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--output, -o</td>
<td>outputPath</td>
<td>Output path</td>
</tr>
<tr>
<td>--outputFormat, -f</td>
<td>[txt</td>
<td>se]</td>
</tr>
</tbody>
</table>

Table A.1: Command line options
| Punctuation | Zdeňce byly vydány spisy jejího otce, <err>jenže...</err><corr>jenže</corr> stačilo málo a z „dcery“ se stal národní štvaneč. |
| Vzniklé vakuum vyplnila jediná politická strana<err/>,</corr> Národní souručenství, jež měla představovat oporu protektorátní vlády a správy. |
| Typographical | „Scatman(Ski-Ba-Bop-Ba-Dob-Bop)" je <err>skladbaamerického</err><corr>skladba amerického</corr> eurodance hudebníka Scatmana Johna. |
| Typographical | Vystudoval Vyšší průmyslovou školu strojnickou v <err>Ostravě - Vítkovicích</err> <corr>Ostravě–Vítkovicích</corr>, kterou kolem roku 1941 zakončil maturitní zkouškou. |
| Typographical | Na dvou pramenech jsou studánky, z nichž jedna se <err>nazýváHolubí</err> <corr>„Holubí"</corr> oko<err/>,</corr> č. druhá je bezejmenná. |
| Spelling | V <err>domě</err><corr>době</corr>, kdy byla pro nemoc upoutána na lůžko, začala psát a skládat písně </corr> |
| Spelling | Podle některých zdrojů <err>byli</err> <corr>byly</corr> v kabině nákladního vozu tři osoby. |
| Spelling | Černobílé, chlupaté vosy připomínající mravence se nazývají pandou, protože jejich vzhled připomíná Pandu <err>Velkou</err> <corr>velkou</corr> |

Table A.2: Samples of error types I
<table>
<thead>
<tr>
<th>Lexico-semantic</th>
<th>30. března 1921 Einstein odjel do New Yorku přednášet o &lt;err&gt;jeho&lt;/err&gt; &lt;corr&gt;své&lt;/corr&gt; nové teorii relativity.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stylistic</td>
<td>Svealand má zvlněné &lt;err&gt;ledovcové hřebeny&lt;/err&gt; &lt;corr&gt;hřebeny ledovcového původu&lt;/corr&gt; a většinu z více než 90 tisíc švédských jezer.</td>
</tr>
<tr>
<td>Stylistic</td>
<td>V r&lt;err&gt;.&lt;/err&gt;&lt;corr&gt;oce&lt;/corr&gt; 1991 se egyptské vojsko účastnilo operace Pouštní bouře na straně vojsk koalice.</td>
</tr>
<tr>
<td>Stylistic</td>
<td>&lt;err&gt;V&lt;/err&gt;&lt;corr&gt;Jan van Amstel se narodil v Amsterdamu, v&lt;/corr&gt; roce 1528 se přestěhoval do Antverp.</td>
</tr>
<tr>
<td>Stylistic</td>
<td>Koncem roku 2016 proběhla bulvárními i seriózními médií kauza BMW Invelt, do které byla &lt;err&gt;vědomě či nevědomě zapletena&lt;/err&gt;&lt;corr&gt;zapletena&lt;/corr&gt;.</td>
</tr>
<tr>
<td>Stylistic</td>
<td>&lt;err&gt;Tři&lt;/err&gt;&lt;corr&gt;Podle prvních informací tři&lt;/corr&gt; lidé zemřeli, osm dalších bylo zraněno.</td>
</tr>
<tr>
<td>Unclassified</td>
<td>Kazujoši Miura (podnikatel) (1947-. 2008) - japonský &lt;err&gt;vrah&lt;/err&gt;&lt;corr&gt;podnikatel podezřelý z vraždy&lt;/corr&gt;</td>
</tr>
<tr>
<td>Unclassified</td>
<td>&lt;err&gt;Znovu se vrátil&lt;/err&gt; &lt;corr&gt;Zůstal&lt;/corr&gt; pak členem Národního divadla do roku 1909, (...)</td>
</tr>
<tr>
<td>Unclassified</td>
<td>&lt;err&gt;Jed blokuje činnost svalů,&lt;/err&gt; &lt;corr&gt;Dospělá&lt;/corr&gt; chobotnice skvrnitá jej má dostatek na usmrcení 26 lidí.</td>
</tr>
</tbody>
</table>

Table A.3: Samples of error types II