**Ministry of Science and Higher Education**
**of the Russian Federation**
**ITMO University**

**Faculty of Digital Transformations**
**Educational program Big Data and Machine Learning**
**Subject area (major) Applied Mathematics and Informatics**

# REPORT

on practical training tech internship

Task topic: Machine Learning Technology for Correcting Electronic Medical Texts in Russian
Student: Dmitry Pogrebnoy, J42332c

Head of Practice from the trainee's host organization: Sergey Kovalchuk, PhD.
Head of Practice from ITMO University: Semyon Kraev, PhD

Practice completed with grade ____
Commission member signatures:

__________________________ full name
(signature)

__________________________ full name
(signature)

__________________________ full name
(signature)

Date ____________

St. Petersburg
2022

# Contents

# Introduction

In today's world, digitalization is rapidly spreading into all areas of our lives. Electronic document management makes it possible to work more efficiently with documents, increases the speed of communication and improves the quality of work of many employees in different professions.

Due to breakthrough achievements in natural language processing in the last two decades it has become possible to build various intelligent systems and machine learning models based on unstructured document storages, of which there is a significant amount. These systems can be applied in many fields, such as medicine and healthcare.

In medicine and health care, it is possible to build various predictive models and decision-making models based on information from patients' medical records, which can potentially improve the quality and effectiveness of treatment, and even save people from the severe consequences of various diseases. However, the accuracy and quality of such models is highly dependent on the quality of the input electronic records, which are usually presented as unstructured text.

One of the main problems complicating automatic processing of electronic records is various spelling errors in medical terms and simple words. According to a study by Toutanova et al. [1], spelling errors occur mainly for two reasons. The first reason is mainly related to the person himself and is that the writer may not know exactly how to spell a word correctly and therefore make mistakes. The second reason has to do with technology and is due to poor-quality typing devices, which can also lead to spelling errors. As a result, electronic documents with spelling errors significantly reduce the quality of the final models, which makes it impossible to achieve acceptable results.

In order to try to solve the problem of spelling errors, Balabaeva et al. [2] conducted a research that resulted in the development of a tool to correct spelling errors in Russian medical texts. Despite the impressive accuracy metrics obtained by the tool, there are still many directions to improve the final result and the speed of the tool, thereby further improving the quality of the source texts, which will positively affect the quality of medical models.

## Problem statement

The purpose of this work is to design a method and implement a tool for automatic spelling correction of medical texts in Russian. The tool should accept raw medical text as input and return corrected text with a minimum number of incorrect words.

In order to achieve the goal, the following objectives were set.

— Perform an overview of the Russian medical texts correction.

— Analyze existing solutions in field of correcting Russian texts.

— Design a new method for correction of spelling of medical texts in Russian.

— Design of the architecture and implement of a new spelling correction tool.

— Conduct an evaluation and approbation of the developed tool.

— Compare results of the developed tool and existing ones.

# 1 Overview of the spelling correction

This chapter provides general information on correcting errors in texts. Types of errors in texts are described and a list of the types of spelling errors that this paper focuses on is highlighted. An overview of the different edit distances is also made. In addition, a general approach to automatic correction of errors in texts is described and a generalized architecture of such tools is presented. A overview of relevant articles devoted to the correction of English, Russian, and separately medical texts is also made.

## 1.1 Types of spelling errors

The development of any system related to correcting errors in real texts requires determining the types of errors that the system should correct. For example, there are several fundamental works [3, 4, 5] for the English language that investigate the mechanisms and causes of errors in texts, as well as classify errors into various groups.

There are many classifications of errors in real texts, depending on the specific language and the task at hand. One of the most generalized and applicable to the Russian language is the classification by reason of an error [6, 1]. According to this classification, there are two main groups of errors.

1) Orthographic mistakes. Such mistakes are primarily related to the writer himself. A person may not know how to write a specific word or is just learning a new language and make mistakes even in simple words. In addition, the writer may also have some cognitive disorders that do not allow him to write a word without mistakes. In addition, the language may have specific rules for writing words that distinguish the pronunciation of a word from its spelling, which also leads to errors.

2) Type mistakes. These errors are mainly related to the technical devices of the writer. Sticking or unpressed keys on the keyboard, a tight spacebar, and the substitution of two close keys due to quick typing on a low-quality keyboard. All of these errors are related to the input device in one way or another, and almost do not depend on a particular language. At the same time, the keyboard

layout has a great impact, which can both significantly increase and decrease the number of such mistakes.

Unfortunately, this classification is quite general and cannot fully reflect the peculiarities of the Russian language. That is why we will use the following classical classification of errors for the Russian language.

1) Grammatical mistakes. Errors related to incorrect word formation.

2) Punctuation mistakes. Errors associated with incorrectly placed or missing punctuation marks in a sentence.

3) Spelling mistakes. Errors connected with a misspelled word.

This paper will focus mainly on correcting misspellings and will not deal with other groups of errors.

## 1.2 Edit distance

In 1966, the mathematician Levenshtein introduced [7] the edit distance model to estimate how much one string differs from another. This distance is the minimum number of operations to delete a character, insert a character and replace a character in order to get from one string to another. Later, Damerau and Levenshtein added another operation of transposition of two neighboring characters [8]. Damerau found that about 80% of spelling errors in English belong to the group of errors with an Damerau-Levenshtein edit distance of one.

The edit distance metric greatly depends on what operations are allowed with symbols. Therefore, there are several different edit distance metrics.

— Levenshtein distance [7]. Uses the operations of inserting a character, deleting a character, and replacing a character.

— Damerau–Levenshtein distance [8]. In addition to the operations of inserting a character, deleting a character and replacing a character, the transposition of two adjacent characters is also used.

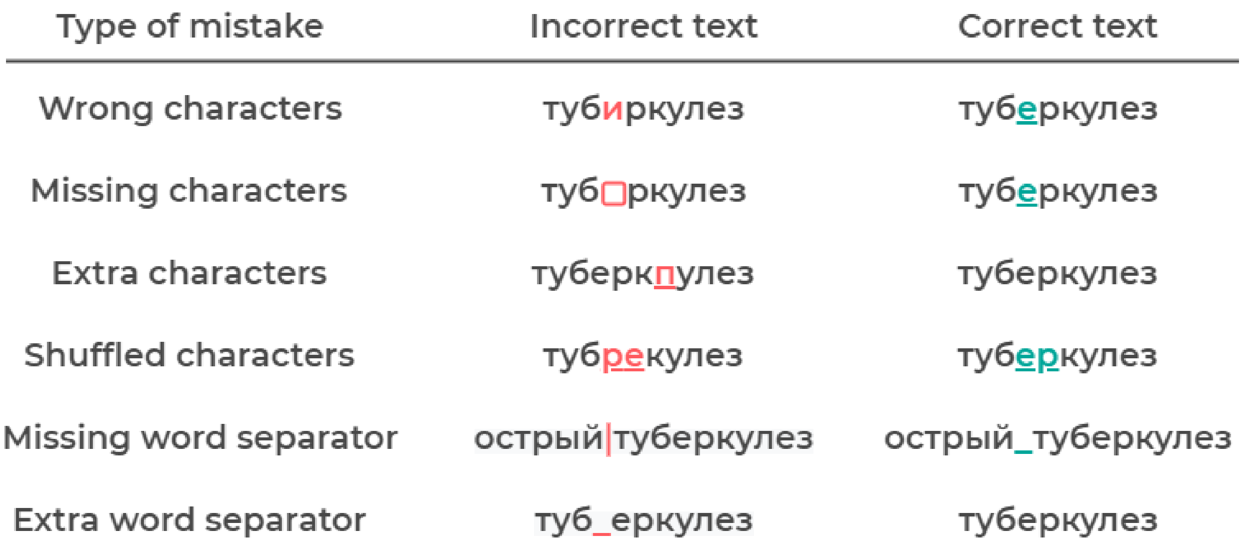
— Longest common subsequence (LCS) [9]. Only inserting and deleting a character is allowed, without substituting a character.

— Hamming distance [10]. Only character substitution is allowed. Due to this limitation, this metric can only be applied to strings of the same length.

— Jaro–Winkler distance [11]. Uses only transpositions of two characters in a string.

Based on the edit distance model operations, the following types of spelling errors naturally stand out. Examples of spelling mistakes are shown in the Figure 1.1. Various combinations of basic errors are possible, and the more errors a word contains, the more difficult it is to correct it. Therefore, most often words with one or two errors are correctable, and with three or more they can no longer be effectively corrected using the edit distance.

Figure 1.1 — Examples of spelling mistakes.



| Type of mistake | Incorrect text | Correct text |
|---|---|---|
| Wrong characters | тубиркулез | туберкулез |
| Missing characters | туб□ркулез | туберкулез |
| Extra characters | туберкпулез | туберкулез |
| Shuffled characters | тубрекулез | туберкулез |
| Missing word separator | острый\|туберкулез | острый_туберкулез |
| Extra word separator | туб_еркулез | туберкулез |

## 1.3 Automatic spelling correction

Modern tools for correcting spelling errors can help people even in the most ambiguous cases. However, the process of operation of such systems can be generalized to the following three stages.

1) Error detection.
2) Candidate proposal.
3) Ranking candidate.

These generalized stages are present in every stand-alone text correction tool. Some variation is possible, but in general these stages are applicable to the description of most such tools. Below we will look at each stage of correction in detail.

At the first stage, it is necessary to detect the presence of an error in the word. For this purpose, prepared dictionaries are usually used. The logic of the tool's work with the dictionary is quite simple. If the word in question is contained in the dictionary, then everything is fine with it, otherwise the tool considers that this word is written with an error and requires correction.

Once the tool has detected an invalid word, the correction process is performed. At first, a list of candidate words for correction is computed. Typically, the same prepared vocabulary is used for this as for detecting incorrect words. A predefined edit distance between the incorrect word and the dictionary word is calculated for each word. Usually the editing distance is limited to one or two, as a larger distance requires more calculations and greatly impairs the performance. Words with larger edit distances are not included in the list of candidates. As a result we get a list of possible correction candidates. Thus the problem of choosing the most suitable word among all candidates arises.

To find the most appropriate candidate word, usually some language model is used that is trained on texts with similar subject matter. Such language model allows one to use information about context, structure, and language features for better correction of spelling errors. For each candidate word, the model calculates the value of some metric which allows you to estimate how suitable the word is for fixing the incorrect one. In this way the model allows ranking the candidates from the most suitable to the most unsuitable. As a result, after ranking, the most suitable candidate is selected and replaces the incorrect word.

Stages of the spelling correction tools are shown in the Figure 1.2.

The effectiveness of a correction tool and its quality metrics depend largely on the vocabulary of correct words and on the quality of the language model. The more complete and error-free the dictionary is, the more words are correctly recognized by the tool as incorrect.

However, in practice it is impossible to assemble a dictionary that will contain absolutely all possible words. Real words may not be dictionary words and therefore also be recognized as incorrect. Therefore, already at this stage there are errors associated with recognition and obviously correct words are

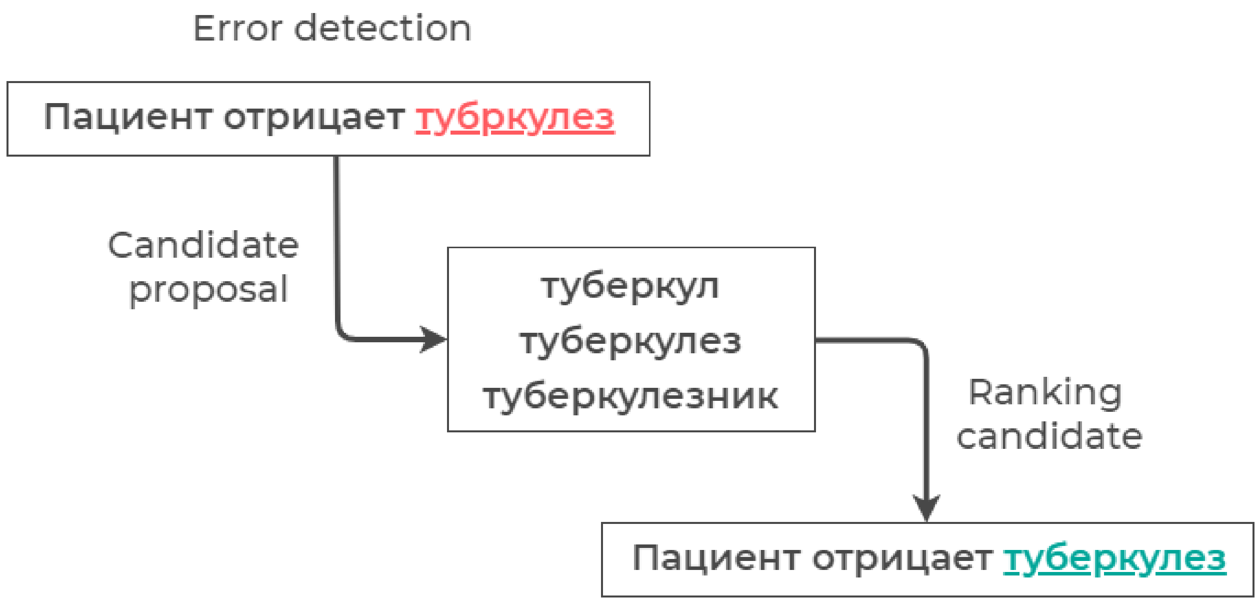


Figure 1.2 — The automatic spelling correction process.

recognized as incorrect. In addition, the prepared dictionaries are not perfect and may contain incorrect words. Thus, the tool can skip an incorrect word because it is in the dictionary of correct words.

Some of these problems can be solved with advanced error and language models. Such models, in addition to generating and ranking candidates for correction, can also consider the original word. Thus, the original word can perform as an edit candidate and be ranked along with the other candidates. This allows the language model to leave the original word as is if the word is considered incorrect, but none of the usual edit candidates is better suited than the original word.

It should also be noted that in practice it is almost impossible to prepare a large enough corpus without errors to train the language model. Therefore, errors in the word correction tool appear at this stage as well. The cleaner and better the training corpus of the language model, the more correct ranking it will show.

However, although it is almost impossible to get rid of these problems altogether, we should try to do what we can to improve the quality of the vocabulary and the language model.

## 1.4 Principle design of spelling correction tools

Special attention should be paid to the architecture of such tools. Typically, a spelling correction tool has pre-processing and post-processing components, as well as language and error models. The generalized architecture is shown in the Figure 1.3. Take a closer look at each component.

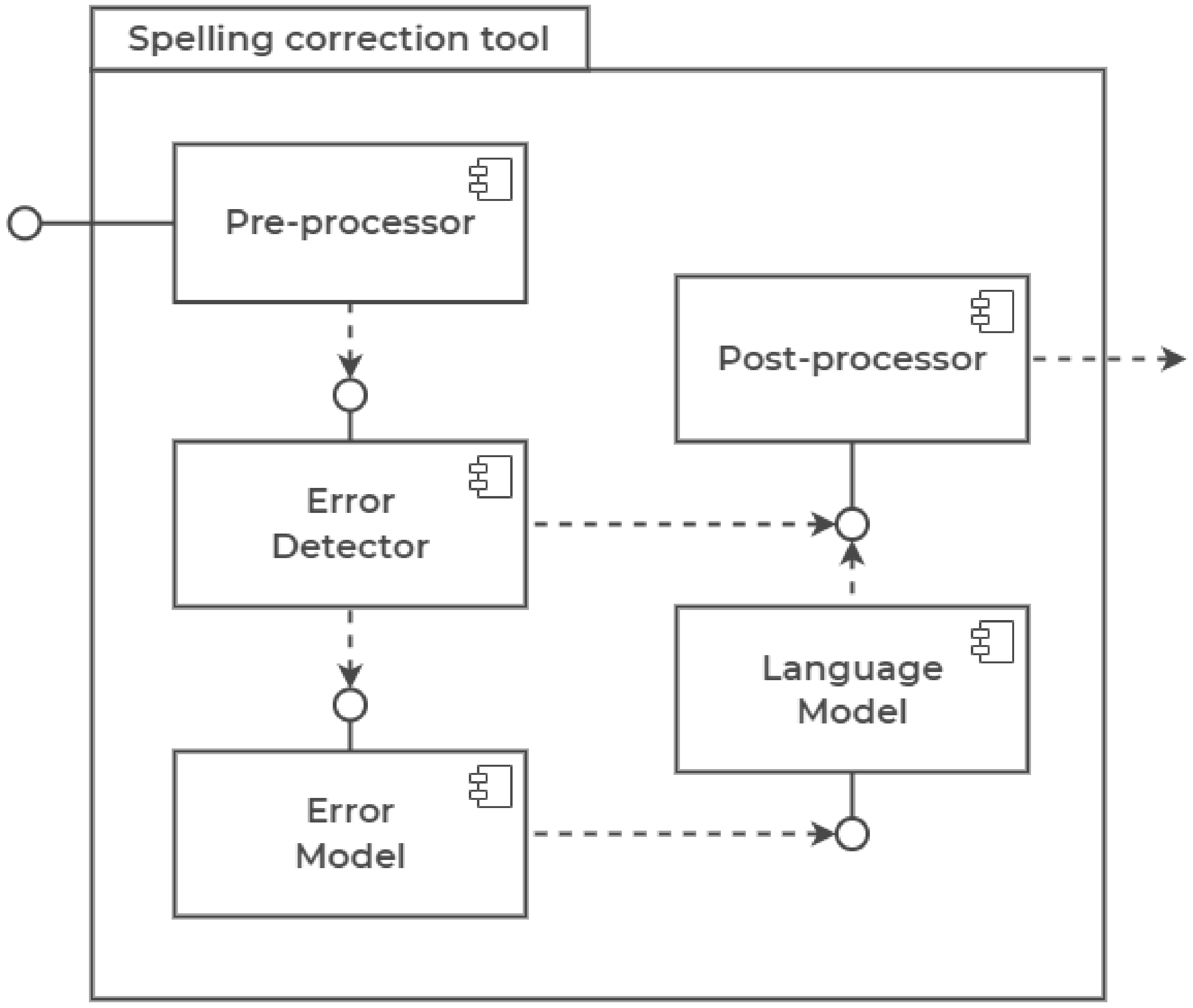


Figure 1.3 — Principle architecture of the spelling correction tool.

Pre-processor is responsible for pre-processing the input data for further processing within the tool. The component accepts text for correction, removes punctuation, and splits text into separate words. This component may also be responsible for extracting lexemes from separate words if the tool logic requires it.

Error Detector checks if there is an error in a particular word. And if the word is recognized as incorrect, then a further process to correct it starts, otherwise the word is skipped and gets into the final result as it is.

As already described in Section 1.3, usually a prepared dictionary is used for this purpose, which should contain only correct words. If a word is present in such a dictionary, it is considered correct. Otherwise, the word is considered a word with an error.

Error Detector component may also be responsible for word filtering. This can be useful for handling abbreviations or any names that are spelled with a capital letter. In this case, such words will be ignored and forwarded directly to the Post-processor component.

Error Model generates a list of potential candidates for which an incorrect word can be corrected. As described in Section 1.3, this usually uses some given edit distance and its operations. Applying valid editing distance operations to an invalid word yields a set of words. Those contained in the prepared dictionary of correct words are added to the list of potential candidates. The number of possible operations on the word increases exponentially with the increasing value of the editing distance. Therefore, for the performance reasons, a distance of one or two is typically used. In addition, according to the study [8], the majority of errors in words are at an edit distance of 1 or 2 from the correct word.

Language Model component is responsible for ranking the candidate list obtained from the error model. The language model can vary greatly from tool to tool. It can be built on frequency dictionaries, which rank words by their frequency of use in a thematic corpus of texts. Or the model can be built on the basis of n-grams. For example, when words around an incorrect word and a corpus of thematic texts are used to rank candidates. Also, various models based on embeddings are used in relevant works. Such language models are built on the basis of such models as Word2Vec [12], FastText [13, 14], ELMo [15], GloVe [16] and others. In the most recent works, language models often use neural networks, such as modifications of the BERT [17] neural network. As a result, after ranking, the component outputs the most appropriate candidate as a fix.

Post-Processor component reassembles the final text from checked and corrected words, and arranges punctuation marks. If necessary, this component can also put the corrected word or lexeme into the desired word form. As a result, this component returns a final corrected text.

The described components are basic and are present in almost all tools for correcting spelling errors. Of course, each tool is unique and has its own special features. Therefore, in addition to the components described above, there may be other equally important functional parts of the tools.

## 1.5 Related works

There are a number of research papers devoted to different approaches to correcting spelling errors. The most advanced tools and articles focus on correcting spelling errors in English texts.

One of the major relevant studies is that of Daniel Hládek et al. [18]. This paper provides a basic overview of the various approaches to text correction and makes an outline of most of the work in this field over the past 20 years. The study pays a lot of attention to methods and techniques of text correction for quite different languages. Among others, several articles aimed at correcting Russian texts are mentioned. In this way, this paper is a compilation of theoretical foundations and describes the progress in text correction to date.

Another relevant work is a new framework for correcting spelling mistakes in English created by Sai Muralidhar Jayanthi et al. [19]. In their paper they present a new open-source tool, which is based on 10 different pre-trained models. The distinguishing feature is that the models are trained on synthetically generated misspellings in words, which the authors say has increased the correction rate by 9 percent. In addition, this toolkit stands out because it takes into account the context around the word to be corrected, which greatly increases its accuracy relative to other well-known tools.

A paper by Yifei Hu et al. [20] investigates the approach of applying the BERT neural network and editing distance to the correction of spelling errors in texts. The authors consider two cases. The first case first applies a BERT model that predicts by context the word in place of the incorrect word. And then the word with the lowest editing distance and a larger metric is selected from the list of candidates. In the second case it is vice versa. First the edit distance is applied and a list of candidates is generated, and then for each candidate the

BERT model is used to calculate how well it fits the context. The candidate with the highest metric is selected as the corrected one. As a result, the second method was the best with 84.91% accuracy.

However, for the Russian language, the correction of spelling errors in texts is much less developed. Nevertheless, there are several relevant articles aimed to this topic.

### Correction of Russian texts

First of all, it is necessary to note the article by Alexei Sorokin et al. [21] about the SpellRuEval[1] competition on automatic correction of spelling errors in Russian media texts. Seven teams from different Russian universities participated. As a result, the winner was the team from Moscow State University, who published the results of their work in the article [22]. In order to win the competition they created a tool based on editing distance and phonetic similarity of words to generate a list of candidates, as well as using a probabilistic language model followed by the use of logistic regression to correctly rank the candidates.

Also one of the most recent articles is the work of Alla Rozovskaya [23] in which she uses three different language models to compare quality metrics for spelling error correction on three different datasets. Two of the models do not take context into account, while the other model does. According to the results of the experiments, the context-based language model showed the best results, which once again proves that context must be taken into account for a better word correction process.

There is also an article by Alexey Sorokin [24] describing a tool for correcting errors in Russian text based on the context noisy channel model and re-ranking the best candidates using logistic regression. According to the results of the experiments the obtained tool obviously outperforms the winners of the SpellRuEval contest.

Only a few works are focused to correcting spelling errors in medical texts.

[1]Official website of the SpellRuEval competition: `https://www.dialog-21.ru/evaluation/2016/spelling_correction/`

### Correction of medical texts

One such study is the work of Lai et al. [25] which investigates an approach for correcting spelling errors using the Aspell [26] tool for generating candidates and a developed noisy channel model for ranking the resulting candidates. The resulting tool was tested on several types of datasets with medical data. As a result, the obtained tool significantly outperformed Aspell on all metrics.

Another relevant study is the article by Pieter Fivez et al. [27], which presents an approach to editing spelling errors in medical texts based on an unsupervised context model using symbols n-gram embeddings. The resulting model outperforms the metrics of the model built by Lai et al. [25].

The only relevant article focused on the correction of spelling errors in Russian medical texts was published by Ksenia Balabaeva et al. [2]. In it the authors use the Levenshtein distance to generate a list of candidates and the trained FastText model to rank them. The results described in the article are impressive, according to the test results the best overall accuracy is 0.86.

It should be noted that despite the good results obtained in the article, the field of correction of Russian medical texts is underdeveloped and requires additional research. The results of which will help improve the quality of existing medical information systems and allow building new more advanced ones. This work will partially fill this gap.

## 2 Overview of existing tools

There are several popular and well-known available tools that can automatically correct spelling errors in words. Such systems are used in many systems that work with unstructured texts such as electronic documents, search queries, news, articles, and others. Thanks to such tools, the quality of the source text increases, which opens up opportunities for further intelligent processing and the construction of more effective information systems.

Most of these tools focus mainly on English text, while the other languages fall by the wayside and cannot boast such a rich range of tools. Nevertheless, there are several tools that support the Russian language.

After analyzing the tools for correcting spelling errors, the following parameters were identified for comparison.

— Error precision. This is the percentage of words with an error that the tool correctly corrected relative to all incorrect words.

— Lexical precision. This is the percentage of correct words without mistakes that the tool does not correct relative to all correct words.

— Overall precision. This is the average of error precision and lexical precision.

— Average number of words processed per second.

Precision metrics and tool performance were evaluated on two different datasets. The first dataset contains correct and incorrect words without their context. This set contains 100 entries for each of the first four error types, 900 entries for the fifth error type and 1000 for the sixth error type shown in Figure 1.1. Thus this test dataset contains a total of 2300 records. Each entry contains a pair of two words. The first word contains a particular type of error and the second word is its correct version.

The second dataset also contains 100 entries for each of the first four error types, 900 entries for the fifth error type and 1000 for the sixth error type shown in Figure 1.1. Thus this test dataset also contains a total of 2300 records. This set does not contain single words, but words with context. Each record consists of three parts. The first part is a coherent passage of 10 words, one of which is written with an error of a certain type. The second part is the same

10 words, but fully correct. The third part is the number of the incorrect word, which is used to calculate the error and lexical precision.

All words and passages in the test datasets are manually collected from various medical texts and anamneses in Russian.

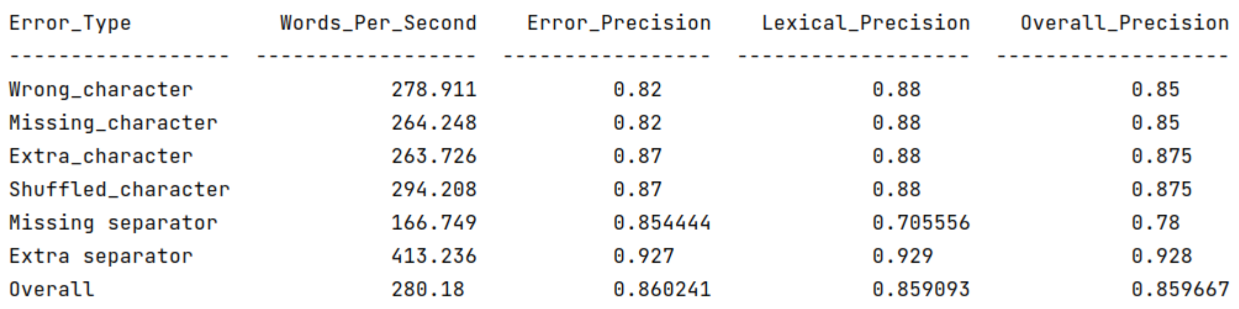
Thus we have two test datasets with Russian medical data. The first dataset allows us to evaluate how good the tool performs in correcting single words, and the second dataset allows us to calculate the quality of word correction in a coherent context.

In order to better compare the tools was also calculated overall precision, which is the average of lexical and error precision. This metric allows estimating the quality of spelling error correction in general. Also it should be noted that there are usually more correct words in a text than there are incorrect words, therefore lexical precision is more important than error precision.

However, in addition to the quality of correction of spelling errors, it is also necessary to take into account the time in which these errors are corrected. Of course, the faster the tool works, the better. The performance test was done on a laptop on Ubuntu 20.04 with 24 GB RAM and Intel Core i5-10210U CPU @ 1.60GHz * 8 and NVIDIA Tesla V100.

The metrics Words Per Second, Error Precision, Lexical Precision, and Overall Precision were calculated per each type of errors to provide more performance data and a deeper understanding of the strengths and weaknesses of existing tools. These metrics were then averaged across all error types.

Figure 2.1 shows an example of the test report output for the ASpell-python tool [28].



```
Error_Type           Words_Per_Second    Error_Precision    Lexical_Precision    Overall_Precision
------------------  ------------------  -----------------  -------------------  -------------------
Wrong_character                278.911           0.82                 0.88                 0.85
Missing_character              264.248           0.82                 0.88                 0.85
Extra_character                263.726           0.87                 0.88                 0.875
Shuffled_character             294.208           0.87                 0.88                 0.875
Missing separator              166.749           0.854444             0.705556             0.78
Extra separator                413.236           0.927                0.929                0.928
Overall                        280.18            0.860241             0.859093             0.859667
```

Figure 2.1 — Example of the test report output for the ASpell-python tool.

An important limitation is that the compared tools must be able to run from Python. This limitation is due to the fact that the medical text correction tool must be easily built into the medical text processing pipeline in the library created by ITMO University Digital Healthcare Laboratory. Therefore, wrappers or bindings of the tools in focus were used in cases where it was not possible to run the tool from Python.

Table 2.1 presents a comparison of some popular available tools for correcting spelling errors in Russian medical texts without context. Unfortunately, there are no production-ready available tools that focus on Russian medical texts at the moment. Therefore, only general-purpose tools are presented in the table.

Table 2.1 — Comparison of popular spelling correction tools for correcting spelling errors without context.

| Tool name | Open source | Error precision | Lexical precision | Overall precision | Average words per second |
|---|---|---|---|---|---|
| Aspell-python [28] | + | **0.86** | 0.859 | **0.859** | 283.7 |
| PyHunspell [29] | + | 0.812 | 0.539 | 0.675 | 9.4 |
| PyEnchant [30] | + | 0.829 | 0.541 | 0.685 | 20 |
| LanguageTool-python [31] | + | 0.762 | 0.904 | 0.833 | 25.1 |
| PySpellChecker [32] | + | 0.354 | 0.86 | 0.607 | 3.4 |
| SymspellPy [33] | + | 0.399 | 0.813 | 0.606 | **9702.8** |
| SymspellPy (compound) [34] | + | 0.465 | 0.512 | 0.489 | 672 |
| Jumspell [35] | + | 0.267 | **0.947** | 0.607 | 2552.1 |
| Spellchecker prototype [2] | + | 0.41 | 0.83 | 0.62 | 0.07 |
| Yandex.Speller [36] | - | 0.916 | 0.971 | 0.944 | 266.3 |

The first three tools are Python wrappers of the well-known general-purpose spelling error correction tools Aspell [26], Hunspell [37], and Enchant [38], respectively. The best result of these three was shown by the Aspell tool, which is used to correct spelling errors in Linux distributions.

The following are metrics for the LanguageTool Python wrapper. LanguageTool is written in Java and uses a rule-based approach. This tool is used to fix texts in many systems. For example, in the Grazie [39] plugin for the popular integrated development environment Intellij IDEA [40].

The following are the metrics for the PySpellchecker tool, which implements Peter Norvig's approach described in the original article [41]. This tool showed the lowest error precision metrics.

Also shown in the table are the metrics of the Symspell tool Python wrapper. The Symspell [33] is based on the new Symmetric Delete spelling correction algorithm, which reduces the complexity of generating edit candidates and dictionary searches for a given Damerau-Levenstein distance. This tool is language-independent and requires only a dictionary of all possible words and a frequency dictionary of the subject texts.

It is worth noting that Symspell has two modes of operation. In normal mode, the tool corrects spelling errors, while in compound mode, the tool also tries to split coupled words. Therefore, the normal mode has an impressive performance, while the compound mode has higher error precision metric.

In addition, the table shows a modification of Symspell, the Jumspell [35] tool, which uses a language model in addition to the error model, and as a result has a higher lexical precision metric than Symspell. The tool is written in C++, but has Python bindings.

Also the table contains metrics of the prototype from the article by Balabayeva et al [2]. Unfortunately, the tool described in the article could not be obtained, but the prototype of this tool has been preserved. This instrument has an extremely low performance, but it has a good lexical precision.

And the last tool is a Ynadex.Speller [36]. This tool, unlike all the others, is proprietary and provided as a service. This service has a strict user agreement which prohibits the use of the tool for commercial purposes, limits the number of requests per day and imposes other important restrictions. In many cases, such strong constraints are a significant barrier to the use of this tool. Nevertheless, Yandex.Speller shows outstanding performance and precision metrics that far outperform those of its open-source analogues.

Table 2.2 provides a comparison of some popular available tools for correcting spelling errors with context.

As in the single-word test, the ASpell tool wrapper was one of the highest overall precision results among open-source tools. PyHunspell and PyEnchant tools showed considerably lower metrics. They have the highest loss in the lexical precision metric.

The LanguageTool wrapper tool showed a result close to the ASpell wrapper with an overall precision metric of 0.835. The error precision and lexical precision metrics are slightly different from those of ASpell, but they are very small differences. The most notable difference is in the performance of these

Table 2.2 — Comparison of popular spelling correction tools for correcting spelling errors with context.

| Tool name | Open source | Error precision | Lexical precision | Overall precision | Average words per second |
|---|---|---|---|---|---|
| Aspell-python [28] | + | **0.731** | 0.93 | **0.831** | 357.3 |
| PyHunspell [29] | + | 0.706 | 0.719 | 0.713 | 11.8 |
| PyEnchant [30] | + | 0.721 | 0.719 | 0.72 | 24.3 |
| LanguageTool-python [31] | + | 0.727 | 0.942 | **0.835** | 43.6 |
| PySpellChecker [32] | + | 0.304 | 0.868 | 0.586 | 6.7 |
| SymspellPy [33] | + | 0.37 | 0.913 | 0.642 | **26060.2** |
| SymspellPy (compound) [34] | + | 0.483 | 0.804 | 0.643 | 1604.2 |
| Jumspell [35] | + | 0.307 | **0.969** | 0.638 | 4322.3 |
| Spellchecker prototype [2] | + | 0.25 | 0.79 | 0.52 | 0.09 |
| Yandex.Speller [36] | **-** | 0.82 | 0.987 | 0.904 | 1308.8 |

tools. The LanguageTool wrapper performed about 8 times lower than ASpell, but has better lexical precision, which makes it more preferable.

The metrics of the PySpellChecker tool are significantly lower than those of previous tools, especially the very low error precision and performance metrics.

The SymspellPy tool in both modes showed impressive performance, but relatively low precision metrics. Compound mode showed slightly better results for error precision, but slightly worse for lexical precision relative to the regular tool mode.

The JumSpell tool showed relatively similar results in the overall precision metric to the SymspellPy tool. However, JumSpell has a rather low error precision metric and the highest lexical precision metric among open source tools. The performance of this tool is slightly more than 6 times lower than SymspellPy in standard mode, but this is not a problem, because the processing speed of more than 4000 words per second should be enough for almost all tasks.

The latest prototype spelchecker tool from the article by Balabayeva et al [2] showed incredibly low performance and error precision.

The Yandex.Speller tool is proprietary and is provided as a service, so it is not possible to determine which hardware configuration it runs on. However, its metrics are the highest in the table, giving confidence that existing open tools and the approaches used in them can be significantly improved.

It is important to note that none of the open source tools discussed have support for GPU computation and do not use it to improve performance. This fact opens up an additional opportunity to use the GPU for computation in the new tool, which will allow the use of more computationally intensive approaches and outperform existing open source tools.

It should be clarified that all the tools discussed have not been fine-tuned for medical texts. The default dictionaries and models provided by the developers of the tools were used for testing.

Also it should also be noted that there are also several commercial tools with closed source code, which are able to correct spelling errors in Russian texts. One of these is a commercial version of Jumspell [42] with an improved language model. Of course, there are many other proprietary closed tools for correcting spelling errors in Russian, including those aimed at medical texts. Webiomed [43], Sberbank AI Lab [44] and Quantori [45] are also involved in processing medical texts. However, they do not have public tools for spelling. Nevertheless, any commercial tools were not considered in this comparison.

### Required performance

In order to define the required tool performance, and then correctly evaluate the performance metric of the compared tools, it is necessary to extract information about the approximate number of words in the medical record. The average value of the number of words in the anamnesis will not suit, as it is necessary to quickly process most such documents, and the average value does not provide such information. Therefore, the decision on the required number of words processed per second will be based on percentiles. Calculations were made for 2,516 anamneses from private datasets provided by the Institute of Artificial Intelligence Problems of the Research Institute of the Russian Academy of Sciences and the Almazov National Medical Research Centre (Almazov Center). The corresponding percentile values are given in Table 2.3.

Table 2.3 — Percentile values according to the number of words in the anamnesis.

| Percentiles | 0.25 | 0.50 | 0.75 | 0.90 | 0.95 | 0.99 |
|---|---|---|---|---|---|---|
| Word number | 37 | 63 | 104 | 149 | 189 | 275 |

According to the data presented in the table, we will focus on the required performance of the tool equal to 150 words per second, which allows the tool to process almost every anamnesis in less than a second.

To summarize, among the reviewed tools, half of them do not meet the performance requirement, in addition to the fact that none of the tools shows acceptable accuracy in correcting medical texts in Russian. Thus, it is necessary to develop a tool that will satisfy the performance requirement and at the same time show the better accuracy metrics.

# 3 Overview of BERT models

This chapter describes the architecture of the language representation model BERT, as well as several popular models built on top of this architecture.

## 3.1 BERT model architecture

A group of scientists from Google Jacob Devlin et al. in 2019 published an article [17] presenting a new architecture of the language representation model called BERT — Bidirectional Encoder Representations from Transformers.

The architecture of the BERT model is a multi-layered bidirectional Transformer encoder, which is very close to the original implementation presented in the article by Vaswani et al. [46].

The article presents two models. The first model, $BERT_{BASE}$, is used for comparison with the Open AI GPT model[47]. The second model, $BERT_{LARGE}$, is used to achieve the best results in tests. This model consists of 24 layers with a hidden size of 1024 and a number of self-attention heads as 16. As a result, the total number of model parameters is 340 million.

The training of this type of model takes place in two stages. At the first stage, the model is pre-trained. At this step, the model is trained on unlabeled data as part of various pre-training tasks. After that, at the stage of fine-tuning, the model is trained for a specific task from the real world. To fine-tune the model for a specific task, first the model is initialized with weights from the pre-trained model, and then this model is fine-tuned to the labeled data of a specific task. This approach allows to adapt the same pre-training model for different tasks by fine-tuning to a specific task.

As an example, consider the masked language model (MLM) task for the BERT model. First, the source text is tokenized, service tokens are added and the necessary tokens are replaced with the $[MASK]$ token. After that, the resulting tokens are converted into a vector form and transferred to to the transformer layers. After that, the result obtained from the transformers is decoded into a token and we get a prediction of the word under the tag $[MASK]$.

A visual example of this process is shown in Figure 3.1.

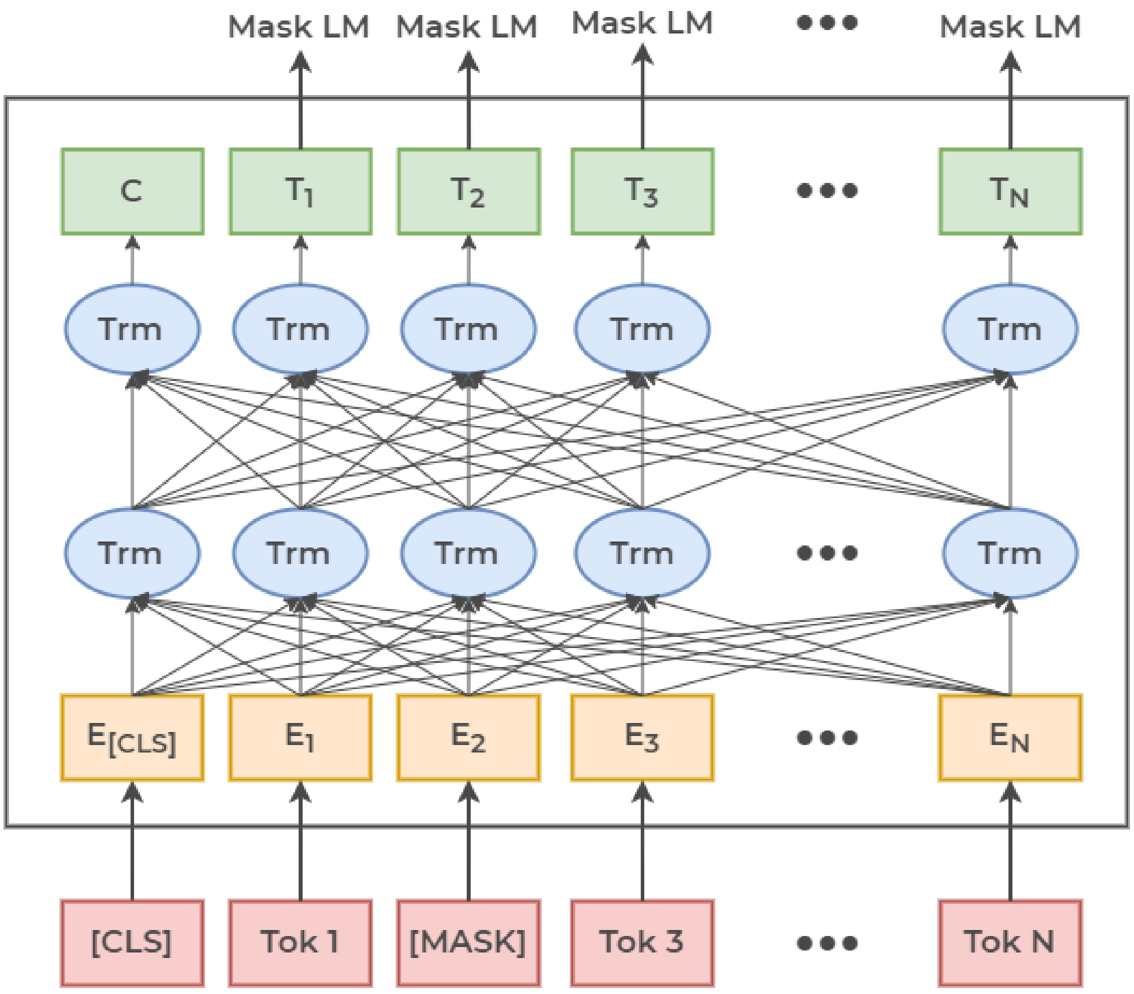


Figure 3.1 — Overall Masked LM procedure of BERT model.

The main key feature of this model, which made it possible to achieve such high results in the texts of the masked language model (MLM) task [48], is that the model uses the context not only to the left or right of the missing word, but simultaneously takes into account both parts of the context, which makes it possible to predict the missing word more precise. Similar feature are also applicable for other tasks, for example, for the Next Sentence Prediction (NSP) task.

There are several different BERT models that support the Russian language. The two original models are bert-base-multilingual-cased[1] and bert-

[1]bert-base-multilingual-cased model on Hugging Faces hub: `https://huggingface.co/bert-base-multilingual-cased`

base-multilingual-uncased[1], which have word case support and support 104 and 102 languages respectively. In addition, there is the Geotrend/bert-base-ru-cased[2] model, which supports only Russian and is obtained by discarding other languages from the original multilingual model. Another alexyalunin/RuBioBERT[3] model is the russian ai-forever/ruBert-large model[4] fine-tuned on a dataset of medical texts in Russian.

## 3.2 Popular BERT model modifications

The architecture of the BERT language model has become a breakthrough technology in the field of NLP and has gained great popularity. Many scientists and research groups wanted to contribute to the development of this approach, improve the architecture of the model and achieve even higher metrics. Next, several of the most famous and important modifications of the original BERT model will be described.

### RoBERTa

One of the most popular modifications of the BERT model is called RoBERTa. A group of Facebook AI scientists Yinhan Liu et al. presented this model in their research [49] in 2019. The authors found out that the BERT model is still under-trained and proposed an improved model pre-training process to achieve better performance.

One of the modifications is that RoBERTa uses a 10 times larger corpus of text than BERT, which includes not only Wikipedia articles but also web pages, books, and scientific papers. This allows RoBERTa to learn from a much more diverse set of texts and improves its generalization ability.

Another modification is that RoBERTa uses dynamic masking during pre-training. In BERT, the same tokens are always masked during training,

[1]bert-base-multilingual-uncased model on Hugging Faces hub: `https://huggingface.co/bert-base-multilingual-uncased`

[2]Geotrend/bert-base-ru-cased model on Hugging Faces hub: `https://huggingface.co/Geotrend/bert-base-ru-cased`

[3]alexyalunin/RuBioBERT model on Hugging Faces hub: `https://huggingface.co/alexyalunin/RuBioBERT`

[4]ai-forever/ruBert-large model on Hugging Faces hub: `https://huggingface.co/ai-forever/ruBert-large`

which can lead to overfitting. In RoBERTa, the masking pattern is randomly generated for each training example, which helps the model learn more robust representations.

RoBERTa also uses a larger batch size and longer training time than BERT. This enables the model to learn more complex patterns in the data and improves its performance on downstream tasks.

In summary, RoBERTa is a transformer-based language model that achieves high performance and shares a similar architecture with BERT. However, RoBERTa includes several modifications to its pre-training algorithm that enhance its performance. These modifications consist of using a more extensive corpus of text, applying dynamic masking during pre-training, using larger batch sizes and longer training times, and altering the method of handling sentence pairs during training.

There are several pre-trained RoBERTa models for the Russian language. One of them is the ai-forever/ruRoberta-large[1] model pre-trained by SberDevices team specifically for the Russian language. Another alexyalunin/RuBioRoBERTa[2] model is the ai-forever/ruRoberta-large model fine-tuned on a dataset of medical texts in Russian. This model was published in April 2022. By that time the DmitryPogrebnoy/MedRuRobertaLarge[3] model, which is presented in this paper, had already been created and was being prepared for publication.

### ELECTRA

One more popular modification of the architecture of the BERT model is called ELECTRA. This model is presented in the work [50] of Kevin Clark et al. in 2020. The key feature of this modification is an improved model pre-training process.

The authors present a new approach to retraining the model called «discriminative token detection». In this task, the model is trained to

[1]ruRoBERTa-large model on Hugging Faces hub: `https://huggingface.co/sberbank-ai/ruRoberta-large`

[2]RuBioRoBERTa model on Hugging Faces hub: `https://huggingface.co/alexyalunin/RuBioRoBERTa`

[3]MedRuRobertaLarge model on Hugging Faces hub: `https://huggingface.co/DmitryPogrebnoy/MedRuRobertaLarge`

distinguish between "real"and "fake"tokens, where the fake tokens are plausible alternatives generated by a small neural network called the generator. Thus, the model is trained on all input tokens, and not just on a masked subset, which usually makes up about 15% percent of all input tokens. In addition, the basic ELECTRA model has 110 million parameters, which is three times less than the RoBERTa model. Thus, the ELECTRA model takes up three times less space and makes inference faster than the RoBERTa.

Overall, the combination of the discriminative token detection task and the efficient pre-training process make ELECTRA comparable to the XLNet [51] and RoBERTa [49] models that has achieved state-of-the-art performance on many natural language processing benchmarks. The key features of the ELECTRA model make it more efficient and effective in representing language with much less train data and computation.

Unfortunately, there are no ELECTRA models with Russian language support at the moment.

### DistilBERT

Another modification of BERT is DistilBERT model. This model was presented by Victor Sanh et al. in their work [52] in 2019.

The architecture of this model is in many ways similar to the $BERT_{BASE}$ architecture, but a significant difference is that DistilBERT has half as many layers, which makes it significantly smaller than the original $BERT_{BASE}$ model. The authors claim that the key feature of the DistilBERT model is that it is 40% smaller and 60% faster than the original $BERT_{BASE}$. But despite the smaller size of the model, it retains 97% of the language understanding capabilities. This is achieved by using of several key techniques.

One of them is knowledge distillation, which involves training a smaller model to mimic the behavior of a larger and more complex model. In this case, DistilBERT is trained to mimic the behavior of BERT, which is a much larger and more complex model. This helps the model to capture the same level of linguistic knowledge as BERT but with fewer parameters.

Another technique used in DistilBERT is parameter sharing. In this approach, the parameters of adjacent layers are shared, reducing the number of parameters in the model. This also leads to faster training and inference times.

Finally, DistilBERT also uses layer dropping, which involves randomly dropping some of the layers during training. This forces the remaining layers to learn more robust representations, making the model more resilient to noise and input variations.

Overall, DistilBERT is a lighter and faster version of the BERT model, which is achieved by reducing the number of layers, attention heads, and parameters. More details about the training of the model and the results obtained can be found in the original article.

There are several pre-trained DistilBERT models for the Russian language. One of them is the original distilbert-base-multilingual-cased [1], which supports 103 other languages in addition to Russian. Another DistilBERT model is Geotrend/distilbert-base-ru-cased[2] model, which is derived from the work of Amine A. et al. [53]. This model was created from the original distilbert-base-multilingual-cased model by discarding support for all languages except Russian, which reduced its size and made it more usable.

### TinyBERT

TinyBERT is a small and efficient version of BERT model. TinyBERT was introduced in a 2020 paper by Jiao et al. [54] and is designed to address the limitations of BERT, which can be computationally expensive and memory-intensive for some applications.

TinyBERT is trained using a distillation approach, where the knowledge from a large teacher BERT model is transferred to a smaller student TinyBERT model by mimicking the teacher's behavior. Specifically, TinyBERT is pre-trained on a large corpus of text using a modified version of predicting masked tokens and next sentences. During this pre-training, the model is trained to capture the same language patterns and relationships as the teacher model.

[1]distilbert-base-multilingual-cased model on Hugging Faces hub: `https://huggingface.co/distilbert-base-multilingual-cased`

[2]Geotrend/distilbert-base-ru-cased model on Hugging Faces hub: `https://huggingface.co/Geotrend/distilbert-base-ru-cased`

TinyBERT uses several techniques to reduce its size and improve efficiency, including knowledge distillation, parameter sharing, and attention transfer. Knowledge distillation transfers the probabilities from the teacher model to the student model, allowing the student model to learn from the teacher's predictions. Parameter sharing involves sharing some of the parameters between the two models, reducing the number of parameters that the student model needs to learn. Attention transfer matches the attention patterns between the teacher and student models, enabling the student model to learn to attend to the same parts of the input as the teacher model.

In summary, TinyBERT achieves state-of-the-art performance on several benchmark NLP tasks while using significantly fewer resources than BERT. This makes it a promising approach for building efficient and effective NLP models for resource-constrained environments.

In addition to the described model modifications, there are many other popular modifications and promising approaches. Although the field is rapidly evolving and new and better models are being created, the models described above are the best known, most available, and most widespread.

# 4 New Spelling Correction Tool

This chapter describes a new tool for correcting spelling errors in Russian medical texts. The first section describes the algorithm of the tool. The second section presents a general overview of the tool's architecture, and the third section details the implementation.

## 4.1 Russian anamnesis corpus

Developing an effective tool for correcting spelling errors in Russian-language medical texts is impossible without collecting the necessaryf data to train machine learning models and testing the resulting tool.

In today's world, real non-scientific open medical data sources and datasets are extremely limited. This is mainly due to the complexity of collecting and processing such data, as they can potentially contain personal data. There are different approaches and tools for removing personal data from texts, but not one of them guarantees the complete removal of all personal data. Nevertheless, there are several corpus of medical texts in Russian in the public domain.

One of the public datasets is RuMedNLI: A Russian Natural Language Inference Dataset For The Clinical Domain [55] by Pavel Blinov et al. This dataset is a manually translated from English MedNLI [56] dataset and contains 14717 medical records. Another public dataset is RuMedPrimeData [57] by Starovoytova Elena et al. This dataset contains 15250 medical anamneses of SSMU hospital visitors.

Private datasets were also used. One dataset with patients' medical anamneses was provided by the Institute of Artificial Intelligence Problems of the Research Institute of the Russian Academy of Sciences. This dataset contains 161 large fragments of patients' anamneses. In addition, a closed corpus of anamneses provided by the Almazov National Medical Research Center (Almazov Center) was also used. This dataset contains 2355 patient anamneses for the period from 2010 to 2015.

Each of the datasets was pre-processed. First all texts were tokenized using the Python package mosestokenizer[1]. Then all the words were lemmatised using the pymorphy2[2] tool. In addition to this, punctuation marks, stop words, numbers and words containing non-Russian characters were removed. A standard set of stop words from the nltk[3] library was used as stop words. In addition, all data was converted to lower case.

Such harsh pre-processing was done because of the extremely limited datasets. The lemmatisation helped to get rid of words in different forms and to use words in their initial form. This increased the number of example sentences for a particular word, which contributed to a better fine-tuning of the language models.

The lemmatisation operation discards information about the form of the word, which can make it difficult to apply some word processing approaches. However, lemmatization is a frequent operation in word processing, so this limitation is not so critical.

After preprocessing, all four datasets were combined into one. In total, all datasets together contain 30,737 medical records in Russian, which takes about 10.25 Mb.

As you can see, there isn't much data at all. However, this is all that was extracted and will probably be enough to correctly train a BERT model on this data. This is one of the reasons why it was interesting to try the BERT model approach and see what results it would show for Russian medical texts with such a limited dataset.

## 4.2 Algorithm

The diagram of the spellchecking process is shown in the Figure 4.1.

The process is arranged as follows. First of all, the medical text is divided into tokens. Next, a couple of conditions are checked. Whether there are any non-Russian letters in the token and whether the word is a name. To check if a word is a name, we check if the word does not contain any capital letters, or if

[1]Github repository of the mosestokenizer package: `https://github.com/luismsgomes/mosestokenizer`

[2]Github repository of the pymorphy2 package: `https://github.com/pymorphy2/pymorphy2`

[3]Github repository of the nltk package: `https://github.com/nltk/nltk`

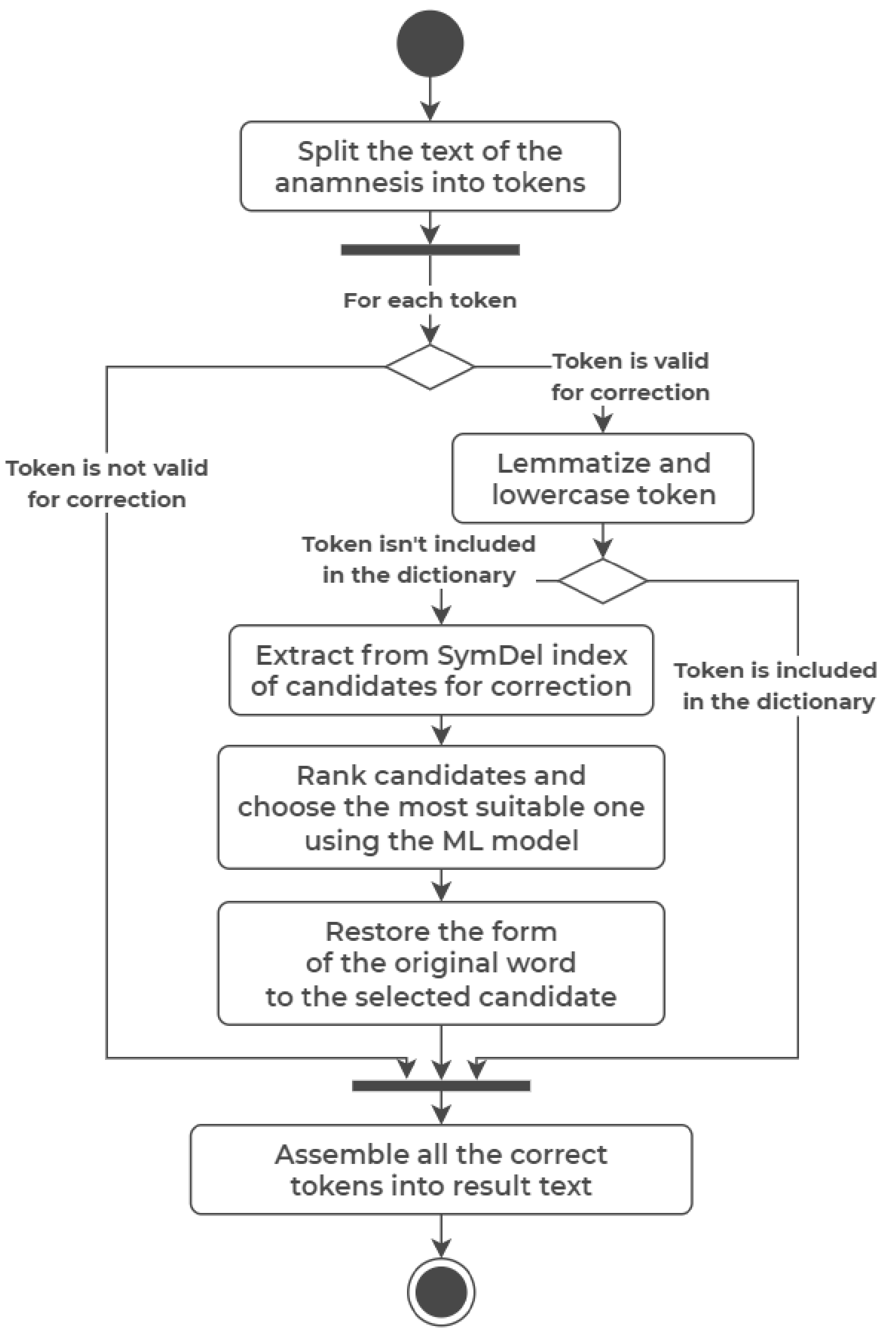

Split the text of the anamnesis into tokens
For each token
Token is valid for correction
Token is not valid for correction
Lemmatize and lowercase token
Token isn't included in the dictionary
Extract from SymDel index of candidates for correction
Token is included in the dictionary
Rank candidates and choose the most suitable one using the ML model
Restore the form of the original word to the selected candidate
Assemble all the correct tokens into result text

Figure 4.1 — The spelling correction process of the new tool.

it is at the beginning of a sentence. If at least one condition is not fulfilled, the token is considered as not valid for correction and gets into the final result as it is. Otherwise, the token is reduced to lowercase letters, and the lemmatized form of the token and information about the form of the original word is retained in the internal representation. After that, it is checked whether the token or its lemmatised form is included in the dictionary of correct words. If it is included, then such a token gets into the final text as it is. Otherwise, a list of candidates is generated to replace the incorrect word. Then this list is ranked by a special language model and the most suitable candidate gets into the final text. The best candidate is transformed into the required form and case of the original word. This happens with every token. At the end, the corrected tokens are assembled into the final text.

A slightly different schema is used to process and correct missing and extra spaces.

To process missing spaces, the following operations are performed. If the word is not contained in the dictionary of correct words, then all the partitions of the word into two parts are found, such that they are contained in the dictionary of correct words. Further, all such found pairs and generated candidates for correcting words for other errors are ranked by the language model. For a pair of words, a metric is calculated for each word and then the average of the two metrics is calculated. As a result, the option with the highest metric is selected and gets into the final text.

To handle extra spaces, a somewhat similar procedure is used. For each pair of words, an attempt is made to combine these two words into one by removing the space between them. If the resulted word is in the dictionary of correct words, then a metric is calculated for this word and it is ranked along with the other candidates for replacing.

Processing cases of missing and extra spaces, despite its external simplicity, requires significant calculations and greatly degrades the performance of the tool. To reduce this effect, it is necessary to develop more efficient algorithms and data structures to handle such cases, as well as more data for training the language model in order to correctly rank such candidates.

The algorithm has two modes. The first mode without processing missing and extra spaces and the second mode with processing cases with spaces. Depending on the need and further tasks, it is possible to choose one or the other mode.

## 4.3 Architecture

The new tool for correcting the spelling of medical texts is written in Python and supports working only with Russian text. The tool consists of seven components. The architecture of the tool is shown in the Figure 4.2.

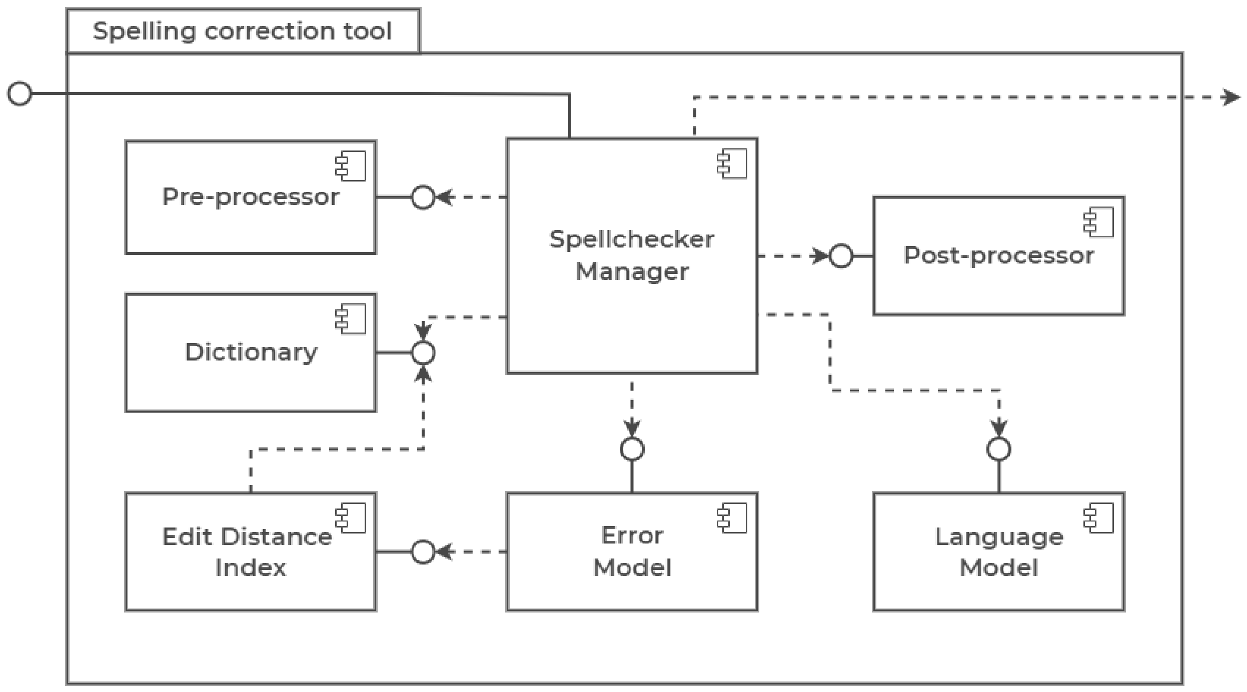


Figure 4.2 — Architecture of the new spelling correction tool.

The architecture of the tool is markedly different from the general architecture. The center of the architecture is the Spellchecker Manager component, which is responsible for coordinating the other components of the system and the top-level business logic of the tool. This feature allows the developer to independently replace the corresponding components, while preserving only the interface of interaction with them. Thus, it becomes possible to independently develop and improve each component individually without having to make changes to the entire system as a whole.

Like the principle architecture, the architecture of the tool includes pre-processing and post-processing components. The Pre-processor component is responsible for splitting the text into separate words, removing punctuation and capitalisation, and determining the lemmatised form and form information of the original word. In contrast, the Post-processor component restores the form of the original words to the corrected words and reassembles the entire text from the individual words.

The Dictionary component contains a dictionary of correct words and is used to quickly determine whether a word is correct or requires correction. This dictionary contains only lemmatised words from the Aspell Russian dictionary and several medical dictionaries. The final dictionary contains 214629 words in the primary form.

Error Model component is responsible for generating a list of candidates for correcting an incorrect word. Compared to the main architecture of the tool, this architecture has some differences. The error model uses the Damerau-Levenstein editing distance to create a list of candidates. This is a computation-intensive operation, therefore, a special index is used to significantly speed up the operation and improve the overall performance of the tool.

Language Model is responsible for ranking the editing candidates and choosing the most suitable one to replace the incorrect word. A finely tuned machine learning models based on the BERT architecture is used to rank candidates for Russian medical texts. The main advantage of this approach is that the model is able to take into account the context around the incorrect word when ranking candidates, which improves the quality of ranking and accuracy of correction.

To summarize, all components together fully provide the implementation of the process of correcting the necessary spelling errors described in the Section 4.2.

The following sections will detail the implementation and features of each described component.

## 4.4 Implementation

### Pre-processor

The Pre-processing component is responsible for splitting the text into separate words, removing punctuation and capitalization, , and determining the lemmatised form and form information of the original word. Incoming text is tokenized first and punctuation marks are removed using the sacremoses[1] library. Then a lemmatized form is extracted for each token using the pymorphy2[2] library. These libraries are the same as those used to preprocess the anamnesis dataset and the correct word dictionary. This is intentionally

[1]Sacremoses tokenizer library: `https://github.com/alvations/sacremoses`

[2]Pymorphy2 lemmatizer library: `https://pypi.org/project/pymorphy2/`

done to preserve word processing persistence and not bloat the dependency list of the new tool.

After lemmatisation, an internal representation of the tokens is generated from the resulting tokens, indicating whether the token is valid or not.

A valid token is one that is not in the nltk[1] of the library's stop word list, must not be less than four letters, and must not contain non-Russian alphabet characters. In addition, a token must not be a name, that is, it must not contain capital letters, unless it is the beginning of a sentence. All these factors indicate proper names or numbers and identifiers, which it is not worth trying to correct in order not to make more mistakes.

In addition to the lemmatized form of the word, a special tag is also added to the meta information, which contains full information about the form of the original word. This data later makes it possible to restore the corrected word to its intended form.

The resulting internal token representations with meta information about the lemmatization form and the tag of the form of the original word are then passed to the Dictionary component to determine whether the word contains an error or is completely correct.

### Dictionary

The Dictionary component contains a dictionary of valid words that are used to construct the Edit Distance Index and select edit candidates. This component is also used to check words from text for correctness. If a word is in the dictionary, it is considered correct. Otherwise, it is not.

The quality of error correction largely depends on the completeness and correctness of the collected vocabulary, as errors in the vocabulary can lead to missing text errors, and missing correct words can conversely lead to trying to correct a correct word.

In order to avoid this and to collect the fullest possible vocabulary of correct words, several public dictionaries have been merged. One of the

[1]Github repository of the nltk library: `https://github.com/nltk/nltk`

dictionaries is the dictionary of correct words from the ASpell tool[1]. This dictionary contains 128,900 words as well as a variety of word forms. Another dictionary is the Russian dictionary from OpenCorpora[2], which contains 395247 words and their various forms. The third dictionary is a private dictionary and contains 421420 different words. This dictionary includes various medical terms in addition to ordinary words. This dictionary was obtained from the work [2] of Balabaeva K. et al.

All three dictionaries have been pre-processed. All words have been lemmatised to reduce the volume of the vocabulary. Reducing the size of the dictionary greatly affects the size of the Edit Distance Index, so it is important not to bloat the dictionary. In addition, the dictionary does not need different word forms from the initial form, because the language model has been trained specifically to deal with words in the initial form only, and the Pre-processor component lemmatises all incoming text.

As a result, after pre-processing and combining the three dictionaries, the final dictionary contains 214629 words in their original form and takes up 4.6 Mb.

### Edit Distance Index

The list of candidates is generated by calculating the Damerau-Levenstein editing distance between an incorrect word and some correct word. If the distance is less than the preset value, then such a correct word gets into the list of candidates. Usually, the maximum editing distance for candidates is set as one or two. The developed tool supports working with both values of the editing distance. However, with an increase in the maximum editing distance, the number of operations for calculating editing distances also increases, since the words that require processing are much more. Therefore, in this work the edit distance of one is used.

In order to make a list of candidates, it is necessary to calculate the editing distance between the incorrect word and all the correct words from the dictionary, which differ in the number of characters in the word by no more

[1]Russian word dictionary from ASpell tool: `https://ftp.gnu.org/gnu/aspell/dict/ru/aspell6-ru-0.99f7-1.tar.bz2`

[2]Russian word dictionary by OpenCorpora: `http://opencorpora.org/dict.php`

than one. With this approach, at the moment of execution, it is necessary to calculate the editing distance a large number of times, which greatly reduces the performance of the tool when using this approach.

To improve performance, it is necessary to reduce the number of calculations of editing distances in runtime. To do this, it is most logical to use some index, which will avoid a large number of calculations. The most advanced approach to building an index is the approach used in the SymSpell [33] tool, the so-called SymDel (Symmetric Deletion) index.

SymDel index allows you to perform all calculations and build an index once at the start of the tool and then use it for the rest of the time.

This index is constructed over the whole vocabulary as follows. For each word in the dictionary, the operation of letter deletion in all possible places is applied. The index contains the words with the initial word after the deletion. When generating the candidates, the operation of letter deletion is also applied to the processed word and the resulting word is searched for in the index. As a result, the initial words found in the index are added to the list of candidates for correction.

Consider an example of constructing index entries for the word «домик» shown in Figure 4.3.

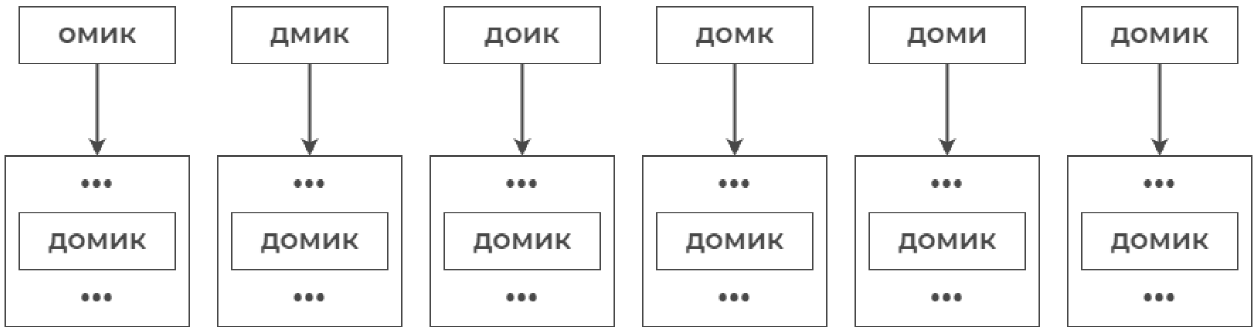


Figure 4.3 — The example of building index entries for the word «домик».

In order to build the necessary entries in the index with the maximum editing distance of one for the word «домик», the following operations happen. For the word «домик», all possible deletions of one letter from the word are made, as well as the original word itself. Then the produced words after deletion are written to the index and mapped to the original word. Since different words can generate the same delete results after deleting letters, the original words are saved in lists. Thus, each deleted result is mapped to a list of original words from which this delete result can be obtained.

When generating candidates for an incorrect word, all sorts of single letter deletions in the incorrect word also happen. The resulting delete words combined with the incorrect word itself, are searched in the index. As a result, we get a set of found lists of correct words that are candidates for replacement. Combining all these lists, we end up with the final set of candidates.

This generates all edit candidates at a distance of one. For a larger distance, it is necessary to remove the letters accordingly. The algorithm reduces the complexity of edit candidate generation and dictionary lookup by using deletes only instead of all four Damerau-Levenstein edit distance operations, thereby increasing performance many times over.

It's worth noting that increased productivity is not achieved for free. This approach significantly increases the amount of memory required to store the index. Therefore, in the future it will be possible to build the index on a data structure that allows you to compress the index without significant loss in performance.

At the moment, the index of the developed tool takes up a bit more than 4 GB of RAM, which is not a problem for modern computers and servers. However, as the dictionary increases, the amount of memory required will grow non-linearly and may require additional optimizations.

Next, a machine learning-based language model trained specifically for the task of ranking candidates for the replacement of an incorrect word will be considered in more detail.

### Error Model

Error model is used to find potential candidates for correcting an incorrect word. There are many different approaches to this task from probabilistic and statistical to machine learning model-based approaches. However, this paper uses the Damerau-Levenstein edit distance approach, as it is one of the most efficient methods used in a variety of tools and algorithms.

Editing distance is used to find correction candidates for incorrect words with errors not associated with missing or extra spaces. To find candidates related to errors with missing spaces a full search of all possible word divisions into two other words is performed. If at least one of the two obtained words is not contained in the dictionary of correct words, such a pair is discarded and further does not participate in the ranking. The concatenation of two consecutive words is used to find candidates for extra space errors. If the word obtained in this way is included in the dictionary of correct words, such a word is also discarded and does not participate in the ranking.

Calculating edit distance is a frequent operation for finding edit candidates. Such an operation must be high-performance and well-optimized. There are several popular packages with the implementation of edit distance calculation. In order to select the most efficient implementation, the performance of popular implementations was measured for insertion, deletion, replacement and transposition operations on 100 random words.

Table 4.1 presents the results of the performance measurements of the packages for calculating the Damerau-Levenstein editing distance.

Table 4.1 — Average time to compute the Damerau-Levenstein distance using different Python packages.

| **Package** | **Avg sec per word pair** |
|---|---|
| pyxDamerauLevenshtein [58] | 8.69 |
| fastDamerauLevenshtein [59] | 4.61 |
| jellyfish [60] | 2 |
| textdistance [61] | 3.1 |
| editdistpy [62] | 1.12 |
| StringDist [63] | 68.6 |

The most effective package is editdistpy. It is the one used in the new tool to calculate the editing distance that is far superior to its competitors.

In addition to the main error model, which uses the Damerau-Levenstein edit distance, a secondary model is also implemented. This model works the same as the main model, but uses the Levenstein edit distance. The developed tool supports the use of either the primary or the secondary model, depending on the user's preferences and the problem to be solved.

To make sure that the performance of the editdistpy package is also the best for calculating the Levenstein distance, the same performance tests of popular implementations as for the Damerau-Levenstein distance were performed. Table 4.2 presents the results of the performance measurements of the packages for calculating the Levenstein editing distance.

Table 4.2 — Average time to compute the Levenstein distance using different Python implementations.

| **Package** | **Avg sec per word pair** |
|---|---|
| editdistance-s [64] | 1.39 |
| jellyfish [60] | 0.55 |
| textdistance [61] | 3.04 |
| editdistpy [62] | 1.09 |
| stringdist [63] | 46.3 |

It should be noted that the editdistpy implementation is not the best, but it is good enough. The best result is shown by the jellyfish implementation. However, to reduce the dependency of the new tool on third party packages, the implementation of editdistpy is also used to calculate the Levenstein distance.

It is also worth noting that both performance tests were conducted on a laptop configuration with Ubuntu 20.04 operating system, Intel Core i5-10210U CPU and 24 GB RAM.

The resulting edit candidates from the error model are passed to the language model for ranking and selecting the most appropriate candidate. The next section describes the language model in more detail.

### Language Model

A machine learning model based on the BERT architecture is used as a language model for ranking candidates. This approach has proven itself well in similar tasks for correcting spelling errors in various languages. It was therefore interesting to apply this approach to Russian medical texts, that is, to a complex language of a narrow subject area with limited training data. The main advantage of this approach is that the model is able to take into account the context around the incorrect word when ranking candidates, which improves the quality of ranking and accuracy of correction.

In this paper, three different BERT-base models were fine-tuned. To adapt these models for the task of ranking editing candidates for correcting Russian medical words, the models were fine-tuned on a specially collected corpus 4.1 of medical anamnesis in Russian, which contained 30.737 records. The test and training parts were divided in the proportion 2:8.

All three models were fine-tuned on the ITMO University GPU cluster. For this purpose, a special docker image with the necessary dependencies and configuration for using CUDA GPU calculations was assembled[1].

Fine-tuning of the models took place using the transformers[2], datasets[3] and accelerate[4] libraries from the Hugging Faces platform. These libraries provide convenient tools for fine-tuning pre-trained models and have integration with the Hugging Faces hub.

Gradient Accumulation, Gradient Checkpointing, and Reduced floating point precision approaches were used in the fine-tuning to reduce the amount of GPU memory required to fine-tune the models.

### MedRuRobertaLarge

The first base model is pre-trained ruRoBERTa-large[5] model from Sberbank AI. This model was chosen first because it is one of the larger and more efficient models of the BERT architecture. This model is publicly available and pre-trained for the Russian language, which makes it easy to use for various applications in Russian.

The fine-tuning parameters of the model are presented in the Table 4.3

A special Python script[6] was written to fine-tune the model. The fine-tuning parameters are chosen experimentally. It was not possible to choose optimal hyperparameters because of the duration of the fine-tuning process.

The change in the perplexity metric on the train and test data after each epoch is shown in Figure 4.4.

[1]Docker image for fine-tuning models on the ITMO University GPU cluster: `https://hub.docker.com/r/pogrebnoy/medspellchecker`

[2]Transformers library: `https://github.com/huggingface/transformers`

[3]Datasets library: `https://github.com/huggingface/datasets`

[4]Accelerate library: `https://github.com/huggingface/accelerate`

[5]ruRoBERTa-large model on Hugging Faces hub: `https://huggingface.co/sberbank-ai/ruRoberta-large`

[6]Python script for fine-tuning ruRoBERTa-large model: `https://github.com/DmitryPogrebnoy/MedSpellChecker/blob/main/medspellchecker/ml_ranging/models/med_ru_roberta_large/fine_tune_ru_roberta_large.py`

Table 4.3 — Fine-tuning parameters of the ruRoBERTa-large model.

| **Parameter** | **Value** |
|---|---|
| Train epoch | 10 |
| Learning rate | 0.00005 |
| Weight decay | 0.01 |
| FP16 training | True |
| Gradient accumulation steps | 256 |
| Per device train batch size | 1 |
| Per device eval batch size | 1 |
| Gradient checkpointing | True |

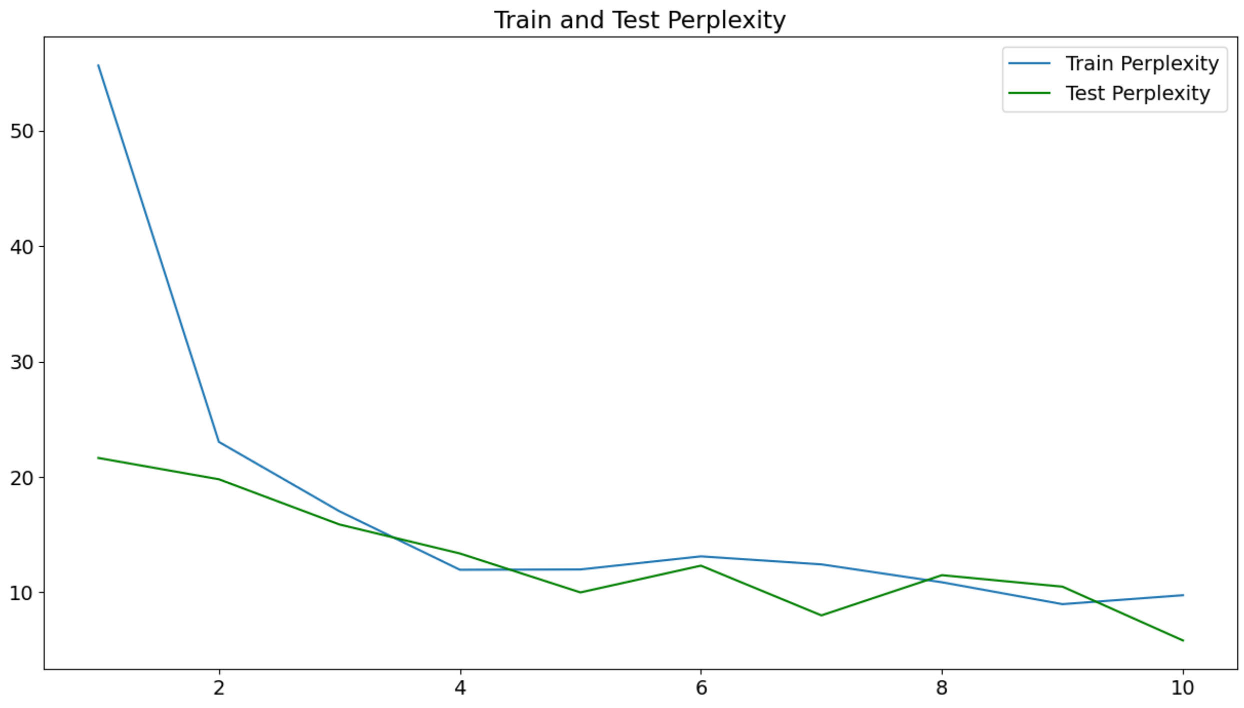


Figure 4.4 — Change perplexity metric on the train and test data after each epoch.

As you can see, the perplexity metric on test data has decreased from just over 22 to about 6 as a result of the training.

The fine-tuned model is published on Hugging Face as DmitryPogrebnoy/MedRuRobertaLarge[1].

[1]MedRuRobertaLarge model: `https://huggingface.co/DmitryPogrebnoy/MedRuRobertaLarge`

### MedDistilBertBaseRuCased

The second basic model was the distilbert-base-multilingual-cased model[1]. This model is slightly inferior in efficiency to the previous model, but has a much smaller size and better performance.

This model supports multiple languages including Russian. For the task of ranking editing candidates in Russian, all other languages are superfluous, so before fine-tuning the model it was necessary to modify it so as to discard languages that are not needed.

In order to discard unnecessary languages, the approach from the research paper by Amine A. et al. Load What You Need: Smaller Versions of Multilingual BERT [53] was applied. However, this approach has been applied with a notable modification. In the original article, a large corpus of texts in this language was processed in order to isolate the necessary vocabulary of language tokens. Thus, the token dictionary included not only tokens in the target language, but also tokens of words from other languages that were contained in the corpus of texts of the target language. For example, as a result of this approach, the model, which should only support the Russian language, contained some tokens of English words, because English words are sometimes found in Russian texts.

This approach makes it possible to prepare a model that can handle almost arbitrary text in the target language with various inclusions of other languages. However, this is not required for the task of ranking editing candidates. Therefore, instead of selecting the necessary tokens with a corpus of texts in the target language, another approach was used. A set of regular expressions was used to select the tokens of the target language, as well as the tokens of digits, punctuation marks, and other special characters.

As a result, a Jupyter Notebook script[2] was developed to convert the multilingual model into a monolingual (Russian) model, which selects the necessary tokens and modifies the model thus significantly reducing it's size.

[1]distilbert-base-multilingual-cased model on Hugging Faces hub: `https://huggingface.co/distilbert-base-multilingual-cased`

[2]Notebook script for converting multilanguage distilbert-base-multilingual-cased to russian language model: `https://github.com/DmitryPogrebnoy/MedSpellChecker/blob/main/medspellchecker/ml_ranging/models/distilbert_base_russian_cased/distilbert_from_multilang_to_ru.ipynb`

As a result, the resulting model is almost two and a half times smaller. Its size was reduced from 542 Mb to 217 Mb. The converted model is published on the Hugging Face service as DmitryPogrebnoy/distilbert-base-russian-cased[1]

The Russian monolingual model was then fine-tuned. The fine-tuning parameters of the model are presented in the Table 4.4

Table 4.4 — Fine-tuning parameters of the distilbert-base-russian-cased model.

| **Parameter** | **Value** |
|---|---|
| Train epoch | 10 |
| Learning rate | 0.00005 |
| Weight decay | 0.01 |
| FP16 training | True |
| Gradient accumulation steps | 256 |
| Per device train batch size | 1 |
| Per device eval batch size | 1 |
| Gradient checkpointing | True |

As for the previous model, a special Python script[2] was written to fine-tune the model. For this model the fine-tuning parameters are chosen experimentally. It was not possible to choose optimal hyperparameters because of the duration of fine-tuning.

The change in the perplexity metric on the train and test data after each epoch is shown in Figure 4.5.

As you can see, the perplexity metric on test data has decreased from about 350 to about 120 as a result of the training.

The fine-tuned model is published on Hugging Face as DmitryPogrebnoy/MedDistilBertBaseRuCased[3]

### MedRuBertTiny2

The third basic model was the cointegrated/rubert-tiny2[4] model. This model supports the Russian language and is based on the BERT architecture.

[1]distilbert-base-russian-cased model on Hugging Face: `https://huggingface.co/DmitryPogrebnoy/distilbert-base-russian-cased`

[2]Python script for fine-tuning distilbert-base-russian-cased model: `https://github.com/DmitryPogrebnoy/MedSpellChecker/blob/main/medspellchecker/ml_ranging/models/med_distilbert_base_russian_cased/fine_tune_distilbert_base_russian_cased.py`

[3]MedDistilBertBaseRuCased model: `https://huggingface.co/DmitryPogrebnoy/MedDistilBertBaseRuCased`

[4]cointegrated/rubert-tiny2 model: `https://huggingface.co/cointegrated/rubert-tiny2`

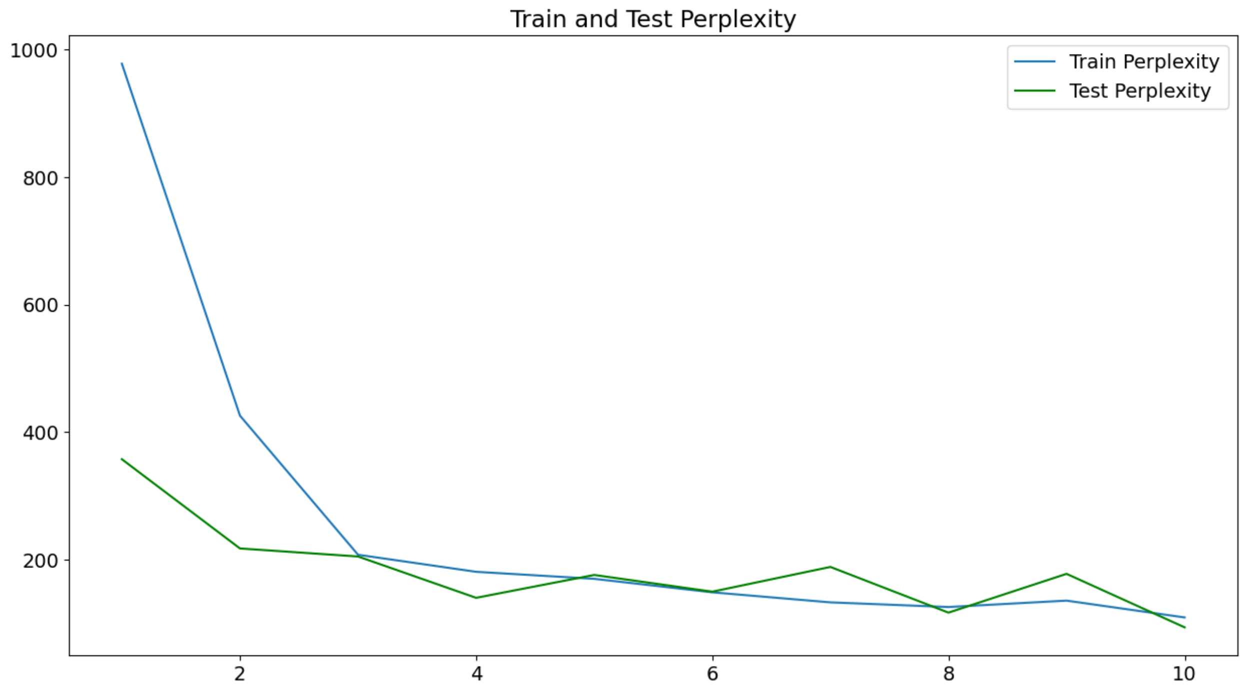


Figure 4.5 — Change perplexity metric on the train and test data after each epoch.

The main feature that distinguishes this model from the previous two is that it is much smaller in size. It is half the size of the MedDistilBertBaseRuCased model.

Model cointegrated/rubert-tiny2 was fine-tuned. The fine-tuning parameters of the model are presented in the Table 4.5

Table 4.5 — Fine-tuning parameters of the cointegrated/rubert-tiny2 model.

| **Parameter** | **Value** |
|---|---|
| Train epoch | 5 |
| Learning rate | 0.00005 |
| Weight decay | 0.01 |
| FP16 training | True |
| Gradient accumulation steps | 256 |
| Per device train batch size | 1 |
| Per device eval batch size | 1 |
| Gradient checkpointing | True |

As for the previous models, a special Python script[1] was written to fine-tune the model. For this model the fine-tuning parameters are chosen

[1]Python script for fine-tuning cointegrated/rubert-tiny2 model: `https://github.com/DmitryPogrebnoy/MedSpellChecker/blob/main/medspellchecker/ml_ranging/models/med_rubert_tiny2/fine_tune_rubert_tiny2.py`

experimentally too. It was not possible to choose optimal hyperparameters because of the duration of fine-tuning.

The change in the perplexity metric on the train and test data after each epoch is shown in Figure 4.6.

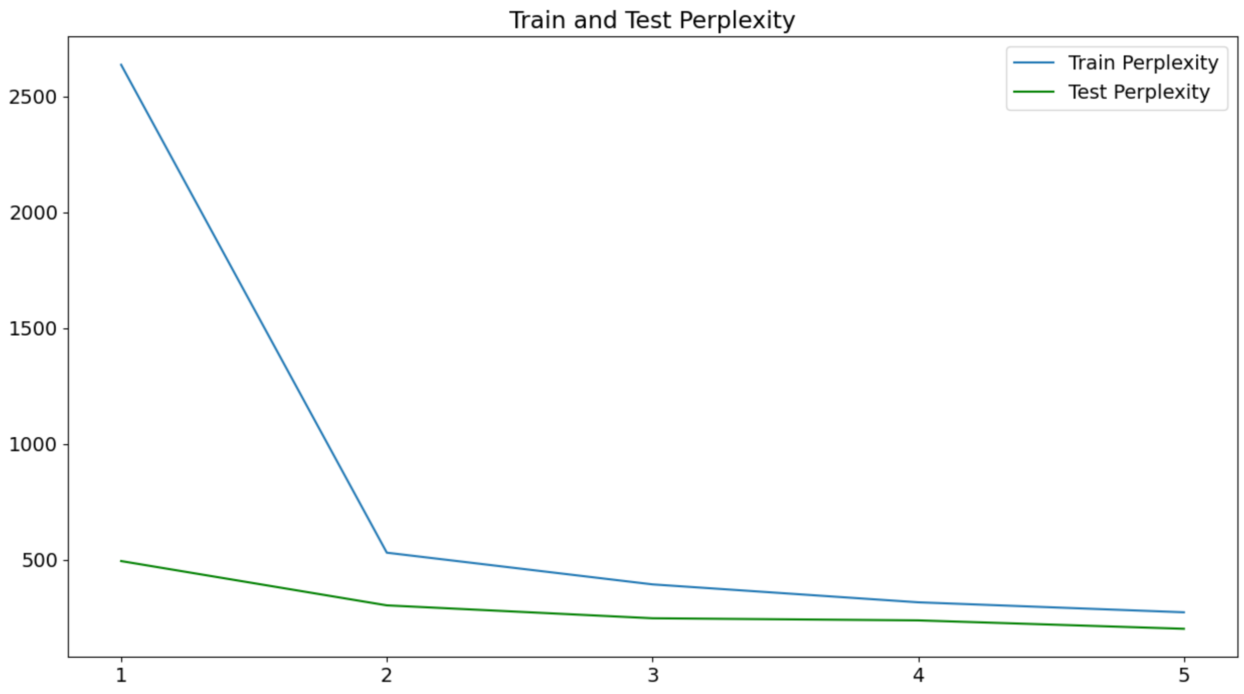


Figure 4.6 — Change perplexity metric on the train and test data after each epoch.

As you can see, the perplexity metric on test data has decreased from just over 490 to about 175 as a result of the training.

The fine-tuned model is published on Hugging Face as DmitryPogrebnoy/MedRuBertTiny2[1].

### RuBioBERT and RuBioRoBERTa

In addition to three rankers for editing candidates based on new finely tuned models, two rankers based on RuBioBERT[2] and RuBioRoBERTa[3] models are implemented. Both models were published as a result of the work of Alexander Yalunin et al. [65] on fine-tuning the Russian BERT and RoBERTa models to medical texts in Russian. Both models were fine-tuned on a smaller amount of data than the previously described models. Nevertheless, the authors

[1]MedRuBertTiny2 model: `https://huggingface.co/DmitryPogrebnoy/MedRuBertTiny2`
[2]alexyalunin/RuBioBERT model: `https://huggingface.co/alexyalunin/RuBioBERT`
[3]alexyalunin/RuBioRoBERTa model: `https://huggingface.co/alexyalunin/RuBioRoBERTa`

provided experimental results that show that these models outperformed the basic Russian BERT and RoBERTa models in almost all medical text tasks.

The rankers for these models have been implemented as an example of how one can easily extend the tool and adapt any suitable BERT model to rank candidates for editing and use in the developed tool. However, it is possible to use not only BERT models but also any arbitrary models as a ranker. It is enough to implement the necessary interface for a particular model and everything else will work right out of the box.

As a result, the Language Model component contains edit candidate rankers with three new different BERT-based models fine-tuned to rank a edit candidate and two BERT models for the medical text tasks. With several models of different sizes, it is possible to use this spellchecker tool on a variety of technical hardware, from a high-performance server with the MedRuRobertaLarge model to an ordinary computer with the MedRuBertTiny2 model.

### Post-processor

The Post-processor component is responsible for assembling the final text from the internal token representation. All text tokens are concatenated with a space, and the resulting text is returned as corrected text. Before concatenation, all corrected tokens are transformed into the form of the original word thanks to the saved word form tags in the meta information of the token internal representation.

Thus, the final text contains corrected tokens in the form of the original words, including untouched stop words, short tokens, and tokens containing numbers and non-Russian alphabet characters. The final text lacks all punctuation marks because they were erased during tokenization. Punctuation marks may contain some useful information. However, for most tasks in natural language processing, this information is not critically important, and removing punctuation marks is part of the usual preprocessing pipeline for medical text data.

## 4.5 Python package

The implementation described above has been wrapped in a Python package and uploaded to the official pip repository. The package with the new tool is called medspellchecker[1]. In addition to the source code and necessary classes, this package also contains a dictionary of correct words. Apart from the dictionary, the package does not contain any other additional datasets or models, which keeps the size of the package small.

The medspellchecker package does not contain any parts or compiled fragments of ranking models. They are all downloaded dynamically from public repositories on Hugging Faces. This way, the user can load only the required model and not load unnecessary data. However, to use the tool, at least for the first time, an internet connection is required. After the model has been loaded for the first time and has been cached, an internet connection is not necessary.

An example of the use of a package with the developed tool is shown in Listing 1.

```
 1  from medspellchecker.tool.medspellchecker \
 2      import MedSpellchecker
 3  from medspellchecker.tool.distilbert_candidate_ranker \
 4      import RuDistilBertCandidateRanker
 5
 6  candidate_ranker = RuDistilBertCandidateRanker()
 7  spellchecker = MedSpellchecker(candidate_ranker)
 8  fixed_text = spellchecker.fix_text(
 9      "У больного диагностирован инфркт"
10  )
11
12  print(fixed_text)
```

Listing 1 — An example of using a new tool to correct an error in the sentence "The patient has diagnosed with a heart attack"

The first two lines import the main class of the package. Lines 3 and 4 then import one of the three available classes to rank the edit candidates. Line

[1]medspellchecker pip package: `https://pypi.org/project/medspellchecker/`

6 creates an instance of the candidate ranking class based on the fine-tuned DistilBert model. Line 7 creates an instance of the class to correct spelling errors. On line 8, the $fix\_text$ method is called, which takes the raw text and returns the corrected text. Finally line 10 prints the corrected result, which looks like «У больного диагностирован инфаркт». In this way, the package can be used to correct spelling errors in Russian medical texts in a few lines.

## 4.6 Method of using the new tool

The effectiveness of using the developed approach and tool to correct spelling errors in medical texts in Russian depends on the method of using this tool. This tool is designed to be embedded in the pre-processing pipeline of raw medical texts for various machine learning tasks. Incorrect use of the tool can not only fail and waste computing resources, but even spoil the original medical texts. The proposed method of using the new tool in the pre-processing pipeline of medical texts for various tasks is shown in Figure 4.7.

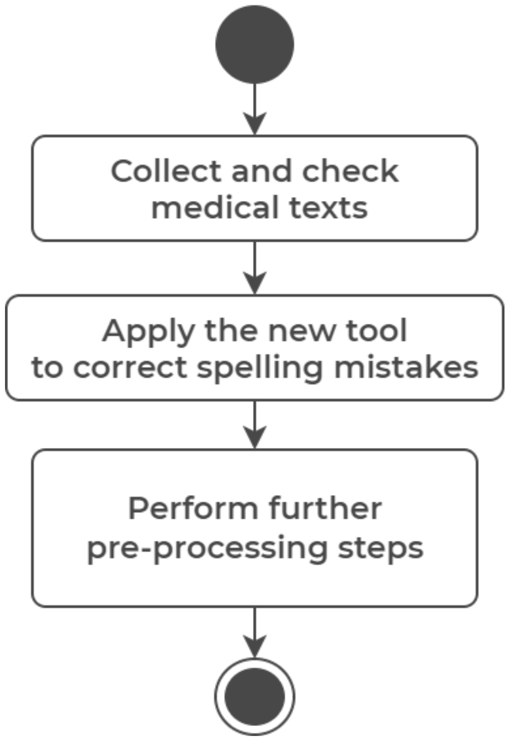

Collect and check medical texts
Apply the new tool to correct spelling mistakes
Perform further pre-processing steps

Figure 4.7 — Proposed scheme for using the developed tool for effective correction of spelling errors in medical texts in Russian.

First of all, it is necessary to collect medical data and verify it. Medical data should be presented as plain text in Russian without any markup. Preferably, the collected data should be anamnesis, patient case histories or something similar. In addition, the text data should be without missing values, as the tool is not intended to handle values of this type.

Then, after the data have collected and before any other pre-processing operations, it is suggested that the developed tool be applied. The absence of any pre-processing before applying the tool is due to the fact that any pre-processing may skew the original text. The developed tool, on the other hand, assumes that it takes the original text as it is. It should also be noted that it is necessary to feed the tool with the whole unit of text, not with words or phrases. This way, the tool will have a full context, which will allow to better use the features of language models based on BERT architecture and perform the best correction of spelling errors.

After applying the developed tool, it is enough to see a few examples of corrected text to make sure that everything is in order. If so, the rest of the pre-processing steps can be done next. If something went wrong, consider making a few experiments to see what caused the incorrect result, or not using the tool in a pipeline with this data.

# 5 Tool approbation

As part of testing, the tool was tested on a test dataset with single words with mistakes and on a test set of incorrect words with context. The test datasets are exactly the same as those used to measure the metrics of the existing general purpose tools in Chapter 2. It should be noted that the developed tool supports all types of errors described in Section 1.2. Including spelling errors related to missing or extra spaces.

The developed tool supports several independent settings. The first setting determines whether the tool will attempt to correct errors due to missing or extra spaces. This mode will be marked as "Handle space"in the metric tables. Handling errors with whitespace significantly reduces performance. This can be useful when more performance is needed or whitespace errors handling is not required.

The second setting allows to select a ranking machine learning model to select the best candidate for correcting an incorrect word. Five different size models based on the BERT architecture are supported for this tool. Three new ones are specially fine-tuned to work with medical anamneses in Russian, as well as two models derived from the Alexander Yalunin et al. article [65]. A pre-defined set of models allows selecting the best model for each case given the available technical resources. In addition, this setting allows one to use any custom model for ranking edit candidates. It requires you to write an adapter class for the custom model and that's it.

The third setting allows to select which computing device to use for computing. Two options are available, CPU and GPU. If the computer have a sufficiently powerful Nvidia CUDA GPU with enough free video memory, the spellchecker will by default perform all the calculations on the freest GPU, otherwise all the calculations will be performed on CPU. It is also possible to force calculations on CPU. This setting allows the flexibility to adapt to the technical environment and take advantage of all possibilities to achieve the best performance.

## 5.1 Single words correction test

The metrics results of the single word test for the different models and different modes of the developed tool are presented in Table 5.1. The performance tests of the tool were carried out on laptop on Ubuntu 20.04 with 24 GB of RAM, Intel Core i5-10210H CPU @ 1.60GHz * 8 and Nvidia Tesla V100.

Table 5.1 — Comparison of metrics on single word test for the new spellchecker.

| Ranking model | Handle space | Compute mode | Error precision | Lexical precision | Overall precision | Average words per second |
|---|---|---|---|---|---|---|
| MedRuRoberta | + | CPU | 0.715 | 0.991 | 0.853 | 2.1 |
| | | GPU | | | | 5.9 |
| | - | CPU | 0.842 | 0.991 | 0.917 | 2.3 |
| | | GPU | | | | 6.7 |
| MedDistilBert | + | CPU | 0.701 | 0.991 | 0.846 | 12.7 |
| | | GPU | | | | 39.7 |
| | - | CPU | 0.821 | 0.991 | 0.906 | 15.8 |
| | | GPU | | | | 43.5 |
| MedRuBertTiny2 | + | CPU | 0.681 | 0.991 | 0.836 | 24.2 |
| | | GPU | | | | 79.1 |
| | - | CPU | 0.793 | 0.991 | 0.892 | 34.9 |
| | | GPU | | | | 98.2 |
| RuBioRoBERTa | + | CPU | 0.695 | 0.991 | 0.843 | 2.2 |
| | | GPU | | | | 5.8 |
| | - | CPU | 0.768 | 0.98 | 0.874 | 2.4 |
| | | GPU | | | | 6.2 |
| RuBioBERT | + | CPU | 0.683 | 0.991 | 0.837 | 8.3 |
| | | GPU | | | | 20.1 |
| | - | CPU | 0.732 | 0.98 | 0.856 | 10.1 |
| | | GPU | | | | 24.3 |

It is worth noting that the metrics for precision without space handling are calculated without considering cases where the test example contains a space-related error.

The single word test results show that the performance of the tool depends to a large extent on the size of the ranking model. The larger the model, the longer it takes to infer and the lower the performance of the overall tool. For example, tool performance can vary by a factor of 15 depending on the ranking model used.

In addition, the tests show that the ranking model affects the precision of the tool. However, this effect is not as strong for new models as it could be and the tests show only a small decrease in accuracy with a significant reduction in model size. This is probably because the words are fed into the tool one at a time without context, so models that take context into account cannot show their full potential. In addition, the result may have been affected by the limited amount of data available to fine-tune the models. For these models, the data collected for training may not have been quite sufficient and with more data they could have shown better results. Furthermore, the results with RuBioBERT and RuBioRoBERTa models were noticeably lower than the results of the new models. This may be due to the fact that the new models were trained on more data and generalized the information better. It is interesting, however, to compare the metrics of the new tool with those of the existing tools.

Table 5.2 shows the results of the single word tests of the existing spellcheckers and the new tool.

Table 5.2 — Comparison of popular spelling correction tools and new one for correcting spelling errors without context.

| Tool name | Processor type | Error precision | Lexical precision | Overall precision | Average words per second |
|---|---|---|---|---|---|
| Aspell-python [28] | CPU | **0.86** | 0.859 | **0.859** | 283.7 |
| PyHunspell [29] | CPU | 0.812 | 0.539 | 0.675 | 9.4 |
| PyEnchant [30] | CPU | 0.829 | 0.541 | 0.685 | 20 |
| LanguageTool-python [31] | CPU | 0.762 | 0.904 | 0.833 | 25.1 |
| PySpellChecker [32] | CPU | 0.354 | 0.86 | 0.607 | 3.4 |
| SymspellPy [34] | CPU | 0.399 | 0.813 | 0.606 | **9702.8** |
| SymspellPy (compound) [34] | CPU | 0.465 | 0.512 | 0.489 | 672 |
| Jumspell [35] | CPU | 0.267 | 0.947 | 0.607 | 2552.1 |
| Spellchecker prototype [2] | CPU | 0.41 | 0.83 | 0.62 | 0.07 |
| Yandex.Speller [36] | CPU | **0.916** | 0.971 | **0.944** | 266.3 |
| RuMedSpellchecker (MedRuRobertaLarge) | CPU | 0.715 | **0.991** | 0.853 | 2.1 |
|  | GPU |  |  |  | 5.9 |
| RuMedSpellchecker (MedDistilBertBaseRuCased) | CPU | 0.701 | **0.991** | 0.846 | 12.7 |
|  | GPU |  |  |  | 39.7 |
| RuMedSpellchecker (MedRuBertTiny2) | CPU | 0.681 | **0.991** | 0.836 | 24.2 |
|  | GPU |  |  |  | 79.1 |
| RuMedSpellchecker (RuBioRoBERTa) | CPU | 0.695 | **0.991** | 0.843 | 2.2 |
|  | GPU |  |  |  | 5.8 |
| RuMedSpellchecker (RuBioBERT) | CPU | 0.683 | **0.991** | 0.837 | 8.3 |
|  | GPU |  |  |  | 20.1 |

It can be seen that the new tool has fairly average results on the single word test. For example, the new tool outperforms LanguageTool-python in lexical precision and performance, but loses to the Aspell-Python tool in error precision and overall precision. It is likely that to fully outperform Aspell-python, an incredible amount of text must also be processed. This requires a lot of processing capacity and time, so the new tool has an average result for the new tool.

MedSpellchecker is also inferior in precision and performance to Yandex.Speller, which is not open source and has many legal restrictions for full use. Therefore, the comparison with Yandex.Speller is not completely correct because it is not an open-source tool. Nevertheless, its results are higher, which indicates that there is still room for improvements.

## 5.2 Words with context correction test

Correcting single words is a bit of an artificial situation. Usually there is not just one word, but all or at least part of the text at once. Fine-tuned language models on the BERT architecture allow one to consider the context around the leading word and perform a better ranking of correction candidates, which improves the overall precision of the correction.

The presence of context is a natural prerequisite, as the spellchecker will work with whole medical texts. In order to assess the quality of the tool under such conditions, a test with context was conducted in the same way as in Chapter 2.

The results of the test with context are presented in Table 5.3.

As in the previous test, it is worth noting that the precision metrics without space handling are calculated without taking into account cases where the test example contains a space-related error.

The results of the test show that in the context test the precision of errors increased significantly, while the lexical precision increased slightly. This effect is explained by the fact that the context of the incorrect word is taken into account by the model to rank the candidates. This makes the ranking more precise and in frequent cases gives the correct result.

As expected, the larger the model, the slightly better it performs. The largest new model MedRuRoberta showed the best result. RuBioBERT and RuBioRoBERTa models showed slightly lower metrics than the new models.

Table 5.3 — Comparison of metrics on test with context for the new spellchecker.

| Ranking model | Handle space | Compute mode | Error precision | Lexical precision | Overall precision | Average words per second |
|---|---|---|---|---|---|---|
| MedRuRoberta | + | CPU | 0.792 | 0.989 | 0.888 | 8.9 |
| | | GPU | | | | 29.1 |
| | - | CPU | 0.873 | 0.99 | 0.932 | 12.5 |
| | | GPU | | | | 31.6 |
| MedDistilBert | + | CPU | 0.765 | 0.99 | 0.878 | 45.5 |
| | | GPU | | | | 153.8 |
| | - | CPU | 0.85 | 0.99 | 0.92 | 62.2 |
| | | GPU | | | | 153.2 |
| MedRuBertTiny2 | + | CPU | 0.742 | 0.989 | 0.866 | 127.3 |
| | | GPU | | | | 356.2 |
| | - | CPU | 0.843 | 0.989 | 0.916 | 182.1 |
| | | GPU | | | | 403.2 |
| RuBioRoBERTa | + | CPU | 0.738 | 0.987 | 0.863 | 9.1 |
| | | GPU | | | | 31.3 |
| | - | CPU | 0.821 | 0.989 | 0.905 | 13.5 |
| | | GPU | | | | 42.6 |
| RuBioBERT | + | CPU | 0.715 | 0.988 | 0.852 | 30.7 |
| | | GPU | | | | 95.6 |
| | - | CPU | 0.843 | 0.989 | 0.916 | 37.5 |
| | | GPU | | | | 123.5 |

But it is much more interesting to compare the results of the new tool with results of other existing tools. This tool was created for this very purpose. The results of the test are shown in Table 5.4.

Test with context results show that the new tool outperforms all of the open source tools reviewed and falls far behind Yandex Speller in terms of precision and performance. The performance of the new tool is average compared to other tools. It is better than about half of the open source tools reviewed. The tool's performance with two of the three new models satisfies the required tool performance of 150 words per second, which was defined in the Section 2. However, with the largest model, with which the tool achieves the best metrics, the required performance is not achieved. This shows that there is still room to improve the algorithm and optimize the implementation.

Figure 5.1 clearly shows the differences in the overall metric of tool precision and performance on the test with context.

It may be noted that, depending on the ranking model, the precision and performance of the tool varies greatly. It is interesting to note that the overall precision of MedSpellchecker depends on the size of the ranking model. The

Table 5.4 — Comparison of popular spelling correction tools and new one for correcting spelling errors with context.

| Tool name | Processor type | Error precision | Lexical precision | Overall precision | Average words per second |
|---|---|---|---|---|---|
| Aspell-python [28] | CPU | 0.739 | 0.93 | 0.835 | 357.3 |
| PyHunspell [29] | CPU | 0.706 | 0.719 | 0.713 | 11.8 |
| PyEnchant [30] | CPU | 0.721 | 0.719 | 0.72 | 24.3 |
| LanguageTool-python [31] | CPU | 0.727 | 0.942 | 0.835 | 43.6 |
| PySpellChecker [32] | CPU | 0.304 | 0.868 | 0.586 | 6.7 |
| SymspellPy [33] | CPU | 0.37 | 0.913 | 0.642 | 26060.2 |
| SymspellPy (compound) [34] | CPU | 0.483 | 0.804 | 0.643 | 1604.2 |
| Jumspell [35] | CPU | 0.307 | 0.969 | 0.638 | 4322.3 |
| Spellchecker prototype [2] | CPU | 0.25 | 0.79 | 0.52 | 0.09 |
| Yandex.Speller [36] | CPU | 0.82 | 0.987 | 0.904 | 1308.8 |
| RuMedSpellchecker (MedRuRobertaLarge) | CPU | **0.792** | 0.984 | **0.888** | 8.9 |
| | GPU | | | | 29.1 |
| RuMedSpellchecker (MedDistilBertBaseRuCased) | CPU | 0.765 | **0.99** | 0.878 | 45.5 |
| | GPU | | | | 153.8 |
| RuMedSpellchecker (MedRuBertTiny2) | CPU | 0.742 | 0.987 | 0.865 | 127.3 |
| | GPU | | | | 356.2 |
| RuMedSpellchecker (RuBioRoBERTa) | CPU | 0.738 | 0.987 | 0.863 | 9.1 |
| | GPU | | | | 31.3 |
| RuMedSpellchecker (RuBioBERT) | CPU | 0.715 | 0.988 | 0.852 | 30.7 |
| | GPU | | | | 95.6 |

larger the model, the higher the overall precision. The same can be said about the performance of the tool with different models. The larger the model, the lower the overall performance of the tool. So there is a trade-off between overall precision and performance. If one need more performance, a smaller model is better. Otherwise, larger models can be used. In addition, it is interesting to see the results of metrics with even smaller models. Perhaps they will show not much lower accuracy, but much higher performance. But this is not part of this paper and will be explored in the future.

The closest existing tools are Aspell-python and LanguageTool-python. Aspell-python has slightly better performance, but worse precision. LanguageTool-python is inferior in both metrics. Only the proprietary Yandex.Speller tool is superior in precision and performance. This means that there is definitely still room for improvement in the correction algorithm and other optimizations of the new spellchecker.

Figure 5.2 clearly shows the differences in the lexical and error precision metrics of tools on the test with context.

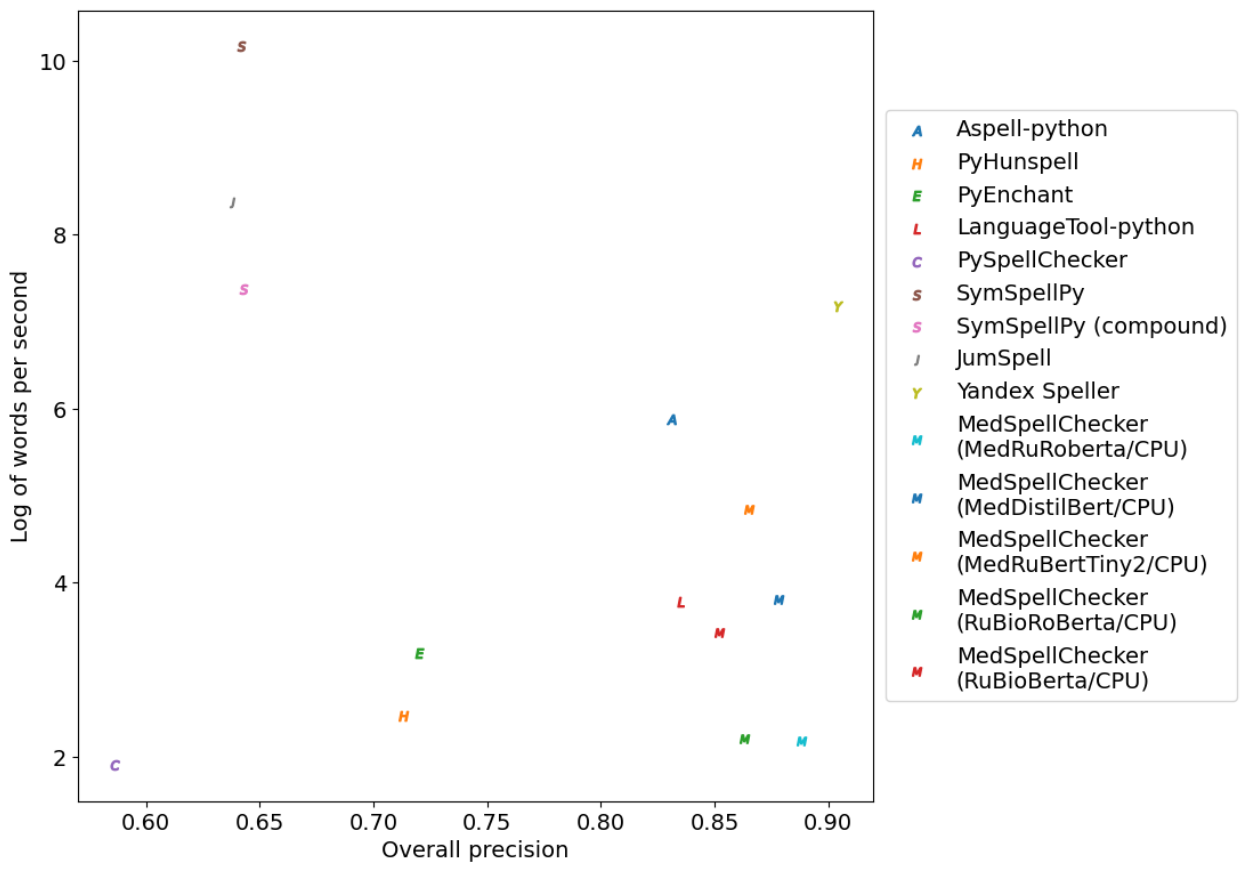


Figure 5.1 — Performance logarithm and overall precision metrics for existing spellcheckers and the new one.

The new tool shows good lexical and error precision and no drop in either of these metrics. The Aspell-python tool wins slightly in error precision, but loses noticeably in lexical precision. The LanguageTool-python tool slightly loses in error precision and significantly loses in lexical precision. JumSpell also has high lexical precision, but one of the lowest error precision. As in the previous graphs and tables, Yandex.Speller clearly stands out, outperforming all opponents in both metrics.

To summarise, the new tool does a good job of correcting incorrect words with context in Russian in medical texts. Among the existing open source tools reviewed, none of them outperforms the new tool completely. Only a few tools outperform the new tool, either in terms of performance or lexical or error precision. The existing closed-source tool Yandex.Speller completely outperforms the new tool in all metrics. This means that there is still room for improvement, research and experimentation.

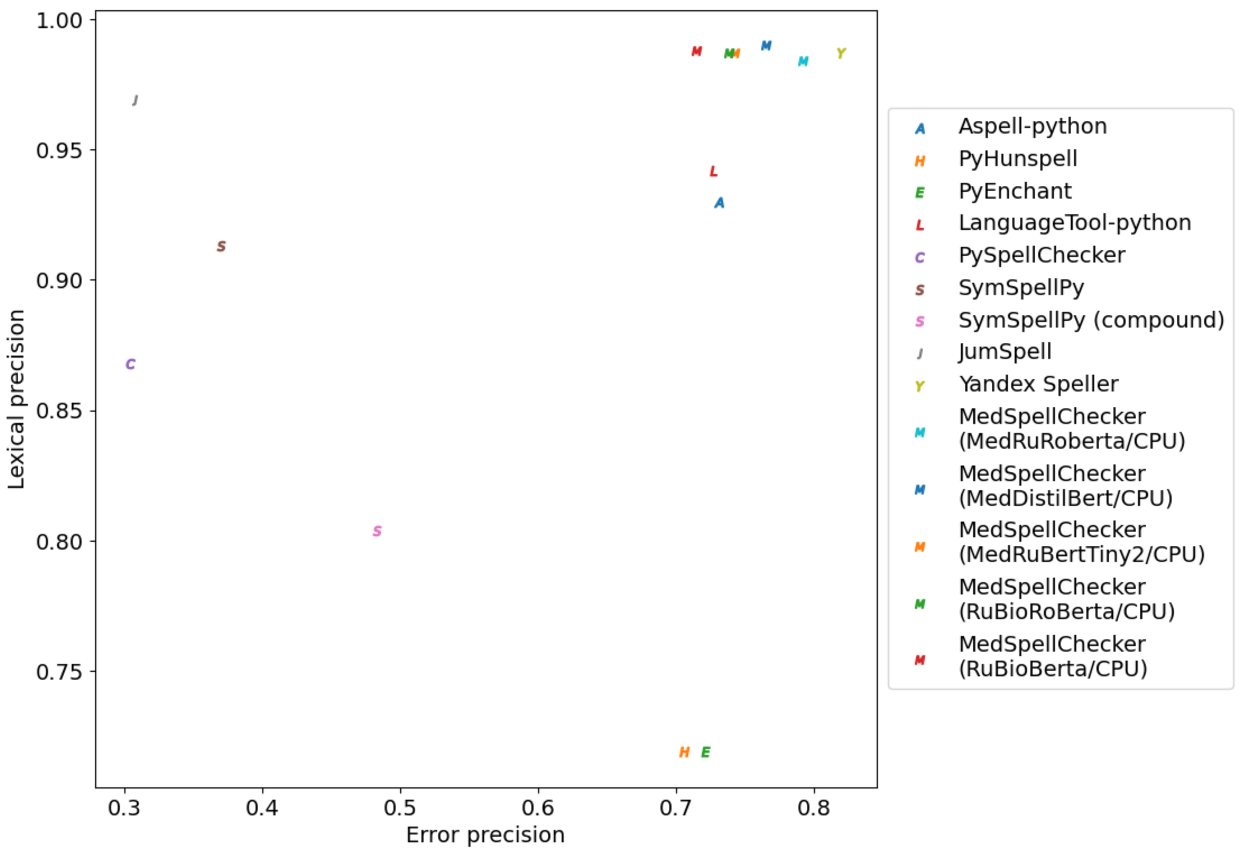


Figure 5.2 — Lexical and error precision metrics for existing spellcheckers and the new one.

## 5.3 Real anamnesis correction test

The new tool showed strong results in the single word test and the context word test, which indicates that the new tool was effective. However, these tests were a bit artificial. Errors in the test words were added automatically in both tests. In addition, the number of correct and incorrect words clearly did not correlate with the real medical history. Therefore, in order to be sure that the new tool is really effective and applicable for automatic correction of medical texts, it is necessary to test its work on real data.

In order to test the new and existing instruments, 100 real anamnesis were randomly taken from the test part. Then, the selected anamnesis were automatically corrected with each of the instruments under review. The results were then compared with the original text and two metrics were manually calculated for each instrument. The first metric is the number of correctly corrected errors. The more, the better. The second metric is the number of corrected words that were corrected. The less, the better. For clarity, the ratio of correct and unnecessary corrections is also counted. This metric the more the better. Thus, using these three metrics, it is possible to evaluate how much cleaner and more correct the text became after the application of a particular tool.

The test results for the new and existing tools are shown in Table 5.5. Dashes in the table mean that after applying the tool the original text changed so much that it became impossible to calculate the necessary metrics relative to the original text.

Table 5.5 — Comparison of existing spelling correction tools and a new tool for correcting spelling errors in real anamnesis.

| Tool Name | Correct fixes | Unnecessary fixes | Fixes Ratio |
|---|---|---|---|
| Aspell-python [28] | 20 | 171 | 0.105 |
| LanguageTool-python [31] | **21** | 135 | 0.135 |
| PySpellChecker [32] | - | - | - |
| SymspellPy [33] | - | - | - |
| SymspellPy (compound) [34] | - | - | - |
| Jumspell [35] | - | - | - |
| Spellchecker prototype [2] | - | - | - |
| Yandex.Speller [36] | **21** | **0** | **1.0** |
| RuMedSpellchecker (MedRuRobertaLarge) | 19 | 6 | **0.76** |
| RuMedSpellchecker (MedDistilBertBaseRuCased) | 18 | 9 | 0.667 |
| RuMedSpellchecker (MedRuBertTiny2) | 17 | 11 | 0.607 |
| RuMedSpellchecker (RuBioRoBERTa) | 15 | 13 | 0.536 |
| RuMedSpellchecker (RuBioBERT) | 13 | 16 | 0.448 |

For most of the existing tools, it was impossible to calculate the necessary metrics. For Aspell-python and LanguageTool-python, the unnecessary fixes metric was very high, which completely overwhelmed the good result in the correct fixes metric. Yandex Speller tool showed outstanding result again and fixed only incorrect words without changing any correct words. Thus its metric of the ratio of two errors is one and it is the highest among all analyzed tools.

The new tool showed much higher metrics than other open source tools. As in previous tests, the larger the model used, the better the fix result. However, with RuBioBERT and RuBioRoBERTa models the tool showed mediocre results. In both cases the ratio metric is very close to 0.5 or less, which suggests that the text became more incorrect than it was after using the tool based on these models. For the new models, the ratio metric is greater than 0.6, indicating that the text after applying the tool contains fewer errors than the original. It means that the tool has a positive effect on the purity of the text and is effective for correcting spelling errors in medical texts in Russian.

It should be noted that the limited data set used to train models may not be representative of all possible types of medical histories and may be skewed in some way. This can lead to a biased model that may inaccurately rank edit candidates and be less effective at correcting spelling errors in other types of medical texts.

In addition, models may be biased toward non-medical texts, such as social media posts or news articles, because they differ in content and vocabulary from medical texts and were not included in the training sample. Thus, the application of the developed tool to non-medical text types can lead to inaccurate text corrections and unpredictable consequences.

To summarize, the test results show that the developed tool outperforms existing open source tools in terms of accuracy and performance. The new tool showed strong results in all the tests, which proves its efficiency in correcting spelling errors in Russian medical texts. Nevertheless, some proprietary tools still outperform the new tool, indicating a great potential for improvement of the tool and the need for further experiments.

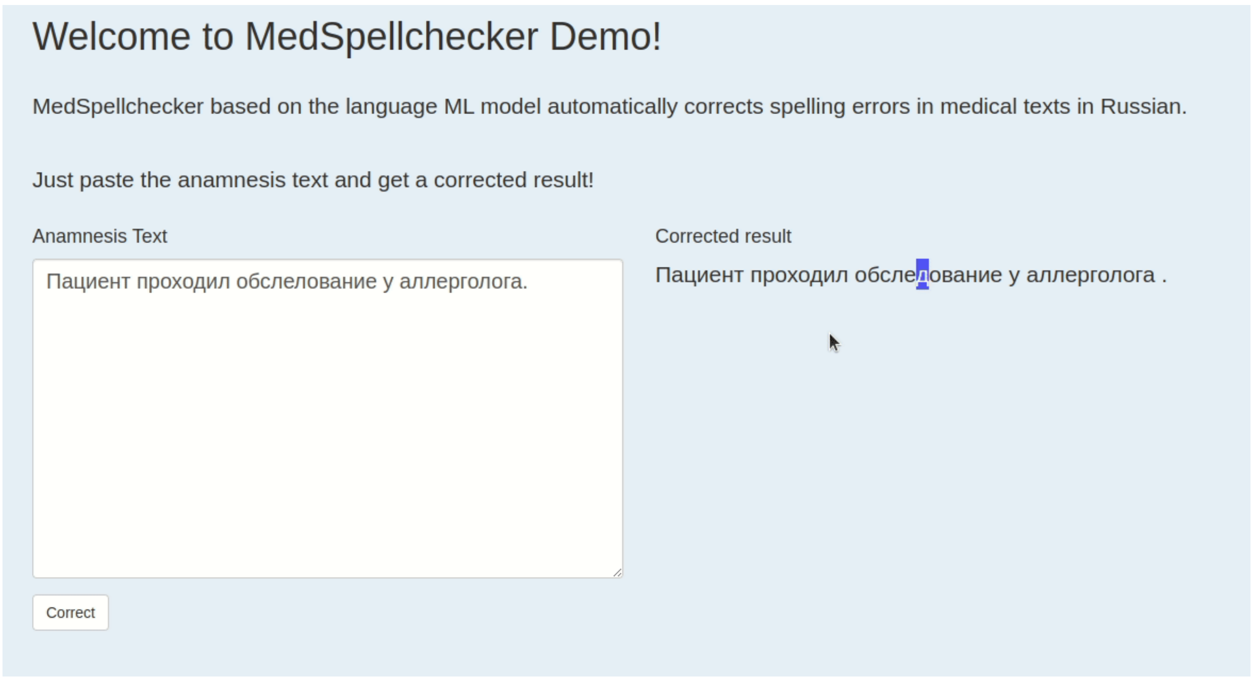
## 5.4 Tool Demo

A demo website has also developed to demonstrate how the tool works. The demo is written in Python using the Flask framework. The Flask-WTF[1] library is used to build the UI forms. The Flask-Bootstrap[2] library is used to style UI form components. It also uses JavaScript and AJAX [66] technology to dynamically load the corrected text without having to reload the page. In

[1]Flask-WTF pip package: `https://pypi.org/project/Flask-WTF/`

[2]Flask-Bootstrap pip package: `https://pypi.org/project/Flask-Bootstrap/`

Figure 5.3 shows the demo page with the correction of the wrong word in the sentence «Пациент проходил обслеловование у аллерголога.».


Welcome to MedSpellchecker Demo!

MedSpellchecker based on the language ML model automatically corrects spelling errors in medical texts in Russian.

Just paste the anamnesis text and get a corrected result!

Anamnesis Text

Пациент проходил обслелование у аллерголога.

Corrected result

Пациент проходил обследование у аллерголога .

Figure 5.3 — The demo website for MedSpellchecker tool.

In this example, the sentence contains the misspelled word «обслелование». To the right of the example sentence is its corrected version with the word «обследование» corrected.

This demo website can be run by yourself using the instructions in the README.md[1] in the project repository on the Github.

[1]Instruction how to run demo website: `https://github.com/DmitryPogrebnoy/MedSpellChecker#demo/`

# Conclusion

To summarize, the following results were obtained in this paper.

1) An overview of the subject area is made. Types of mistakes in texts are described and a list of types of spelling errors to correct in this paper is highlighted. An overview of the different approaches to edit distances is also made. In addition, a general approach to automatic correction of errors in texts is described, and a generalized architecture of such tools is presented. An overview of relevant articles focused on correcting English, Russian, and separately medical texts is also made.

2) The review and testing of existing open-source tools, which have the ability to correct spelling errors in Russian, are made. According to the test results, none of the tools has both sufficient precision and performance metrics. Also based on the available anamnesis the required performance of 275 words per second was specified.

3) A new algorithm for correcting spelling errors in Russian is developed. To generate candidates for correction, the algorithm uses the Damerau-Levenstein distance, and the SumDel index is used to optimize the distance calculation. A fine-tuned model based on the BERT architecture is used to rank candidates.

4) A tool for automatic correction of medical texts has been developed that implements a new algorithm. Three different models on the BERT architecture were fine-tuned to rank edit candidates in medical texts. Two existing BERT models for Russian medical texts were adapted for use in the developed tool. The resulting tool was packaged in a pip package and published.

5) The developed tool and the existing tools were tested on single words, on words with context and on real medical anamneses. In the test with single words, the new tool performed slightly above average. In the test with words with context, the tool showed metrics slightly or significantly higher than other existing tools. At the same time, the performance of the tool reached the desired level only with two of the five language models. In the test with real anamneses the new tool showed its efficiency and was better in comparison with existing open source tools. In addition, a demo website was implemented to demonstrate the new tool.

This work was presented at the $XI$ Conference of Young Scientists [67], the theses of the work were published [68] in an online collection.

In addition, an article «RuMedSpellchecker: correcting spelling errors for natural Russian language in electronic health records using machine learning techniques» describing this work was accepted for the Computational Health track of the International Conference on Computational Science 2023.

## Acknowledgments

I would like to thank Anastasia Funker for her help with obtaining medical data and reproducing the results of previous work.

And I also want to thank my supervisor, Sergey Kovalchuk, who efficiently and diligently assists his students in writing research papers, as well as for his help in gaining access to the GPUs cluster for testing the developed tool.

## References

1. *Toutanova, Kristina.* Pronunciation Modeling for Improved Spelling Correction / Kristina Toutanova, Robert C. Moore // Proceedings of the 40th Annual Meeting on Association for Computational Linguistics. — ACL '02. — USA: Association for Computational Linguistics, 2002. — P. 144–151. https://doi.org/10.3115/1073083.1073109.

2. *Balabaeva, Ksenia.* Automated Spelling Correction for Clinical Text Mining in Russian / Ksenia Balabaeva, Anastasia A. Funkner, Sergey V. Kovalchuk // *Studies in health technology and informatics.* — 2020. — Vol. 270. — Pp. 43–47.

3. *Mitton, R.* English Spelling and the Computer / R. Mitton. Real Language Series. — Longman, 1996. https://books.google.cd/books?id=L1ANAQAAMAAJ.

4. *Yannakoudakis, E.J.* The rules of spelling errors / E.J. Yannakoudakis, D. Fawthrop // *Information Processing & Management.* — 1983. — Vol. 19, no. 2. — Pp. 87–99. https://www.sciencedirect.com/science/article/pii/0306457383900456.

5. *Kukich, Karen.* Techniques for Automatically Correcting Words in Text / Karen Kukich // *ACM Comput. Surv.* — 1992. — 12. — Vol. 24, no. 4. — Pp. 377–439. http://doi.acm.org/10.1145/146370.146380.

6. *Pirinen, Tommi A.* State-of-the-Art in Weighted Finite-State Spell-Checking / Tommi A. Pirinen, Krister Lindén // Computational Linguistics and Intelligent Text Processing / Ed. by Alexander Gelbukh. — Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. — Pp. 519–532.

7. *Levenshtein, V. I.* Binary Codes Capable of Correcting Deletions, Insertions and Reversals / V. I. Levenshtein // *Soviet Physics Doklady.* — 1966. — 02. — Vol. 10. — P. 707.

8. *Damerau, Fred J.* A Technique for Computer Detection and Correction of Spelling Errors / Fred J. Damerau // *Commun. ACM.* — 1964. — mar. — Vol. 7, no. 3. — P. 171–176. https://doi.org/10.1145/363958.363994.

9. *Hirschberg, D.* A linear space algorithm for com-puting longest common subsequences / D. Hirschberg // *Communications of The ACM -*

*CACM.* — 1975. — 01.

10. *Hamming, Richard.* Coding and Information Theory / Richard Hamming. — 1980. — 01.

11. *Winkler, William.* String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage / William Winkler // *Proceedings of the Section on Survey Research Methods.* — 1990. — 01.

12. *Mikolov, Tomas.* Efficient Estimation of Word Representations in Vector Space. — 2013.

13. *Bojanowski, Piotr.* Enriching Word Vectors with Subword Information. — 2017.

14. *Joulin, Armand.* Bag of Tricks for Efficient Text Classification. — 2016.

15. *Peters, Matthew E.* Deep contextualized word representations. — 2018.

16. *Pennington, Jeffrey.* Glove: Global Vectors for Word Representation. — 2014. — 01.

17. *Devlin, Jacob.* BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. — 2019.

18. *Hládek, Daniel.* Survey of Automatic Spelling Correction / Daniel Hládek, Ján Staš, Matúš Pleva // *Electronics.* — 2020. — Vol. 9, no. 10. `https://www.mdpi.com/2079-9292/9/10/1670`.

19. *Jayanthi, Sai Muralidhar.* NeuSpell: A Neural Spelling Correction Toolkit. — 2020.

20. *Hu, Yifei.* Misspelling Correction with Pre-trained Contextual Language Model. — 2021.

21. *A., Sorokin A.* SpellRuEval: the First Competition on Automatic Spelling Correction for Russian. — 2016. — 06.

22. *Sorokin, Alexey.* Automatic spelling correction for Russian social media texts. — 2016. — 06.

23. *Rozovskaya, Alla.* Spelling Correction for Russian: A Comparative Study of Datasets and Methods / Alla Rozovskaya // Proceedings of the International Conference on Recent Advances in Natural Language Pro-

cessing (RANLP 2021). — Held Online: INCOMA Ltd., 2021. — 09. — Pp. 1206–1216. https://aclanthology.org/2021.ranlp-main.136.

24. *Sorokin, Alexey.* Spelling Correction for Morphologically Rich Language: a Case Study of Russian. — 2017. — 01.

25. Automated misspelling detection and correction in clinical free-text records / Kenneth H. Lai, Maxim Topaz, Foster R. Goss, Li Zhou // *Journal of Biomedical Informatics.* — 2015. — Vol. 55. — Pp. 188–195. https://www.sciencedirect.com/science/article/pii/S1532046415000751.

26. Official website of Aspell tool. — 2000. http://aspell.net/.

27. *Fivez, Pieter.* Unsupervised Context-Sensitive Spelling Correction of Clinical Free-Text with Word and Character N-Gram Embeddings / Pieter Fivez, Simon Šuster, Walter Daelemans // BioNLP 2017. — Vancouver, Canada,: Association for Computational Linguistics, 2017. — 08. — Pp. 143–148. https://aclanthology.org/W17-2317.

28. GitHub repository of Aspell-python — Python wrapper of Aspell tool. — 2021. https://github.com/WojciechMula/aspell-python.

29. GitHub repository of PyHunspell — Python wrapper of Hunspell tool. — 2014. https://github.com/blatinier/pyhunspell.

30. Official website of PyEnchant — Python wrapper of Enchant tool. — 2021. http://pyenchant.github.io/pyenchant/install.html.

31. GitHub repository of language-tool-python — Python wrapper of LanguageTool. — 2021. https://github.com/jxmorris12/language_tool_python.

32. GitHub repository of PySpellchecker. — 2021. https://github.com/barrust/pyspellchecker.

33. GitHub repository of SymSpell tool. — 2018. https://github.com/wolfgarbe/SymSpell.

34. GitHub repository of SymSpellPy — Python wrapper of SymSpell tool. — 2021. https://github.com/mammothb/symspellpy.

35. GitHub repository of JumSpell tool. — 2018. https://github.com/bakwc/JamSpell.

36. Official website of Yandex.Speller. — 2021. https://yandex.ru/dev/speller/.

37. GitHub repository of Hunspell tool. — 2014. `https://github.com/hunspell/hunspell`.

38. Official website of Enchant tool. — 2021. `https://github.com/AbiWord/enchant`.

39. GitHub repository of Grazie plugin. — 2021. `https://github.com/JetBrains/intellij-community/tree/master/plugins/grazie`.

40. Official website of Intellij IDEA. — 2021. `https://www.jetbrains.com/idea/`.

41. Peter Norvig's article about spelling corrector. — 2007. `https://norvig.com/spell-correct.html`.

42. Official website of Jumspell. — 2021. `https://jamspell.com/`.

43. Official website of Webiomed. — 2021. `https://webiomed.ai/`.

44. GitHub repository of Sberbank AI Lab. — 2021. `https://github.com/sberbank-ai-lab`.

45. Official website of Quantori. — 2023. `https://www.quantori.com/`.

46. Attention is All you Need / Ashish Vaswani, Noam Shazeer, Niki Parmar et al. // Advances in Neural Information Processing Systems / Ed. by I. Guyon, U. Von Luxburg, S. Bengio et al. — Vol. 30. — Curran Associates, Inc., 2017. `https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf`.

47. *Radford, Alec.* Improving Language Understanding by Generative Pre-Training / Alec Radford, Karthik Narasimhan. — 2018.

48. *Taylor, Wilson L.* "Cloze Procedure": A New Tool for Measuring Readability / Wilson L. Taylor // *Journalism Quarterly.* — 1953. — Vol. 30, no. 4. — Pp. 415–433. `https://doi.org/10.1177/107769905303000401`.

49. *Liu, Yinhan.* RoBERTa: A Robustly Optimized BERT Pretraining Approach. — 2019. `https://arxiv.org/abs/1907.11692`.

50. *Clark, Kevin.* ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators. — 2020. `https://arxiv.org/abs/2003.10555`.

51. *Yang, Zhilin.* XLNet: Generalized Autoregressive Pretraining for Language Understanding. — 2019. `https://arxiv.org/abs/1906.08237`.

52. *Sanh, Victor.* DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. — 2019. https://arxiv.org/abs/1910.01108.

53. *Abdaoui, Amine.* Load What You Need: Smaller Versions of Multilingual BERT / Amine Abdaoui, Camille Pradel, Grégoire Sigel. — 2020. https://arxiv.org/abs/2010.05609.

54. *Jiao, Xiaoqi.* TinyBERT: Distilling BERT for Natural Language Understanding. — 2020.

55. *Shivade, Pavel Blinov; Aleksandr Nesterov; Galina Zubkova; Arina Reshetnikova; Vladimir Kokh; Chaitanya.* RuMedNLI: A Russian Natural Language Inference Dataset For The Clinical Domain. — 2022.

56. *Romanov, Alexey.* Lessons from Natural Language Inference in the Clinical Domain / Alexey Romanov, Chaitanya Shivade. http://arxiv.org/abs/1808.06752.

57. *A., Starovoytova Elena.* RuMedPrimeData. — 2021. https://doi.org/10.5281/zenodo.5765873.

58. GitHub repository of pyxDamerauLevenshtein package. — 2022. https://github.com/lanl/pyxDamerauLevenshtein.

59. GitHub repository of fastDamerauLevenshtein package. — 2022. https://github.com/robertgr991/fastDamerauLevenshtein.

60. GitHub repository of jellyfish package. — 2022. https://github.com/jamesturk/jellyfish.

61. GitHub repository of textdistance package. — 2022. https://github.com/life4/textdistance.

62. GitHub repository of editdistpy package. — 2022. https://github.com/mammothb/editdistpy.

63. GitHub repository of StringDist package. — 2022. https://github.com/obulkin/string-dist.

64. GitHub repository of editdistance-s package. — 2022. https://github.com/asottile-archive/editdistance-s.

65. *Yalunin, Alexander.* RuBioRoBERTa: a pre-trained biomedical language model for Russian language biomedical text mining. — 2022.

66. *Garrett, Jesse James.* Ajax: A new approach to web applications / Jesse James Garrett et al. — 2005.

67. XI Conference of Young Scientists. — 2022. https://kmu.itmo.ru/.

68. Theses 'Automatic spelling correction method for analyzing clinical text in Russian' on XI Conference of Young Scientists. — 2022. https://kmu.itmo.ru/digests/article/8367.