Improving GEC based on combining algorithmic approach and sequence-to-edit model

Yevgeniy Bodyanskiy[†], Maksym Kharchenko[*,†] and Nataliya Ryabova[†]

Kharkiv National University of Radio Electronics, Nauky ave 14, Kharkiv, 61166, Ukraine

Abstract

Nowadays, deep learning systems are replacing traditional algorithmic approaches in grammatical error correction (GEC) task. Still, these systems have problems such as requiring much more computational resources, insufficient robustness and unpredictable behavior. Traditional sequence-to-sequence approach has a serious limitation in the GEC task due to the overgeneration problem. To overcome this problem, a sequence-to-edits approach is widely used, which is described in this research. To make the sequence-to-edits approach more controllable and predictable, the architecture that allows combining traditional algorithmic approaches, like dictionary-based spell-check systems and a sequence-to-edits model based on transformers architecture is described in this research. The result of such combination is better quality of the result system that can be controlled.

Keywords

Natural Language Processing, GEC, Transformers Architecture, ModernBERT

1. Introduction

Text communication takes large part in the world of information. However, errors in writing may disrupt the meaning of the text. That is why today there is still demand for grammatical error correction systems. Over the past years such systems have improved significantly. The first solutions were based on rules and dictionaries, then neural networks approaches were used that significantly improved the performance of such systems. RNN and LSTM architectures greatly increased the quality of such solutions, then breakthroughs like transformers architecture allowed to achieve even better results [1, 2].

Although traditional approaches like sequence-to-sequence models [3] can handle the task well, they have certain limitations. One of the limitations is the performance of such systems. To correct an input text, the model needs to regenerate the whole sequence. However, unlike in language translation, in this domain many inputs shouldn't be changed at all, and others need only minor changes, which results in overgeneration problem. To resolve the issue sequence-to-edits approaches are used. These are the models that instead of regenerating the whole sequence from scratch generate only edits that need to be applied to the input sequence. Another issue with neural networks approaches is that the same problem can be fixed in one context but skipped in another.

Positions of certain errors could be obtained using algorithmic approaches. For example, misspellings could be determined using dictionary-based approach, if word is not present in dictionary, then it is misspelled and should be corrected. This research is aimed at combining algorithmic approaches with sequence-to-edits model. The assumption is that such a hybrid approach could strengthen the model's robustness and make it more predictable.

CLW-2025: Computational Linguistics Workshop at 9th International Conference on Computational Linguistics and Intelligent Systems (CoLInS-2025), May 15–16, 2025, Kharkiv, Ukraine

[*] Corresponding author.

[†] These authors contributed equally.

yevgeniy.bodyanskiy@nure.ua (Y. Bodyanskiy); maksym.kharchenko1@nure.ua (M. Kharchenko); nataliya.ryabova@nure.ua (N. Ryabova)

0000-0001-5418-2143 (Y. Bodyanskiy); 0009-0006-0706-9424 (M. Kharchenko); 0000-0002-3608-6163 (N. Ryabova)

2. Related papers

One of the main drawbacks of sequence-to-sequence solutions is the inference time and computational resources usage on regenerating the entire input sequence. Unlike in other NLP tasks like language translation, in GEC most of input's part shouldn't be changed at all. To overcome this problem, Aggressive Decoding [4] was introduced, it is aimed to reduce the number of autoregressive steps preserving strengths of sequence-to-sequence approach. However, this approach requires encoder and decoder parts of the model to have shared tokenizer vocabulary. Shared tokenizer vocabulary doesn't allow to obtain benefits from the reduction of decoder's tokenizer vocabulary, which is believed can be much smaller than encoder's vocabulary [5]. Smaller decoder vocabulary can benefit in faster inference because it reduces the softmax layer complexity, also it helps model to better generalize on smaller amounts of data.

Another common approach in GEC task is sequence-to-edits models. The main goal of these approaches is less computational resources usage, preserving the quality of error correction. There is a wide variety of different architectures with this approach. One of the solutions is GECToR [6], in its core there is pretrained encoder model, like BERT, RoBERTa, XLNet or others. This approach is using token level corrections, which are obtained without using traditional transformer decoder layer. To obtain the transformations first, transformation vocabulary is created with the most common transformations used to correct input sequence, which is DELETE, APPEND, REPLACE combined with the text, token should be replaced with, or which should be appended, also token can be kept without any transformations. Projection layer is used to convert the encoder's hidden state to the transformations. Although not all corrections could be done in one launch, after applying the corrections process repeats, until there are no new corrections.

Another approach of obtaining corrections to an input sequence is using pointer networks, which is used in RedPenNet [5] model. Here, the transformer decoder is used to generate corrections. Each correction consists of a span, where correction should be done and tokens from decoder's tokenizer vocabulary. On each autoregressive step, the decoder generates a token, which is obtained using projection layer on decoder outputs and a span, which is obtained by using projection layer on decoder attention. The assumption is, that attention mechanism points on the most valuable information in input sequence at the time of generating current token, so the position of the correction can be obtained. Each correction consists of start and end positions of the correction, and at least two tokens, one of which is a separator. If any tokens should be deleted, a special deletion token is generated. This approach provides the ability to have a separate vocabulary for decoder, also it allows decoder to generate new corrections based on already generated corrections which allows to make all corrections at one model execution. However, the limitation of these approach is that the model couldn't be forced to correct any known mistakes, which positions can be determined using another approach, which leads to behavior, when model corrects certain mistakes in one sentence, but skips it in another.

3. Methods and materials

In our research, we introduce model architecture that allows us to combine algorithmic approaches, like dictionary-based misspellings detection. We assume that by doing this we can make our solution more predictable and robust.

3.1. Detecting errors in the input sequence

There are three main operations that are done during text error correction:

- Replace.
- Delete.
- Insert.

These operations are done on token-level to transform the input sequence into the corrected one. To detect such tokens, we use a similar method as described in the GECToR [6] paper. For this, pretrained transformer encoder is used, like BERT, RoBERTa, XLNet etc. The input sequence is tokenized using pretrained tokenizer of specific model and then forwarded to the encoder, projection layer is applied to the encoder's last hidden state. Projection layer is used for binary classification of each input token, where 0 is correct, and 1 is incorrect and need to be corrected. The overview of how detecting incorrect tokens in input sequence is done is shown on Figure 1.

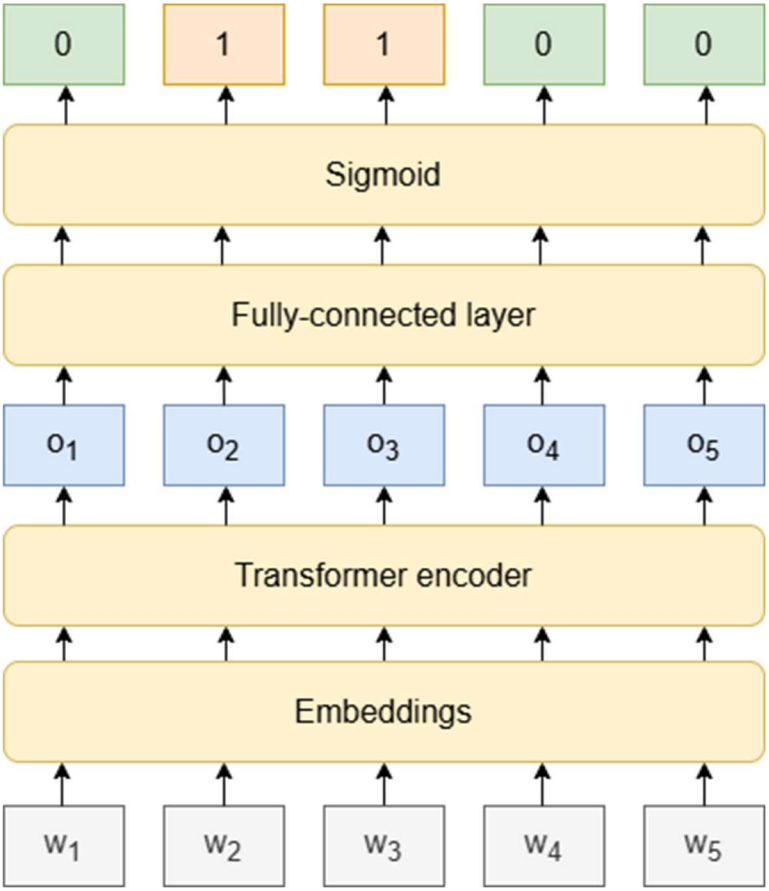

Figure 1: Detecting erroneous tokens in the input sequence

After getting a vector of the same size as input sequence, which consists of 0 and 1, it should be used to rewrite those tokens, that were classified as incorrect. Consequence of ones in such vector form a chunk, that should be rewritten entirely. So, the next step is to convert this vector to a matrix, in which each row contains a single chunk to process them independently. For a given vector v, we compute the difference vector:

$$d_i = v_i - v_{i-1}, v \in \{0,1\}^n, \tag{1}$$

where $d_0 = v_0$, $d_n = -v_{n-1}$.

Using the difference vector, start and end indices could be found:

$$S = \{s_k | d_{s_k} = 1\}, \tag{2}$$

$$E = \{e_k | d_{s_k} = -1\}. \tag{3}$$

For each start and end pair we can create a vector, containing a single chunk:

$$c_k(i) = \begin{cases} 1, & if\ s_k \le i < e_k \\ 0, & otherwise \end{cases}. \tag{4}$$

After this, all chunks can be stacked into a matrix:

$$C = \begin{matrix} c_0 \\ c_1 \\ c_m \end{matrix}, \tag{5}$$

where m – number of chunks.

The example of dividing correctness vector to chunks is shown in Figure 2.

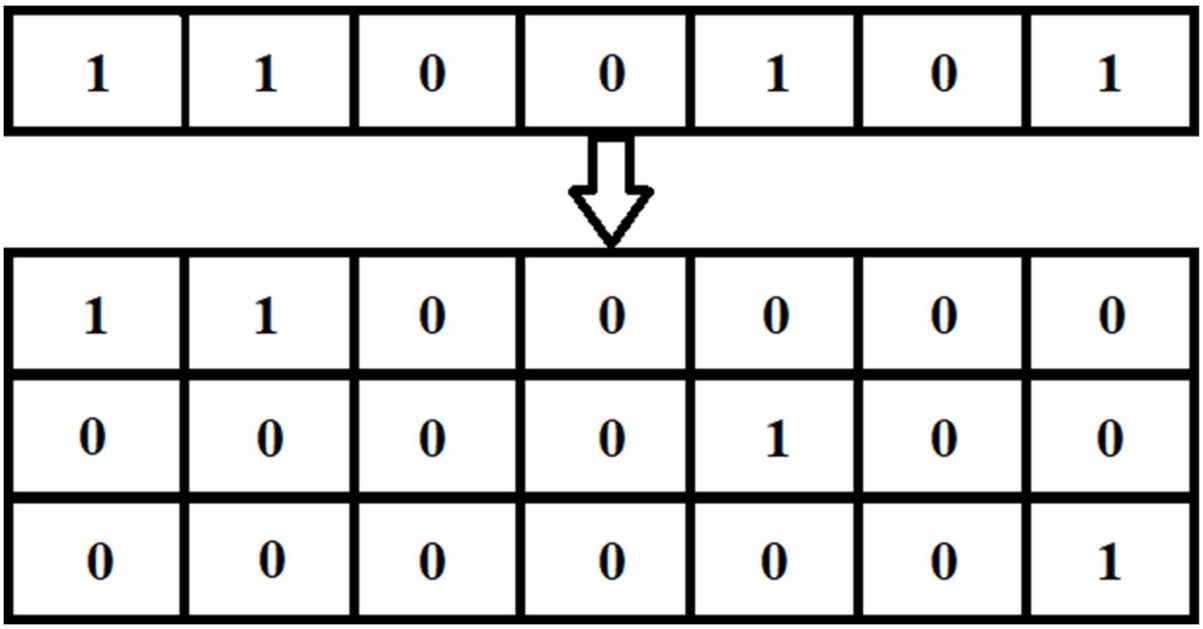

Figure 2: Dividing correctness vector into chunks example

3.2. Rewriting selected parts of input

After obtaining the positions in the input sequence that are needed to be corrected, replacements should be generated. In this research it is done by using highlight and decode technique with a transformer decoder [7]. Highlight and decode is a method of pointing the decoder exact tokens in the input sequence that require changes. It is done by adding a trainable embedding vector to those tokens. For each chunk, trainable embedding is added to the encoder last hidden state, where chunk vector value is one which is done according to formula:

$$H' = 1_m H + C \odot E, \tag{6}$$

where H – encoder last hidden state, E – trainable embedding, n – input sequence length, m – number of chunks, C – chunk matrix.

For each input sequence a batch size m is created for the decoder. Then, this batch of highlighted encoder hidden states is fed into decoder which is used to generate output sequence for each chunk. After detokenizing the generated sequences from the decoder, corresponding chunks in the input sequence are replaced by sequences generated by decoder.

3.3. Combining with algorithmic methods

As was stated earlier, some of the tokens can be considered as incorrect without using the model itself. For example, it can be done using a dictionary-based spell-checking system. For this, input sentences should be tokenized into words, and then each word is checked against the dictionary. If the word is not present in the dictionary, it is considered as incorrect. Given this, additional correctness vector can be created, that can be merged with correctness vector of the model using OR logical operation as shown in formula:

$$H_i = H_m(i) \vee H_s(i), i = 1,2, \dots, n, \tag{7}$$

where H_m – highlight vector produced by model, H_s – highlight vector produced by spell-checking system, n – input sequence length.

This combining increases the robustness of classifier, which could not detect some of the errors in certain contexts. In this research, combining with a spell-checking system is examined, but it also can be combined with other grammar checking systems, which allow to obtain the positions of error in the input sequence, but struggling to generate good suggestions. The whole system, including merging with algorithmic-based system results, is shown on Figure 3.

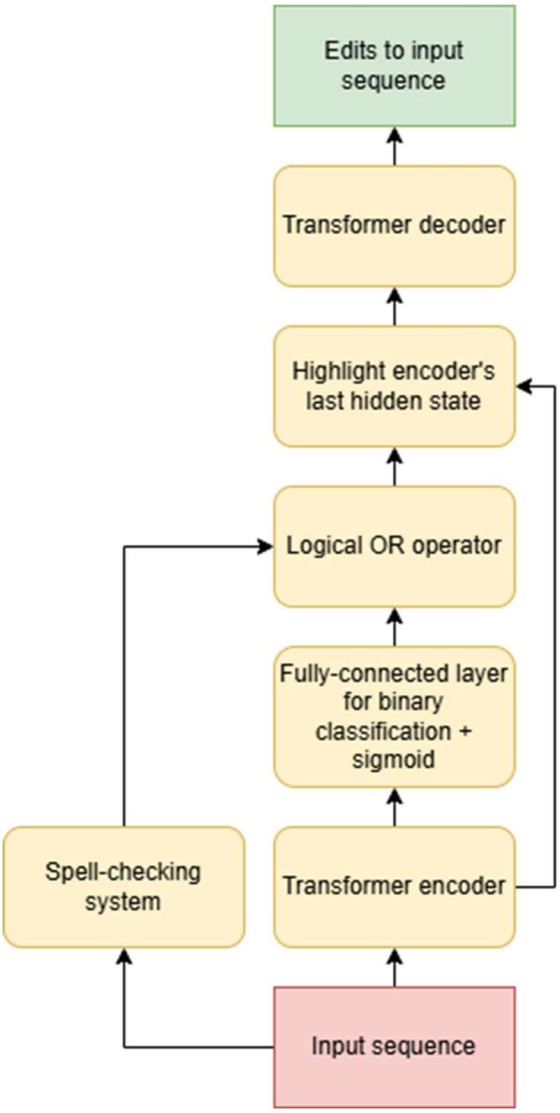

Figure 3: General diagram of the system

4. Experiment

4.1. Selecting pretrained encoder

As it was stated, a pretrained encoder is a core of the solution. The quality of the entire model depends on the chosen encoder. To test the pretrained model performance an experiment was done. The solution described in section 3 was implemented using different pretrained models. Every solution was fine-tuned on WI+LOCNESS [8] dataset train set and evaluated on validation set. Overall, three models were tested: BERT, RoBERTa and ModernBERT [9]. The result scores of solutions are shown in Table 1.

Table 1
Score of the solution based on different pretrained models fine-tuned on WI+LOCNESS dataset.

Model	Precision	Recall	F0.5
BERT	0.2566	0.2111	0.246
RoBERTa	0.3508	0.2375	0.3203
ModernBERT	0.376	0.2647	0.3469

As a result, ModernBERT was picked as a pretrained encoder, because it has shown the best preliminary results.

4.2. Training data

To train the model W&I+LOCNESS and CoEdIT [10] datasets were combined. CoEdIT was cleared of identical sentences from W&I+LOCNESS dataset. The number of train and test sentences is shown in Table 2. As this number of data instances is too small for model to generalize on the task in GEC task, synthetic data is often used. In this research c4_200m [11] synthetic dataset was used. This dataset contains millions of incorrect/correct sentence pairs.

Table 2
Datasets train and test splits size.

Dataset	Train	Test
W&I+LOCNESS	34308	4384
CoEdIT (filtered)	12522	485
Total	46830	4869

Most datasets for grammatical error correction task come in the format of sentences pairs, one sentence in a pair is a source and another one is a target sentence. First, the data was cleaned and normalized, which includes Unicode NFKC normalization, single and double quotes, hyphens and dashes, semicolons were normalized to their base variants. Additional spaces were removed, so there is not more than one space between the words. Text was tokenized using spaCy python library to divide words from punctuation. As model architecture described in this work is a sequence-to-edits type, pair of sentences need to be annotated, to find target edits that are needed to be made to convert the input sequence to the target sequence. For this, the ERRANT [12] library was used. This library provides a tool to annotate pairs of sentences and generate a list of edits that are stored using M2 format which is standard format for GEC task.

4.3. Training the model

During training, in one forward pass model is producing correctness vector using encoder's last hidden state and a classifier. BCE loss is used to calculate the loss of the classification part. In decoder, target correctness vector is used instead of produced one, to allow the decoder to generalize better and prevent error accumulation. Encoder's last hidden state is highlighted and fed into decoder, which is learned to produce target correction. For decoder, cross entropy loss function is used. The losses of decoder and classifier are combined using formula below.

$$l = l_c + l_d, \tag{8}$$

where l_c – classifier loss, l_d – decoder loss.

Model training was done in two steps: pretraining and fine-tuning. Pretraining was done using synthetic c4_200m dataset on 5 million random sentences. Half of the first encoder layers were frozen for the first hundred thousand steps, then whole model weights were unfrozen. All hyperparameters used during pretraining step can be found in Table 3.

Table 3
Pretraining step train hyperparameters.

Parameter	Value
Batch size	16
Learning rate	3e-5
Dropout rate	0.3
Optimizer	Adam

Separate losses of classifier and decoder during training are shown on Figure 4.

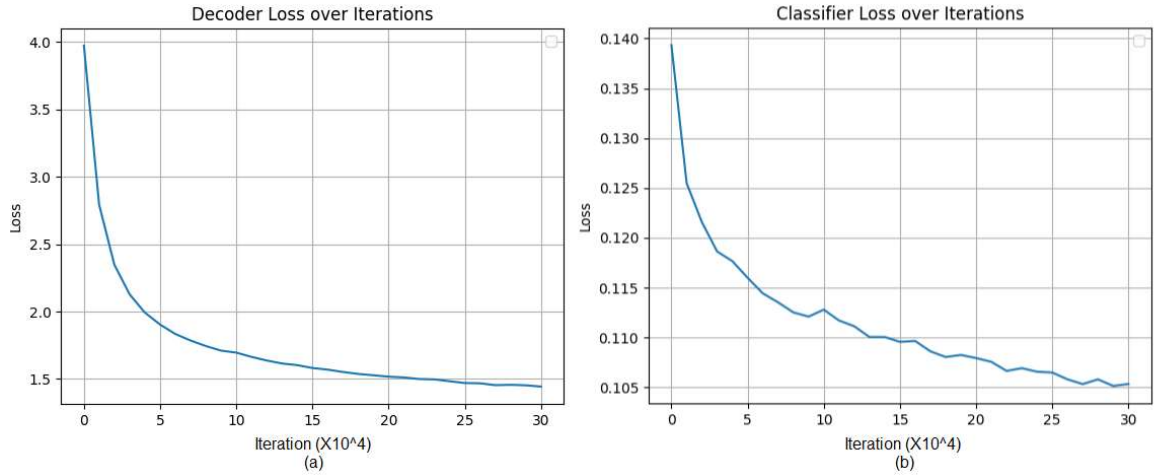

Figure 4: Loss graphs during pretraining: (a) – Decoder loss; (b) – Classifier loss

Fine-tuning part was done on combination of W&I+LOCNESS and CoEdIT datasets. At this part, all encoders were unfrozen, but to prevent overfitting the classifier, it was frozen for the first epoch. During the first epoch, only the decoder part was trained. Fine-tuning was done in 3 epochs, all training hyperparameters are shown in Table 4.

Table 4
Fine-tuning step train hyperparameters.

Parameter	Value
Batch size	16
Learning rate	1e-5
Dropout rate	0.15
Optimizer	Adam

5. Results

Evaluation of the model was done using test splits of W&I+LOCNESS and CoEdIT datasets separately. Validation was done using the positions of misspellings found by spell-check engine. There were two types of tests for each dataset:

- Only model is used without spellings positions.
- Model mixed with misspellings positions from spell-check engine.

Precision, recall and F0.5 score were calculated using the ERRANT tool. The results are shown in Table 5.

Table 5
Model evaluation on W&I+LOCNESS and CoEdIT datasets.

Dataset	Spell-check engine used?	Precision	Recall	F0.5
W&I+LOCNESS	No	0.5201	0.4005	0.4908
W&I+LOCNESS	Yes	0.5198	0.4024	0.4911
CoEdIT	No	0.5302	0.4143	0.5021
CoEdIT	Yes	0.5292	0.4156	0.5018

As these datasets mainly focused on grammatical errors, testing combination of spell-checking system with the model isn't affecting the results much. To test this combination the BookCorpus [13] dataset was used. Ten thousand text samples were taken and were considered grammatically correct. Randomly, spelling errors were made in texts using next operations:

- Delete a random character.
- Replace a random character with another character.
- Swap two random adjacent characters.

Making from 1 to 3 errors in a random word, ten thousand sentences with synthetically generated spelling errors were obtained. This dataset was used to evaluate the model in two scenarios:

- Without merging with spell-check engine results.
- With merging with a spell-check engine found errors positions.

As with other evaluation datasets, precision, recall and F0.5 score were calculated for these two scenarios. Results are shown in Table 6.

Table 6
Model evaluation results with and without using misspelling positions.

Method	Precision	Recall	F0.5
Without using misspelling positions	0.4745	0.6088	0.4964
Using misspelling positions	0.4782	0.6203	0.5011

6. Discussions

6.1. Discussions of the results

As shown in Table 5, the usage of spell-check engine to find coordinates of misspellings and use it inside the model isn't giving much impact on the results. The reason might be that W&I+LOCNESS and CoEdIT datasets are aimed mostly at grammatical problems rather than spellings. Table 6 is showing results on a synthetically created dataset which consists only of spellings. As it is shown, precision is not affected much by using spell-check engine, but recall is growing. It can be explained that positions from the spell-check engine can fix potential misclassifications of the classifier, but it's not giving decoder any advantage. So, by using a spell-check engine we achieve that the network isn't missing spelling errors. Combining the results of algorithmic approach and neural network allows us to control the behavior of the system, because we can control spell-check system by maintaining the dictionaries, allowing users to add their own words etc.

6.2. Limitations

Even though we can control the positions in the input sequence that will be changed by the model, model's decoder can generate the same suggestion as highlighted text in the input sequence. Another issue is that all corrections are generated by one batch, which means that the decoder is generating new corrections without the information about previous corrections. This results in the system being unable to generate two dependent corrections in different parts of the text, for example deleting the word in the beginning and inserting it at the end of the text.

6.3. Further development

To improve the quality of the system, additional steps in training can be taken. In this research, only two training steps were done: pretrain using synthetic data and fine-tuning. Additional datasets can be used as additional steps of training. Additional high-quality data can help model to generalize better on the task.

Another technique that wasn't used in this research but could potentially improve the results is training model to reproduce highlighted text. This can help to learn trainable highlight embedding and decoder to rewrite specific parts of the input sequence of different length. Another potential benefit is that it can allow the decoder to generate the same text that was highlighted if spell-check engine or classifier highlighted part of the text that doesn't need to be corrected which can turn a model's limitation into a potential advantage.

Another possible use-case of the system that is not connected with the GEC task but worth considering is a rewriting system. By removing the correctness classifier from the model and replacing it with a user's input rewriting system can be created where the user can control which part of the sentence needs to be rewritten. Currently, large language models are primarily used in text rephrasing tasks, which use a lot of computational resources. Such system can offer rewriting only a part of the text or even one word and don't spend resources on regenerating the whole input sequence.

7. Conclusions

In this paper the Grammatical Error Correction (GEC) model was presented. The model uses a sequence-to-edits approach to reduce the use of computational resources which is crucial for systems where user's text is checked on fly which results in a lot of requests. Presented model architecture allows to combine it with algorithmic approaches which are good at finding the positions of errors in the text but struggling to propose suitable correction. In this research combination with spell-check engine was shown. The result of combination with algorithmic approach is higher quality of proofreading and more predictive system results. Although the system demonstrated in this paper has certain limitations that were discussed, it shows good performance of evaluation metrics and can be further improved.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] A. Koyama, K. Hotate, M. Kaneko, M. Komachi, Comparison of grammatical error correction using back-translation models, Assoc. Comput. Linguistics (2021) 126–135. doi:10.18653/v1/2021.naacl-srw.16.

[2] A. Bout, A. Podolskiy, S. Nikolenko, I. Piontkovskaya, Efficient grammatical error correction via multi-task training and optimized training schedule, Assoc. Comput. Linguistics (2023) 5800–5816. doi:10.18653/v1/2023.emnlp-main.355.

[3] H. Zhou, Y. Liu, Z. Li, M. Zhang, B. Zhang, C. Li, J. Zhang, F. Huang, Improving seq2seq grammatical error correction via decoding interventions, Assoc. Comput. Linguistics (2023) 7393–7405. doi:10.18653/v1/2023.findings-emnlp.495.
[4] X. Sun, T. Ge, F. Wei, H. Wang, Instantaneous grammatical error correction with shallow aggressive decoding, Assoc. Comput. Linguistics (2021) 5937–5947. doi:10.18653/v1/2021.acl-long.462.
[5] B. Didenko, A. Sameliuk, RedPenNet for grammatical error correction: outputs to tokens, attentions to spans, Assoc. Comput. Linguistics (2023) 121–131. doi:10.18653/v1/2023.unlp-1.15.
[6] K. Omelianchuk, V. Atrasevych, A. Chernodub, O. Skurzhanskyi, GECToR – grammatical error correction: tag, not rewrite, Assoc. Comput. Linguistics (2020) 163–170. doi:10.18653/v1/2020.bea-1.16.
[7] B. Didenko, J. Shaptala, Multi-headed architecture based on BERT for grammatical errors correction, Assoc. Comput. Linguistics (2019) 246–251. doi:10.18653/v1/W19-4426.
[8] C. Bryant, M. Felice, Ø. E. Andersen, T. Briscoe, The BEA-2019 Shared Task on Grammatical Error Correction, Assoc. Comput. Linguist. (2019) 52–75. doi:10.18653/v1/W19-4406.
[9] B. Warner, A. Chaffin, B. Clavié, O. Weller, O. Hallström, S. Taghadouini, A. Gallagher, R. Biswas, F. Ladhak, T. Aarsen et al., Smarter, better, faster, longer: A modern bidirectional encoder for fast, memory efficient, and long context finetuning and inference, 2024. doi:10.48550/arXiv.2412.13663.
[10] V. Raheja, D. Kumar, R. Koo, D. Kang, CoEdIT: Text editing by task-specific instruction tuning, Assoc. Comput. Linguistics (2023) 5274–5291. doi:10.18653/v1/2023.findings-emnlp.350.
[11] F. Stahlberg, S. Kumar, Synthetic data generation for grammatical error correction with tagged corruption models, Assoc. Comput. Linguistics (2021) 37–47. URL: https://aclanthology.org/2021.bea-1.4/.
[12] C. Bryant, M. Felice, T. Briscoe, Automatic annotation and evaluation of error types for grammatical error correction, Assoc. Comput. Linguistics (2017) 793–805. doi:10.18653/v1/P17-1074.
[13] Y. Zhu, R. Kiros, R. Zemel, R. Salakhutdinov, R. Urtasun, A. Torralba, S. Fidler, Aligning books and movies: towards story-like visual explanations by watching movies and reading books, IEEE Int. Conf. Comput. Vis. (ICCV) (2015). doi:10.1109/ICCV.2015.11.