Master's Thesis

Transformers and Graph Neural Networks for Spell Checking

Sebastian Walter

Examiner: Prof. Dr. Hannah Bast
Adviser: Matthias Hertel

University of Freiburg
Faculty of Engineering
Department of Computer Science
Chair for Algorithms and Data Structures

May 23rd, 2022

Writing Period

22. 11. 2021 – 23. 05. 2022

Examiner

Prof. Dr. Hannah Bast

Second Examiner

Prof. Dr. Frank Hutter

Adviser

Matthias Hertel

Declaration

I hereby declare, that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.
I also hereby declare that my thesis has not been prepared for another examination or assignment, either in its entirety or excerpts thereof.

Dietrichsweiler, 23.05.2022

Place, date

Signature

Abstract

Spell checking is a general term for methods that detect or correct spelling errors in natural language text. Such methods can not only help with correcting human written text, they can also improve the performance and robustness of natural language processing systems dealing with misspelled text, when applied as a preprocessing or postprocessing step.

We study the usage of Transformers and graph neural networks for spelling error detection on sequence and word level, as well as spelling error correction. We show that open vocabulary sequence-to-sequence Transformers can perform well for spelling correction. We also experiment with ways to represent text not as sequences of tokens, but rather as word graphs to be processed with graph neural networks. As a simple way to boost spelling error detection performance we propose to enrich misspelled input texts with additional word features.

Our models perform better than most of the strong baselines for both artificially generated benchmarks and benchmarks built from real data. They achieve word accuracies of over 98% for word-level spelling error detection and F_1 scores of over 80% for spelling error correction, while running fast enough to be used in practice.

Contents

List of Figures

List of Tables

List of Abbreviations

NLP Natural language processing

OCR Optical character recognition

SEC Spelling error correction

SEDS Sequence-level spelling error detection

SEDW Word-level spelling error detection

GEC Grammatical error correction

LSTM Long short-term memory

RNN Recurrent neural network

Seq2Seq Sequence-to-sequence

BPE Byte pair encoding

GNN Graph neural network

MLP Multilayer perceptron

TR Tokenization repair

TR+ Tokenization repair plus spelling error detection

TR++ Tokenization repair plus spelling error detection and correction

MNED Mean normalized edit distance

1 Introduction

The earliest works on computer-based automatic spelling correction date back to the 1960s. And although the problem of detecting and correcting spelling errors has received a lot of attention in research ever since, it is still considered to be an ongoing and challenging problem today. In this work we focus on ways to apply and extend two recent deep learning architectures, the Transformer and graph neural networks, to different tasks revolving around detecting and correcting spelling errors in natural language text. We subsume these tasks under the term spell checking.

1.1 Motivation

The obvious use case for spell checking systems is to correct errors in human written text of any form, including emails, documents, text messages, and more. While this alone should be reason enough to get people interested in spell checking methods, there are more potential applications.

Wherever humans interface with computer systems via natural language, there exists the possibility of misspelled human input. Misspellings can happen while entering data into databases, searching for information, communicating with a chat bot, and so on. In almost all cases they come with detrimental effects on performance of natural language processing (NLP) systems working in the background. For example, Belinkov and Bisk (2018) report that noisy input texts can easily break machine translation systems. J. Gao et al. (2018) find that only small character-level perturbations in text

can lead to large perfomance drops for modern text classfication, sentiment analysis, and spam detection models. And Google reports that about 10% of the search queries they receive are misspelled (Pandu Nayak, 2021). If Google did not make an effort to be to able to deal with misspelled queries, one out of ten users might not find what they are looking for. For all these problems two possible solutions come to mind: Firstly, designing NLP systems in such a way that they are inherently robust against corrupted input. And secondly, fixing the corrupted input before putting it into the system. Spell checking is particularly useful to support the latter approach.
Sometimes we also have to deal with misspellings in text that was not put into, but generated by a computer system. The most prominent examples are optical character recognition (OCR) and PDF text extraction systems. These systems are known to be prone to confusing similar looking characters or omitting characters completely (Hládek, Staš, and Pleva, 2020; Kukich, 1992). Spell checking methods are one way to reduce such errors afterwards.
Overall, there is great benefit in detecting and correcting spelling errors both as a preprocessing method to improve performance of subsequent NLP systems and as a postprocessing method to correct outputs from OCR, text extraction, or other systems, that produce natural language text.

1.2 Problem statement and task definitions

To formalize the problem of detecting and correcting spelling errors in text we define three distinct but related tasks which we will look at in this work:

1. Sequence-level spelling error detection (SEDS)

Given a potentially misspelled text S, assign it a label $l \in \{0, 1\}$ where 0 means that S does not contain a spelling error and 1 means that S does contain a spelling

error.

Examples

This tetx has an eror! $\rightarrow 1$

This is a text without misspellings. $\rightarrow 0$

2. Word-level spelling error detection (SEDW)

Given a potentially misspelled text S, assign it a sequence of labels $L = (l_1, ..., l_n)$ with $l_i \in \{0, 1\}$. Each label l_i corresponds to the i^{th} whitespace separated word s_i in S. A label $l_i = 0$ means that s_i does not contain a spelling error whereas $l_i = 1$ means that s_i does contain a spelling error.

Examples

This tetx has an eror! $\rightarrow (0, 1, 0, 0, 1)$

This is a text without misspellings. $\rightarrow (0, 0, 0, 0, 0, 0)$

3. Spelling error correction (SEC)

Given a potentially misspelled text S, predict the corrected text without misspellings S'.

Examples

This tetx has an eror! $\rightarrow$ *This text has an error!*

This is a text without misspellings. $\rightarrow$ *This is a text without misspellings.*

The three tasks are ordered in increasing complexity and scope of application. This is because each task can be used as a subroutine to solve its preceding tasks. In other words, any spelling error correction algorithm can be used to detect spelling errors on

a word or sequence level. Also any algorithm that detects spelling errors on a word level can be used to detect spelling errors on a sequence level. We show how this can be done below. However, this does not mean that the SEDS and SEDW tasks are irrelevant in practice. On the one hand, detecting spelling errors might be enough for a particular application so there is no need to make the task more complex. On the other hand, spelling error detection methods can be leveraged to improve spelling error correction methods both regarding runtime and performance, which we will demonstrate later in this work.

SEDW using SEC For every whitespace separated word s_i in the input text $S = (s_1, ..., s_n)$, find whether s_i was changed by the SEC algorithm predicting S'. Afterwards the corresponding SEDW output $L = (l_1, ..., l_{|S|})$ is obtained by predicting

$$l_i = \begin{cases} 1 & \text{if } s_i \text{ was changed} \\ 0 & \text{else} \end{cases}.$$

How exactly one should determine if an input word s_i was changed can be dependent on the particular SEC algorithm in use. If e.g. a SEC algorithm guarantees that every input word s_i will be corrected to exactly one output word s'_i, meaning $|S| = |S'|$, the SEDW output $L = (l_1, ..., l_{|S|})$ can be obtained by predicting

$$l_i = \begin{cases} 1 & \text{if } s_i \neq s'_i \\ 0 & \text{else.} \end{cases}$$

An example of such an algorithm would be to replace every input word by its most similar word from a dictionary.
We present a way to convert SEC outputs to SEDW independent from the choice of SEC algorithm in section 5.2 and use it for our SEDW baselines.

SEDS using SEDW Reducing the output $L = (l_1, ..., l_n)$ of a SEDW algorithm

to the corresponding SEDS output can be achieved by predicting

$$l = \begin{cases} 1 & \text{if } \sum_1^n l_i > 0 \\ 0 & \text{else.} \end{cases}$$

This is equivalent to predicting that a text must contain a spelling error if any single word in the text contains a spelling error.

SEDS using SEC By transitivity every SEC output can also be reduced to a SEDS output by first reducing it to SEDW and then from SEDW to SEDS. Another equivalent and more intuitive way is to predict

$$l = \begin{cases} 1 & \text{if } S \neq S' \\ 0 & \text{else,} \end{cases}$$

implying that a text must contain a spelling error if it is changed by a SEC algorithm.

Limitations and boundaries

In this work we focus on spell checking English text only. Following Kukich (1992) and Hládek, Staš, and Pleva (2020), we define a spelling error to be an error falling into one of these two categories:

> **Cognitive errors**: Errors created by a person not knowing or having the ability to determine the correct spelling of a word. These errors are most of the time orthographically or phonetically similar to the intended word (e.g. *whale* $\rightarrow$ *wale*).
>
> **Typographic errors**: Errors created by a person mistyping on the input device by accident (e.g. *typing* $\rightarrow$ *typign*). OCR errors are usually also seen as typographic errors because they often originate from unintentional substitutions, insertions or deletions of single characters.

Note that one can not always make a clear decision about which category a spelling error belongs to. Both cognitive and typographic errors can be further assigned to two error types:

> **Nonword errors**: Errors that result in invalid words (e.g. *error* → *erorr*).
>
> **Real-word errors**: Errors that result in orthographically correct words, but are misspellings within their context (e.g. *He is from Germany.* → *He is form Germany.*).

We attempt to address all of these variants of spelling errors in this work: nonword, real-word, cognitive, and typographic errors. We think that errors regarding e.g. word choice, word ordering, inserting missing words, deleting superfluous words, or punctuation fall into the broader category of grammatical errors and do not consider them. In general, we think of correcting spelling errors as the minimum amount of editing we have to apply to obtain a correct English text.
Additionally, we assume that the text to be spell checked has correct whitespacing. This means that we expect no spelling errors of the following two forms:

- Merged words: *He isplaying football.*
- Split words: *He is playing foot ball.*

But since split and merged words occur frequently in practice (Kukich, 1992), e.g. in OCR systems, we will look at ways to relax this assumption later during this work.

2 Related Work

This chapter presents classical and modern Deep Learning based methods for spell checking and the related field of grammatical error correction (GEC). We state our contributions to spell checking at the end.

2.1 Classical methods

We consider all spell checking methods that do not involve a Deep Learning component as classical methods. Most of the early classical works focus on correcting misspelled words in isolation. That is, given a single misspelled word, find its correct spelling without having access to its surrounding context.
The pioneering work from Damerau (1964) identifies that about 80% of all misspellings are the result of applying one of the following four edit operations to a correct word:

- Deletion of a character,
- insertion of a character,
- replacement of a character,
- or transposition of two adjacent characters.

Based on this findings Damerau (1964) presents a spell checking technique to identify the correct spelling of words by comparing them with correctly spelled words from a dictionary and checking whether they either match exactly or they differ by one of the above mentioned edit operations. His simple technique is able to identify the correct

spelling for over 95% of words with single character misspellings. Together with Levenshtein (1966) this work resulted in the Damerau-Levenshtein edit distance metric for computing distances between text. Today, computing edit distances between misspelled words and words from a dictionary is still the central working principle of many modern spell checkers like Aspell (Atkinson, 2009).

By definition, correcting misspellings in isolation can only detect nonword errors. Real-word errors, which make up between 25% and 40% of all spelling errors (Kukich, 1992), need context information to be detected and corrected. Therefore, later works increasingly focused on context-dependent spelling correction. For example, Mays, Damerau, and Mercer (1991) use trigram word probabilities extracted from a large text corpus as a language model to score the well-formedness of sequences of words. Golding and Roth (1998) learn an ensemble of linear classifiers for every word in a vocabulary to predict whether a word belongs into the sentence it appeared in. The input to these classifiers are over 10,000 different features that are extracted from the word context in the sentence. During inference, the learned ensemble for every word in a confusion set produces an output activation given by a weighted majority mechanism among all classifiers. The highest output activation then determines the word from the confusion set that will be selected as the correct word belonging into the sentence.

2.2 Deep Learning methods

In contrast to classical methods, methods relying on Deep Learning are exclusively context-dependent. In fact, the ability to learn how to incorporate large contexts into their predictions from data is one of the main reasons for their success. Improvements in neural language models, the availability of large-scale datasets from the web, and advancements in machine learning accelerators such as GPUs and TPUs add to that.

2.2.1 Spelling error correction

We divide the research on spelling error correction into three groups based on the type of model each work uses: a unidirectional language model, a bidirectional encoder model, or an encoder-decoder model.

Unidirectional language model

Hertel (2019) proposes NLMSpell, a spelling corrector based on an unidirectional long short-term memory (LSTM) language model with attention. The language model is used to score candidate corrections from a candidate set for each word in an input sequence. When combined with beam search decoding this approach achieves high F_1 scores of over 91% and 88% for artifical and realistic misspellings respectively. However, the approach is slow because of the large amount of candidates it has to score and limited to correct misspellings consisting of up to two edit, merge or split operations with a fixed ouptut vocabulary of 100,000 words.

Recent large-scale unidirectional Transformer (Vaswani et al., 2017) language models like GPT-3 (Brown et al., 2020) or PaLM (Chowdhery et al., 2022) show remarkable task-agnostic zero-shot and few-shot performance. They are able to perform well on tasks they were not explicitly trained for simply due to their size (GPT-3 has 175B parameters, PaLM has 540B parameters) and the amount of data they are trained on. We are interested how they generalize to spell checking and use GPT-3 as a baseline for our spelling correction benchmarks later in this work. We especially expect it to excel at resolving ambiguous or context-dependent spelling errors because of its high language modeling capabilities.

Bidirectional encoder

Neuspell (Jayanthi, Pruthi, and Neubig, 2020) is a neural spelling error correction toolkit. It treats spelling error correction as a word-level classification task with a fixed output vocabulary. All available models in Neuspell compute character or subword-level representations using bidirectional neural models and then aggregate them to the word level before classification. Their best model is a finetuned version of BERT (Devlin et al., 2019). Neuspell also provides three training datasets that are all artificially corrupted versions of the One Billion Word Benchmark (Chelba et al., 2014) using different noising strategies, as well as four spelling error correction benchmarks created by extracting real-world spelling errors from the GEC datasets of the BEA-2019 GEC task (Bryant et al., 2019) and JFLEG (Napoles, Sakaguchi, and Tetreault, 2017).

Li et al. (2018) and Tran et al. (2021) propose similar two-stage approaches to spelling error correction that utilize both character and word information.
Li et al. (2018) first process the characters of each input word individually with a character recurrent neural network (RNN) for orthographic information and afterwards incorporate context across words with a second bidirectional RNN. A classifier then predicts a correction for each word. They outperform two other RNN based models and an open source spell checker on the JFLEG dataset.
Tran et al. (2021) predict for each word in the input sequence whether it is misspelled and, if so, predict a correction from a fixed output vocabulary. For encoding they use two standard Transformer encoders, one on character and one on word level, that are applied after each other. The authors achieve state-of-the-art results, but only train and evaluate their approach on Vietnamese data.

HCTagger (M. Gao, Xu, and Shi, 2021) builds upon the work by Awasthi et al. (2019) on local sequence transduction. It uses a pretrained bidirectional character-level language model and a bidirectional LSTM on top of that to encode misspelled text.

The encoded character representations are then used to predict edit operations for each character that specify how to correct the misspelled text. These edit operations are part of a predetermined fixed set of basic character transformations (e.g. keep a character, replace *a* with *o*, insert *e*, etc.). The authors achieve comparable results to a sequence-to-sequence (Seq2Seq) Transformer while being much faster. However, HCTagger is particulary designed for short text spelling error correction and only trained and evaluated on Twitter texts and search queries.

Encoder-decoder

Encoder-decoder models are widespread for tackling Seq2Seq problems. Generally, the encoder generates hidden representations of an input sequence that are then used by the decoder to autoregressively predict the output sequence.
Both Y. Zhou, Porwal, and Konow (2019) and Ahmadi (2018) treat spelling correction as a Seq2Seq machine translation problem from misspelled to correct texts.
Y. Zhou, Porwal, and Konow (2019) apply character-level and word-level LSTMs to e-commerce queries and get better results than prior work that uses statistical machine translation on the same domain. Ahmadi (2018) applies a character-level LSTM to Arabic text and compares it to unidirectional and bidirectional sequence-labeling LSTMs.
Finally, Hertel (2019) also studies the Transformer encoder-decoder model for translating incorrect to correct sequences, but finds that it performs worse than the language-model-based spelling corrector NLMSpell mentioned above.

Overall, we consider Jayanthi, Pruthi, and Neubig (2020), Li et al. (2018), and Hertel (2019) to be the most closely related works to ours because they are the only ones that focus on general purpose English spelling correction.

2.2.2 Grammatical error correction

GEC can be seen as a superset of spelling correction. It includes spelling error correction as a subproblem, but further cares about issues related to text fluency, choice of words, punctuation and more.
For example, a spelling corrector might correct the sentence *We take an elavator to reach the top of the high bulding.* to *We take an elevator to reach the top of the high building.*, a GEC method to *We take an elevator to the top of the skyscraper.*.

Most works from the literature model GEC as a Seq2Seq problem, relying on different Deep Learning architectures like RNNs (Ge, Wei, and M. Zhou, 2018), convolutional neural networks (Chollampatt and Ng, 2018), Transformers (Junczys-Dowmunt et al., 2018), or hybrid approaches combining statistical and neural machine translation (Grundkiewicz and Junczys-Dowmunt, 2018).
Because Seq2Seq models in general require large amounts of data to work well, there has also been significant effort to improve GEC by synthetically generating datasets or unsupervised pretraining (Lichtarge et al., 2019; Grundkiewicz, Junczys-Dowmunt, and Heafield, 2019; Stahlberg and Kumar, 2021; Yasunaga, Leskovec, and Liang, 2021).

Current state of the art for GEC is achieved by GECToR (Omelianchuk et al., 2020). Similar to HCTagger (M. Gao, Xu, and Shi, 2021) for spelling correction, GECToR is a local sequence transduction method (Awasthi et al., 2019) that can alter sequences by predicting edit operations for each input token in parallel. GECToR defines its edit operations on token level (e.g. capitalize token, append *for* to token, keep token, etc.) and uses pretrained bidirectional Transformers like BERT (Devlin et al., 2019) or XLNet (Z. Yang et al., 2020) for encoding. Because correcting some grammatical errors might depend on the correction of others, GECToR can be applied for multiple iterations, such that the output of the previous iteration is the input to the current one.

2.3 Contributions

We contribute the following to the field of spell checking:

- We investigate the usage of Transformer-based model architectures for spelling error correction in the sequence-to-sequence framework.
- We investigate the usage of Transformer-based model architectures for spelling error detection in the bidirectional encoder framework.
- We propose a novel model architecture for spelling error detection formulated as a graph neural network.
- We propose the use of word features to enhance spelling error detection models.
- We study the use of pretrained tokenization repair models as fixed feature extraction backbones for spelling error detection and correction models.
- We provide an extensive evaluation of our models and various baselines on benchmarks generated by us and benchmarks from the literature.
- We show that spelling error detection can be used to improve spelling error correction models in runtime and performance.
- Our models match or exceed the performance of strong baselines on most of the benchmarks while achieving runtimes fast enough to be used in practice.
- We make all of our models accessible in an easy to use spell checking toolbox.[1]

[1] https://bastiscode.github.io/spell_check

3 Background

This chapter introduces relevant background information, notation, and definitions for the reader to understand the following chapters.

3.1 Tokenization

Tokenization is the process of splitting up text into smaller blocks. These block are called tokens. The set of all unique tokens a tokenization scheme can produce form a vocabulary. This vocabulary is typically limited in size to a maximum number of tokens and extended with special entries for unknown tokens (*<unk>*), padding tokens (*<pad>*), and tokens that mark the beginning (*<bos>*) or end (*<eos>*) of sequences.

The most common ways to tokenize text are:

- Whitespace tokenization: Split text on whitespaces.
  Tokenize this sentence! → *(Tokenize, this, sentence!)*
- Regex tokenization: Split text with a regular expression. This is often used to separate words from punctuation.
  Tokenize this sentence! → *(Tokenize, this, sentence, !)*
- Subword tokenization: Split text into chunks of one or more characters up to full words called subwords. Subwords are usually determined by extracting

frequency statistics from a training corpus.[1]

Tokenize this sentence! → (To, ken, ize, #this, #sent, ence, !)

- Character tokenization: Split text into its characters.

  Tokenize this sentence! → (T, o, k, e, n, i, z, e, #, t, h, i, s, #, s, e, n, t, e, n, c, e, !)

In this work all of the above mentioned tokenization schemes come into use. For subword tokenization in particular we use the byte pair encoding (BPE) algorithm.

Byte pair encoding BPE (Sennrich, Haddow, and Birch, 2016) starts with an initial vocabulary containing only single characters as tokens. It then extracts frequency statistics for all consecutive pairs of tokens in the vocabulary from a training corpus. The token pair with the highest frequency gets merged into a single token, added to the vocabulary and all of its occurrences in the training corpus are replaced with the merged token. This process is repeated for a given number of iterations. Merges across word boundaries are not allowed, which means that the final vocabulary can never contain tokens longer than a single word.
While the original formulation in Sennrich, Haddow, and Birch (2016) uses characters, BPE can also be used with all 256 bytes as initial vocabulary. In this case calculating frequency statistics and merging is done with consecutive pairs of bytes rather than characters. This approach has the benefit that it can represent all possible texts without a special *<unk>* token.

3.2 Transformer

The Transformer is an encoder-decoder model introduced by Vaswani et al. (2017). Its ability to dynamically exchange information over large contexts and fast parallel training and inference (in the encoder) has led to it being the backbone of virtually any state-of-the-start model in the NLP domain today.

[1] Spaces are shown as #.

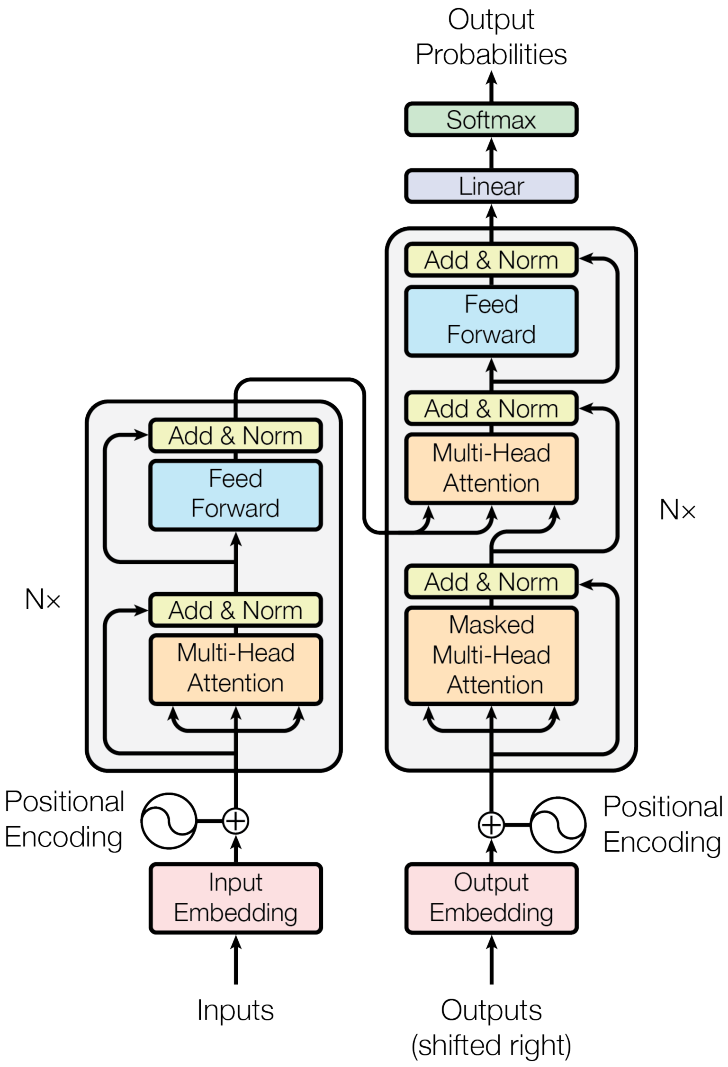

Figure 1: Transformer architecture (Image from Vaswani et al. (2017))

Figure 1 shows the Transformer model architecture. Its basic building blocks are a feed forward block[2] applied to all input positions separately

$$\text{ffn}(x) = \text{linear}(\text{activation}(\text{linear}(x)))$$

[2]Usually the inner linear layer projects to a high number of dimensions and the outer linear layer projects back to the original number of input dimensions.

and a multi-head attention block[3]

$$\begin{aligned} \text{multihead}(Q, K, V) &= (\text{head}_1 \| \dots \| \text{head}_h) W^O \\ \text{with attention}(Q, K, V) &= \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} V\right) \\ \text{and head}_i &= \text{attention}(QW_i^Q, KW_i^K, VW_i^V) \end{aligned}$$

where h is the number of attention heads, d_k is the dimensionality of K, and W^O, W_i^Q, W_i^K, W_i^V are learnable weight matrices.

The feed forward and multi-head attention blocks then form together with residual connections and layer normalization (Ba, Kiros, and Hinton, 2016) Transformer encoder and decoder layers. Encoder layers use a single self-attention[4] block for exchanging information among all inputs. Decoder layers use one masked self-attention block[5] for retrieving previous outputs and one cross-attention[6] block for retrieving encoder outputs.

Because the Transformer processes its inputs in parallel, the attention block is permutation invariant[7], and the feed forward block is permutation equivariant[8] there is no notion of position imposed on the inputs by the architecture itself (as opposed to e.g. RNNs and CNNs). However, for language tasks the order in which tokens appear usually contains important information. Vaswani et al. (2017) solve this by adding learned or fixed positional encodings to the inputs before feeding them into the Transformer.

3.3 Graph neural network

Graph neural networks (GNNs) are a class of neural models that can process graph-

[3] $\|$ denotes concatentation.
[4] A multi-head attention block with inputs $Q = K = V$.
[5] During training the self-attention mechanism at each decoder position is restricted to itself and previous positions to emulate the conditions during inference.
[6] A multi-head attention block with inputs Q and $K = V$.
[7] The output does not change when shuffling the inputs.
[8] The output changes in the same way the inputs are shuffled.

structured inputs. We follow the common way of formalizing a GNN via the message passing framework and adapt the notation from Hamilton (2020) and Hu et al. (2020).

A directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{A}, \mathcal{R})$ is a tuple of nodes $\mathcal{V}$, edges $\mathcal{E}$, node types $\mathcal{A}$, and edge types $\mathcal{R}$. The functions $\tau : V \to \mathcal{A}$ and $\phi : E \to \mathcal{R}$ map nodes $v \in \mathcal{V}$ and edges $e \in \mathcal{E}$ to their types. An edge $e = (u, v)$ from source node u to destination node v can be associated with a relation $(\tau(u), \phi(e), \tau(v))$ that is a tuple of the types of the source node, edge, and destination node. Graphs with one node type and one edge type ($|\mathcal{A}| = 1$ and $|\mathcal{R}| = 1$) are called homogeneous, graphs with more than one node or edge type are called heterogeneous.
Given a graph $\mathcal{G}$ and a set of node features $\{x_u \mid u \in \mathcal{V}\}, x_u \in \mathbb{R}^d$ we want to compute node representations[9] $z_u, \forall u \in \mathcal{V}$. Usually we initialize hidden representations

$$h_u^0 = x_u, \forall u \in \mathcal{V}$$

with the node features, perform L steps of message passing

$$h_u^l = \text{messagepassing}^l \left(h_u^{l-1} \right)$$

to update the hidden representations, and use the final hidden representations

$$z_u = h_u^L, \forall u \in \mathcal{V}$$

as node representations. Each message passing iteration can be seen as the equivalent to a layer in other neural architectures. Similar to other architectures we can either use the same set of learned parameters θ (parameter sharing) or learn and use a new set of parameters θ^l for every GNN layer.
The message passing step is executed in parallel for all nodes $u \in \mathcal{V}$ and composed

[9] There are also ways to incorporate edge features and compute edge representations with GNNs but we do not use them in this work.

of an aggregation and an update function. The aggregation function produces an aggregated representation $m^l_{\mathcal{N}(u)}$ over node neighbors[10] called a message. The message is then passed to the update function together with the current node representation:

$$\begin{aligned} h^l_u &= \text{messagepassing}^l\left(h^{l-1}_u\right) \\ &= \text{update}^l\left(h^{l-1}_u, m^{l-1}_{\mathcal{N}(u)}\right) \\ &= \text{update}^l\left(h^{l-1}_u, \text{aggregate}^l\left(\left\{h^{l-1}_v \mid v \in \mathcal{N}(u)\right\}\right)\right). \end{aligned}$$

The aggregation function has to be permutation invariant because it takes a set as input, but besides that it and the update function can be arbitrary differentiable functions like e.g. multilayer perceptrons (MLPs) (Hamilton, 2020). For heterogeneous graphs the aggregation function often consists of two stages: First, the messages for every relation in the neighborhood of u are computed separately. And then the messages are aggregated over all relations.

Transformer as GNN From a graph perspective the Transformer is an instance of a GNN that uses multi-head attention as aggregation function and a feed forward block as update function to perform message passing over a homogeneous fully-connected graph (Hamilton, 2020; Joshi, 2020; Bronstein et al., 2021).

3.4 Edit operations and distances

Given two strings a and b, we can calculate a sequence of basic character transformations that turn a into b. We call them edit operations. The minimum number of edit operations it takes to turn a into b is called the edit distance. If we allow insertions, deletions, and replacements of characters as possible transformations, we call this the Levenshtein edit distance. Further allowing the transposition of two adjacent character gives us the Damerau-Levenshtein edit distance (Damerau, 1964;

[10] The neighbors $\mathcal{N}(u) = \{v \mid (v, u) \in \mathcal{E}\}$ of a node u are all nodes for which an edge from v to u exists.

Levenshtein, 1966; Boytsov, 2011). In practice, calculating the edit distance between two strings is typically implemented with the dynamic programming approach by Wagner and Lowrance (1975).

Unless stated otherwise, we always refer to the Levenshtein edit distance without transpositions when we talk about edit distances or edit operations in the following chapters.

3.5 Beam search

In Seq2Seq architectures the decoder at each step t models a distribution $p(y_t \mid x, y_{<t}), y_t \in \mathcal{Y}$ over an output vocabulary $\mathcal{Y}$ given an input sequence x and previous outputs $y_{<t} = (y_1, \ldots, y_{t-1})$.
The overall probability of an output sequence $y = (y_1, \ldots, y_t)$ generated by such a decoder is the product of probabilities at each step:

$$p(y_1, \ldots, y_t) = \prod_{i=1}^{t} p(y_i \mid x, y_{<i}).$$

A common procedure is to generate the output sequence by choosing the output with the highest probability at each step. This is called greedy search:

$$y_t = \underset{y_v \in \mathcal{Y}}{\operatorname{argmax}}\, p(y_v \mid x, y_{<t}).$$

However, greedy search can find sub-optimal solutions because decoding paths with lower probability will never be visited even if they would turn out to have a higher overall probability in the end.
In contrast, beam search with beam width b keeps the top b partial solutions at each step based on their sequence probability (Y. Yang, Huang, and Ma, 2018). Note that beam search with $b = 1$ is equal to greedy search. At step t the set of partial solutions

B_t will therefore be (notation adapted from (Y. Yang, Huang, and Ma, 2018)):

$$B_0 = \{((<bos>), 1)\},$$
$$B_t = \underset{b}{\text{top}}\{(y' \| y_t,\ s \cdot p(y_t \mid x, y')) \mid (y', s) \in B_{t-1} \text{ and } y_t \in \mathcal{V}\}.$$

While still not being optimal, beam search usually performs better than greedy search at the cost of being slower and requiring more memory.

In our batched implementation of beam search, we record all finished paths (ending with $<eos>$) that we encounter during decoding and stop once the overall best path finishes or we hit a predetermined maximum search depth $d_{\max}$. All solution candidates are scored by their normalized sequence probability and the best one is returned as solution:

$$\underset{1}{\text{top}} \left\{ \left(y, \frac{p(y)}{|y|} \right) \;\middle|\; y \text{ ends with } <eos> \text{ or } |y| = d_{\max} \right\}.$$

4 Approach

This chapter describes the proposed models for the SEDS, SEDW, and SEC tasks as well as other important aspects like the datasets used or the spell checking procedure.

4.1 Data

We use a full English Wikipedia dump[1] (Wikidump) and the Bookcorpus dataset[2] as our base datasets. They contain diverse texts from various domains and can be assumed to be mostly free of misspellings.
We extract articles from Wikidump using WikiExtractor[3] and split them and all books in Bookcorpus into paragraphs on newlines. Finally, to clean the data we remove empty paragraphs, paragraphs containing any kind of markup, and paragraphs that do not contain any alphabetic characters using regular expressions and remove duplicate, leading, and trailing whitespaces from those who did not get removed.
By this procedure we end up with 69,174,513 cleaned paragraphs for Wikidump and 35,459,363 for Bookcorpus. We reserve about 1% of all cleaned data for development and testing, the rest is used for training. We also use the three training datasets by Neuspell (Jayanthi, Pruthi, and Neubig, 2020) with a total of 4,095,638 misspelled-correct sentence pairs for finetuning on sentence-level data.

[1] https://dumps.wikimedia.org/enwiki/latest/enwiki-latest-pages-articles.xml.bz2
[2] https://battle.shawwn.com/sdb/books1/books1.tar.gz or https://huggingface.co/datasets/bookcorpusopen
[3] https://github.com/attardi/wikiextractor

4.1.1 Tokenization

For regex tokenization we employ the following regex: `\w\S+\w|\w+|[^\w\s]+`. This is a modified version of the regex used by Hugging Face (2022) for splitting strings into words. It differs by the first expression `\w\S+\w` which is there to avoid splitting hyphenated words or words containing other punctuation marks in between their first and last characters. For example, the Hugging Face regex `\w+|[^\w\s]+` splits *Welcome to U.S.A!* into *(Welcome, to, U, ., S, ., A, !)*, but with our added expression the text splits into *(Welcome, to, U.S.A, !)*. Note that we keep track of the whitespace information in the input text, which is removed when splitting with our regex, so we can restore the original text anytime.

For subword tokenization we use the byte-level BPE implementation from Hugging Face (2022). We respect whitespace information as leading whitespaces belonging to the word that comes after them (e.g. *Welcome to U.S.A!* is tokenized into *(W, el, c, ome, #to, #U, ., S, ., A, !)*. We train our tokenizer to a maximum vocabulary size of 10,000 tokens on a subset of the Wikidump and Bookcorpus training paragraphs. This is a decent tradeoff between vocabulary size and tokenization granularity which affects training and inference speed.

4.1.2 Misspellings

To our knowledge there exist no large datasets with pairs of misspelled and correct sequences for spelling error correction. Similar to most prior work we therefore focus on corrupting correctly spelled texts to generate training data ourselves. We define two methods to inject misspellings into text that we expect to cover the spelling error types occurring in real data:

- **Artificial** for generating arbitrary typographic errors and
- **Realistic** for generating real-world typographic and cognitive errors.

Both methods are able to generate real-word and nonword errors, so they are covered too.

Artificial

We produce artificial misspellings by applying one or more of the four edit operations insertion, deletion, replacement and transposition defined by Damerau (1964) to correctly spelled words.
First, we sample the number of edit operations to apply k from a geometric distribution

$$\mathrm{P}(X = k) = p \cdot (1 - p)^{k-1}$$

with $k \in \mathbb{Z}^+$ and $p = 0.8$. This follows the findings in Damerau (1964) because about 80% of our artificial misspellings will be the result of applying one edit operation.
Afterwards, we perform k rounds of editing. Each round, we first uniformly sample the type of edit operation from all valid edit operations (e.g. we can not perform a transposition if the word only has one character). For insert and replace operations we also uniformly sample the character to insert or use as replacement from the set of all lowercase and uppercase letters in the English alphabet. Then we uniformly sample a valid character position within the word and apply the edit operation at that position. We keep track of the character positions that were previously edited, such that when we sample $k > 1$ we do not edit the same characters multiple times. This way we can be sure the word will be edited and avoid cases like e.g. deleting a previously inserted character.

Realistic

To generate realistic misspellings we replace correctly spelled words with a uniformly sampled misspelling from a confusion set. We build the confusion sets using the

following sources of misspellings:

- Misspelling corpora made available by Roger Mitton[4]
- Wikipedia's list of common misspellings[5]
- Spelling errors extracted from essays written for TOEFL exams [6]
- Aspell's and Hunspell's[7] top five replacement suggestions for every word in an English dictionary
- Typos extracted from Tweets[8]
- List of homophones[9]
- Facebook's list of 20M pairs of correctly and misspelled words, generated based on an error model extracted from search engine query logs (Piktus et al., 2019)[10]
- Errors of type *R:SPELL*, *R:ORTH*, *R:NOUN:INFL*, *R:NOUN:NUM*, *R:VERB:FORM* *R:VERB:INFL*, *R:VERB:SVA*, or *R:VERB:TENSE* extracted from the annotated datasets of the BEA-2019 GEC task[11]
- Neuspell's word replacement list with pairs of correctly and misspelled words [12]

After filtering out any invalid words or misspellings we obtain a total of 2,303,867 misspellings for 119,725 correct words, which means there are on average 19 misspellings in the confusion set of every word. For example, the confusion set for the word *playing* contains 155 misspellings such as *plaiing*, *plaiyng*, *pleying*, *pplaying*, *playinng*, or *Playing*.

We split the misspellings into disjoint sets for training, development, and testing holding 95%, 2.5%, and 2.5% of all misspellings respectively.

Since we later want to evaluate on the Neuspell benchmarks which are built from sequences and spelling errors from the BEA-2019 datasets, we create a second set

[4]https://www.dcs.bbk.ac.uk/ ROGER/corpora.html
[5]https://en.wikipedia.org/wiki/Wikipedia:Lists_of_common_misspellings/For_machines
[6]https://github.com/EducationalTestingService/TOEFL-Spell
[7]http://hunspell.github.io/
[8]https://luululu.com/tweet/
[9]https://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/speech/database/homofonz/0.html
[10]https://github.com/facebookresearch/moe
[11]https://www.cl.cam.ac.uk/research/nl/bea2019st/
[12]https://github.com/neuspell/neuspell

of misspellings that contain misspellings from all but the last two sources from above. We use this set for training all models that will be evaluated on the Neuspell benchmarks. Not excluding those misspellings from our confusion sets could lead to serious data leakage.

Injecting misspellings

To inject misspellings into text we first split it with our regex tokenizer and then corrupt each word with probability $p_{\text{corrupt}} = 0.2$. Because we believe the errors created by the Realistic method to be harder to correct and more relevant in practice, we choose to create 80% of misspellings with the Realistic and 20% of the misspellings with the Artificial method. Therefore, we set $p_{\text{realistic}} = 0.8 \cdot p_{\text{corrupt}} = 0.16$ and $p_{\text{artificial}} = (1 - 0.8) \cdot p_{\text{corrupt}} = 0.04$, such that overall about 16% and 4% of all words will be realistic and artificial misspellings respectively.

We adjust $p_{\text{artificial}}$ and $p_{\text{realistic}}$ on a per-sample basis to account for words that can or must not be edited with our methods. Let n be the number of words in a text, $n_{\text{artificial}}$ the number of words which we can corrupt with the Artificial method,[13] and $n_{\text{realistic}}$ the number of words which we can corrupt with the Realistic method.[14] We update the corruption probabilities as follows:

$$p_{\text{artificial}} \leftarrow \begin{cases} \frac{p_{\text{artificial}} \cdot n}{n_{\text{artificial}}} & \text{if } n_{\text{artificial}} > 0 \\ 0 & \text{else,} \end{cases}$$

$$p_{\text{realistic}} \leftarrow \begin{cases} \frac{p_{\text{realistic}} \cdot n}{n_{\text{realistic}}} & \text{if } n_{\text{realistic}} > 0 \\ 0 & \text{else.} \end{cases}$$

Note that if $n = n_{\text{artificial}} = n_{\text{realistic}}$ $p_{\text{artificial}}$ and $p_{\text{realistic}}$ do not change. We ignore that $p_{\text{artificial}} + p_{\text{realistic}}$ in theory could get larger than one, because this never happens

[13] All except for numerical words or punctuation marks.
[14] All except for numerical words, punctuation marks, and words with an empty confusion set.

in practice due to $n_{\text{artificial}}$ and $n_{\text{realistic}}$ typically being very close to n.

4.2 Models

This section introduces our model architectures for spelling error detection and correction.
In accordance with our problem statement (see section 1.2) most of the models are designed to work on text without split and merged words. However, we also state that handling such errors is an important problem in practice that, if done properly, significantly improves the capabilities of a spell checking system. Therefore, we adapt the tokenization repair problem as an auxiliary task for spell checking and integrate it into some of our models.

Tokenization repair The term tokenization repair (TR) was coined by Bast, Hertel, and Mohamed (2020) and describes the task of correcting missing or spurious spaces in text. Bast, Hertel, and Mohamed (2020) show that it is possible to achieve good results for this problem both with and without misspelled input texts. They also show that first repairing the whitespacing in text can significantly improve spelling error correction methods applied afterwards. For our implementation we follow the work by Walter (2021) which uses a Transformer encoder to predict whitespace repair tokens for all characters in a text at once. This approach is much faster than the one from Bast, Hertel, and Mohamed (2020) who use bidirectional LSTMs and beam search, but gives similar quality results.

All our models are implemented using PyTorch (Paszke et al., 2019). To implement GNNs we use the PyTorch backend of DGL (M. Wang et al., 2019).
We use a hidden dimensionality $d_{\text{hidden}} = 512$ and learned input token embeddings to which we add fixed sinusoidal positional encodings (Vaswani et al., 2017) for all models.

4.2.1 Models for spelling error detection

We only define models for SEDW. For SEDS we convert the outputs from SEDW models to the sequence level. During initial experiments with models explicitly designed for SEDS we found them to be much more unstable during training[15] and more sensitive to distribution shifts in the number of misspellings between training and test time than SEDW models.

We treat SEDW as a binary classification task and optimize all models with a standard binary cross-entropy loss function. To achieve fast inference speeds we employ two bidirectional encoder architectures, a Transformer encoder and a GNN, both of which can detect spelling errors in parallel.

Furthermore, we have the idea of enriching regular text input with word-level features specifically tailored towards spelling error detection. We perform our experiments with the following 13 binary features:[16]

- *is_punct*: All characters of the current word are punctuation marks.
- *is_currency*: The word is a currency symbol.
- *is_digit_or_like_num*: The word is a number or represents one.
- *like_url*: The word resembles an URL.
- *like_email*: The word resembles an email address.
- *has_trailing_whitespace*: The character after the word is a whitespace.
- *is_title*: The word is in title case (first character is capitalized, the others are lowercase).
- *is_upper*: All characters of the word are capitalized.
- *is_lower*: All characters of the word are lowercase.
- *is_stop*: The word is in a list of common English words (e.g. *the*, *a*, or *and*).
- *is_alpha*: The word consists of alphabetic characters only.

[15] One reason could be that sequence level labels are much more prone to noise in the training data than word level labels.

[16] We derive all of the features except for *in_dict* and *lower_in_dict* from spaCy token objects (https://spacy.io/api/token).

- *in_ dict*: The word is in an English dictionary.
- *lower_ in_ dict*: The lowercase version of the word is in an English dictionary.

We hope that adding these features improves neural models by

- contributing strong low-noise signals for the presence (e.g. *in_ dict=false*) or absence (e.g. *is_ stop=true* or *is_ punct=true*) of spelling errors and
- providing a better starting point for training[17] which can help in speeding up convergence and finding regions of low loss especially during early stages.

The word features are supposed to be calculated for words after regex tokenization. Features like *in_ dict* or *is_ punct* only make sense for separated punctuation and word tokens. However, the output for SEDW is defined for words obtained by splitting on whitespaces. To avoid confusion between these two notions of a word, we will call words obtained by splitting on whitespaces whitespace words and words obtained by regex tokenization regex words from now on.
We propose the use of GNNs because they learn token-level and regex-word-level representations simultaneously and allow us to inject the regex-word features into the graph directly from the beginning. These are both reasons why it might be able to learn better representations than the Transformer architectures. Because GNNs take graph structured input we need to define how to convert texts into graphs first.

Graph representation

We tokenize a text into a sequence of regex words $w = (w_1, \ldots, w_n)$. We then tokenize every regex word w_i into a sequence of subword tokens x_i respecting leading whitespaces. Let $\mathcal{W}$ be the set of all words, $\mathcal{T}$ be the set of all subword tokens and

[17] One can argue that, given the *in_ dict* feature, the neural model starts training as out-of-dictionary classifier and not from a random initialization.

$s : T \to W$ be a function that given a subword token returns the word it belongs to. A word graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{A}, \mathcal{R})$ is then defined by nodes $\mathcal{V} = \mathcal{W} \cup \mathcal{T}$, edges

$$\begin{aligned}\mathcal{E} =& \{(w_{src}, w_{dst}) \mid w_{src}, w_{dst} \in \mathcal{W}\} \\ & \cup \{(x_{src}, x_{dst}) \mid x_{src}, x_{dst} \in \mathcal{T} \text{ and } s(x_{src}) = s(x_{dst})\} \\ & \cup \{(x_{src}, w_{dst}) \mid x_{src} \in \mathcal{T} \text{ and } w_{dst} \in \mathcal{W} \text{ and } s(x_{src}) = w_{dst}\},\end{aligned}$$

node types $\mathcal{A} = \{\textit{word}, \textit{token}\}$ and edge types

$$\mathcal{R} = \{\textit{connects to token}, \textit{connects to word}, \textit{in}, \textit{self}\}.$$

We define the node type function τ to be

$$\tau(v) = \begin{cases} \textit{word} & \text{if } v \in \mathcal{W} \\ \textit{token} & \text{else} \end{cases}$$

and the edge type function ϕ to be

$$\phi(u, v) = \begin{cases} \textit{in} & \text{if } u \in \mathcal{T} \text{ and } v \in \mathcal{W} \\ \textit{connects to token} & \text{if } u, v \in \mathcal{T} \text{ and } u \neq v \\ \textit{connects to word} & \text{if } u, v \in \mathcal{W} \text{ and } u \neq v \\ \textit{self} & \text{else.} \end{cases}$$

It follows that the set of possible relations in a word graph is

$$\begin{aligned}\{&(\textit{word, connects to word, word}), (\textit{word, self, word}), \\ &(\textit{token, connects to token, token}), (\textit{token, self, token}), \\ &(\textit{token, in, word})\}.\end{aligned}$$

A graphical depiction of a word graph for the text *Corect this sentesne!* is shown in Figure 2.

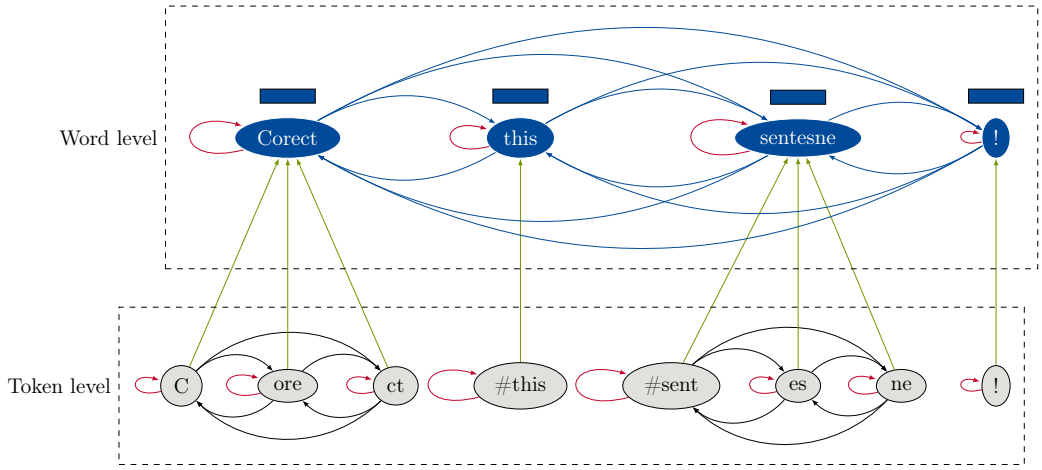

Figure 2: Word graph: A directed graph representing text on two levels: the word and the token level. On the word level the graph contains a node for every regex word in the text. The word nodes are then fully connected including self loops (blue and red edges on word level). On the token level we split each word further into subword tokens. We then fully connect all tokens within the same word including self loops (black and red edges on token level).[18] Finally we connect the two levels by adding an edge from every token to the word it is contained in (green edges from token to word level). The blue rectangles represent the additional word features associated with each regex word.

For reference we also show a token graph like it would be processed by the GNN equivalent of a Transformer encoder in Figure 4. The self-attention mechanism, being the aggregation function of a graph-based Transformer encoder, always aggregates over all tokens (see Figure 5).

Model architectures

Transformer and Transformer$^+$ Both models use the default Transformer encoder to encode sequences of subword tokens. The whitespace-word-level classification head for both models is a two-layer MLP with a GELU activation (Hendrycks and Gimpel, 2020) in between the layers.

For the Transformer model the inputs to the classification head are the whitespace-

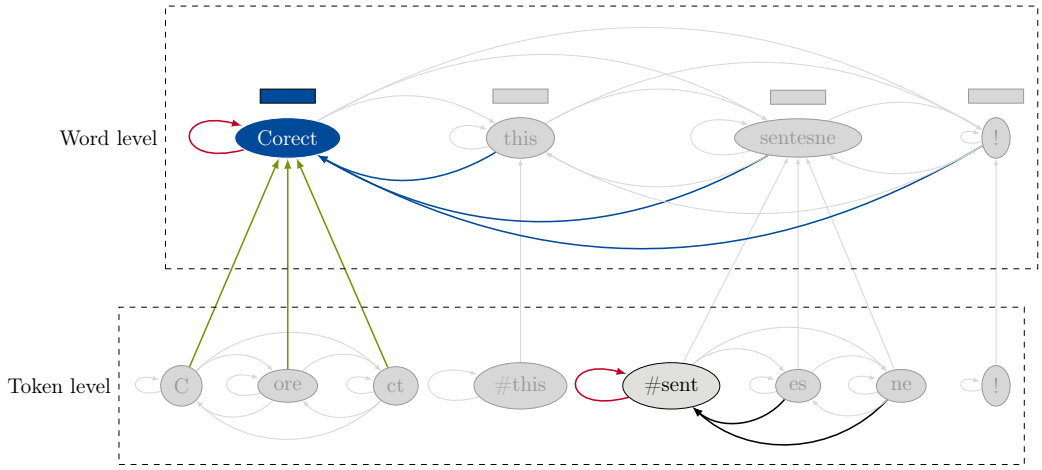

Figure 3: Word graph neighborhood: In a word graph the neighborhood of a word is given by all other words, the word itself, and all tokens belonging to it. In our example the *Corect* word has incoming edges from all other words and itself as well as from the tokens *C*, *ore*, and *ct*. A token in a word graph is connected with all other tokens that belong to the same word and itself. For example, the *#sent* token has incoming edges from the *es* and *ne* tokens and itself.

word-averaged token representations outputted by the last Transformer encoder layer. For the Transformer$^+$ model we add an additional aggregation stage between the encoder and the classification head. In this stage we first average the token representations to regex-word representations, concatenate those with our 13 additional word features, and pass them through a two-layer MLP with GELU activation. The regex-word representations we get out the MLP are further averaged to whitespace-word representations and fed into the classification head.

Figures 6 and 7 show the architectures of the Transformer and Transformer$^+$ models.

GNN and GNN$^+$ Both models receive a word graph as defined above as input. We use our learned input token embeddings to initialize the hidden representations $h_v^0, \forall v \in \mathcal{T}$ for all token nodes. To initialize the hidden representations $h_u^0, \forall u \in \mathcal{W}$ of

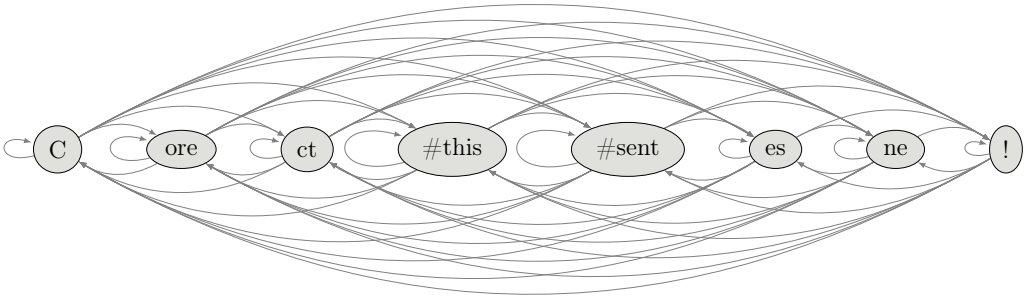

Figure 4: Token graph: A homogeneous, fully-connected, and directed graph of all tokens in a text including self loops. This graph structure is implicitly assumed when we process token sequences using a Transformer encoder with self-attention as mentioned in section 3.3.

the regex-word nodes, we average the token hidden representations per regex word:

$$T_u = \{v \mid v \in \mathcal{T} \text{ and } s(v) = u\},$$
$$h_u^0 = \frac{1}{|T_u|} \sum_{v \in T_u} h_v^0.$$

Next we need to define our aggregation and update functions. There are three flavors of GNN aggregation functions from which we can choose: Convolutional, attentional, or general message passing (Bronstein et al., 2021). We choose the attentional style which computes weighted aggregations based on the similarity between a node and its neighbors because it is the most comparable to the Transformer's multi-head attention.

There are a number of previous works that use attention-based aggregation functions in GNNs like Veličković et al. (2018), Hu et al. (2020), and X. Wang et al. (2021). Since our word graph is a heterogeneous graph we also want to respect the different types of relations in our aggregation function. That is why we choose to adapt a similar parameterization approach to Hu et al. (2020) that uses separate weights for each node and edge type and shares them across relations. To aggregate over the neighborhood

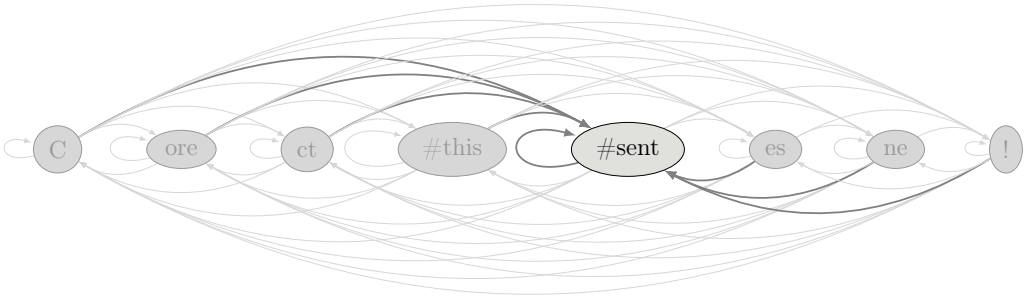

Figure 5: Token graph neighborhood: In a token graph every token is connected to every other token. This means that every single token has all tokens of the sequence in its neighboorhod. In our example the *#sent* token has incoming edges from all other tokens and itself.

of a node u, we determine all relations $R = \{(\tau(v), \phi(v,u), \tau(u)) \mid v \in \mathcal{N}(u)\}$[19] the node is a part of, separately aggregate over each relation using a multi-head attention mechanism, then sum and project the results to get the message $m^{l-1}_{\mathcal{N}(u)}$:

$$
\begin{aligned}
m^{l-1}_{\mathcal{N}(u)} &= \left(\sum_{r \in R} (\text{head}^r_1 \| \ldots \| \text{head}^r_h) \right) W^O_{\tau(u)}, \\
\text{head}^r_i &= \text{attention}\left(Q W_i^{(Q, r_{dst})} W_i^{r_{edge}}, K W_i^{(K, r_{src})} W_i^{r_{edge}}, V W_i^{(V, r_{src})} W_i^{r_{edge}} \right) \\
\text{with } Q &= h^{l-1}_u \| f_u \\
\text{and } K = V &= H^{l-1}_{\mathcal{N}(u,r)} \| F_{\mathcal{N}(u,r)},
\end{aligned}
$$

where $\mathcal{N}(u,r)$ are all neighbors of u for relation r and $F_{\mathcal{N}(u,r)}$ and f_u are the additional word features for the neighbors and u respectively that we concatenate with their hidden representations.[20] We can see that the learnable weight matrices W for the multi-head attention now also depend on node and edge types and that they are shared across multiple relations.[21]

[19] We access the type of the source node, edge, and destination node for a relation r with r_{src}, r_{edge}, and r_{dst}.

[20] Since we do not have additional features for token nodes they are just zero-dimensional vectors in this case.

[21] We also learn type dependent bias vectors together with every weight matrix but omit them here for clarity.

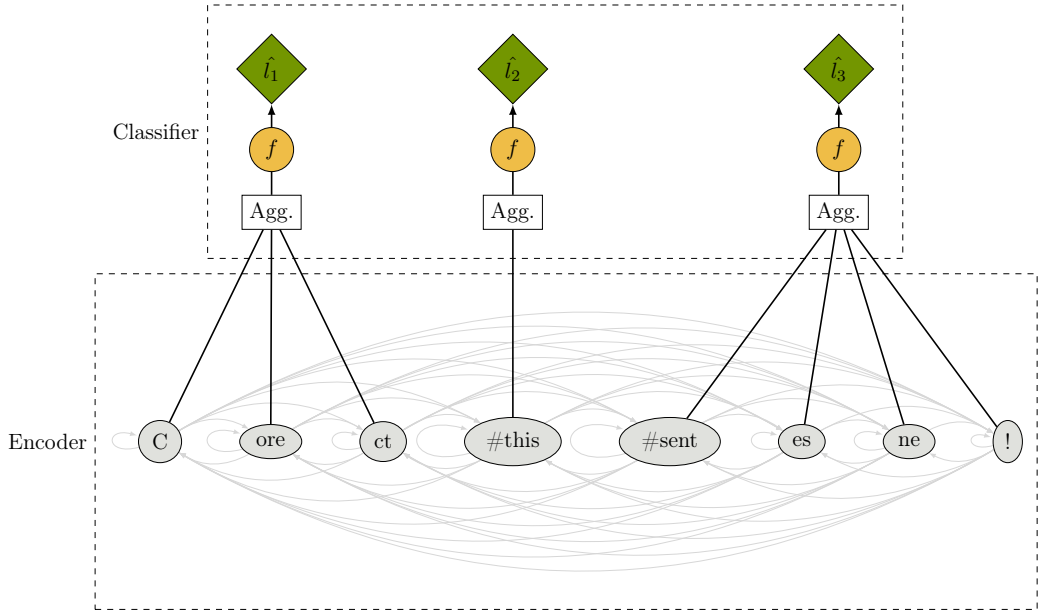

Figure 6: Transformer for word-level spelling error detection: The input token representations are encoded with a standard Transformer encoder, averaged to whitespace-word representations and passed through a shared MLP classifier.

After aggregation we employ a simple residual connection and layer normalization as update function to get h_u^l:

$$h_u^l = \text{layernorm}\left(h_u^{l-1} + m_{\mathcal{N}(u)}^{l-1}\right).$$

The final whitespace-word-level classification head is again a shared two-layer MLP whose inputs are the whitespace-word-averaged regex-word representations extracted from the word nodes of the word graph. The only difference between the GNN and GNN$^+$ models is that GNN does not use the additional word features in the word graph. A visualization of the GNN approach is shown in Figure 8.

Tokenization repair plus spelling error detection (TR+) This model uses the medium-sized pretrained tokenization repair model by Walter (2021) as backbone. We keep the parameters of the backbone fixed during training and only use it for TR and feature extraction. In particular, we choose to learn a weighted average over

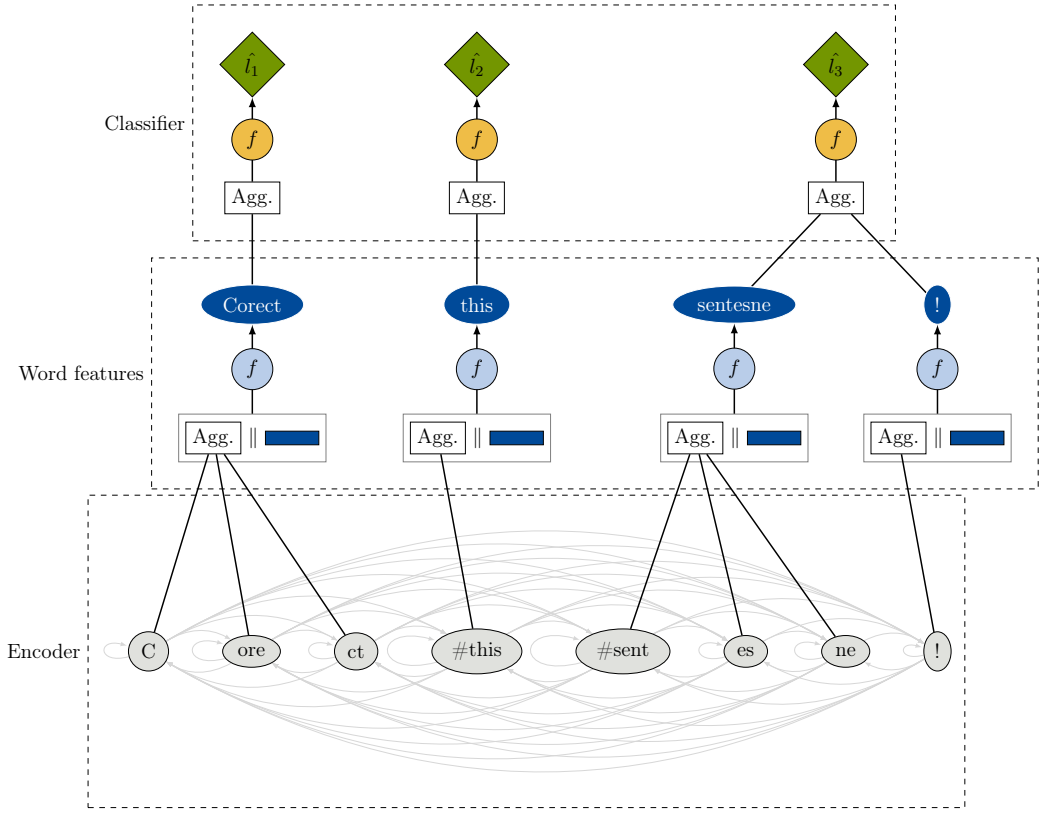

Figure 7: Transformer$^+$ for word-level spelling error detection: First, we encode the input tokens with a standard Transformer encoder. As an intermediate step we average the token representation to regex-word representations and concatenate them with additional word features. After passing the regex-word representations through a shared MLP we further average them to whitespace-word representations and feed them into a shared MLP classifier.

representations extracted from the last three layers of the model. The reason is that we can not tell which layer produces the best features for detecting spelling errors. After running the TR backbone, we are given the repaired text and the weighted character representations for every character in the original input text. We then average the character representations of those characters that end up in the same word in the repaired text to obtain a representation for every regex-word in the repaired text. If e.g. the input text *Cor ectt hi s sente sne!* gets correctly repaired to *Corect this sentesne!* we will average the representations of the characters at position 1, 2, 3,

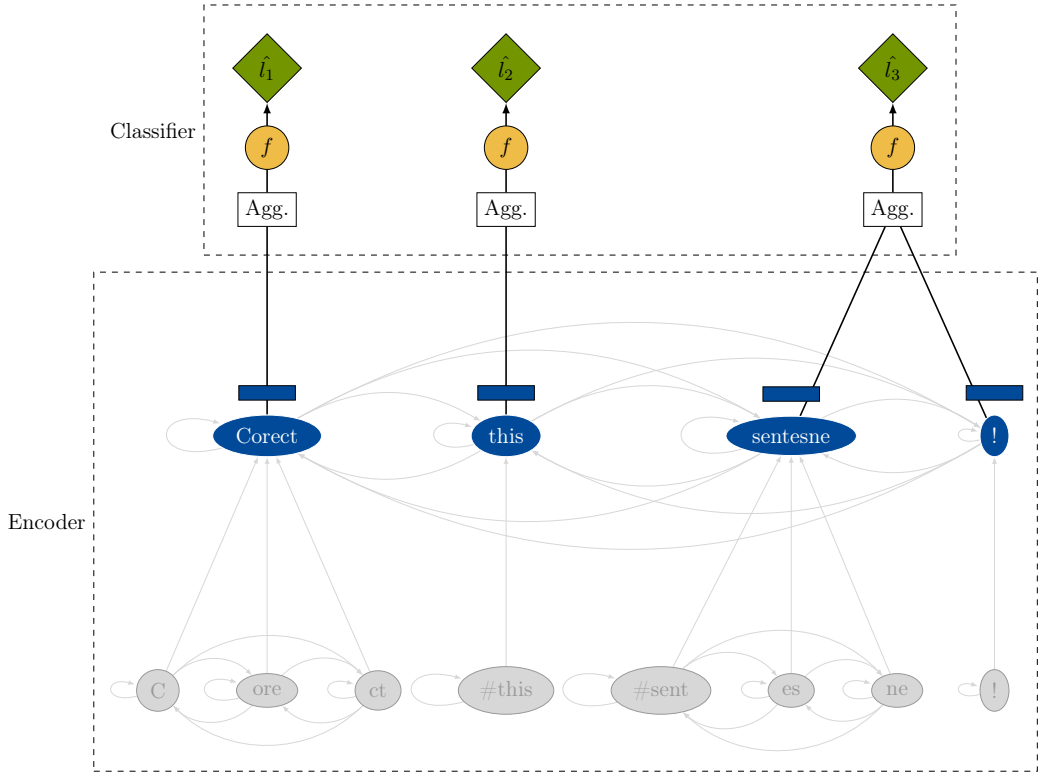

Figure 8: Graph neural networks for word-level spelling error detection: First, the word graph is encoded using a GNN. Afterwards we extract the regex-node hidden representations from the word graph, further average them to whitespace-word representations and pass them through a shared MLP classifier.

5, 6, and 7 in the input text to get the regex-word representation of the word *Corect*. At this point we add our word features to the regex-word representations and encode them further using a regular Transformer encoder. Then, the final step again consists of averaging the regex-word representations to whitespace-word representations and feeding them into a shared MLP classifier for spelling error detection. See Figure 9 for a depiction of this model.

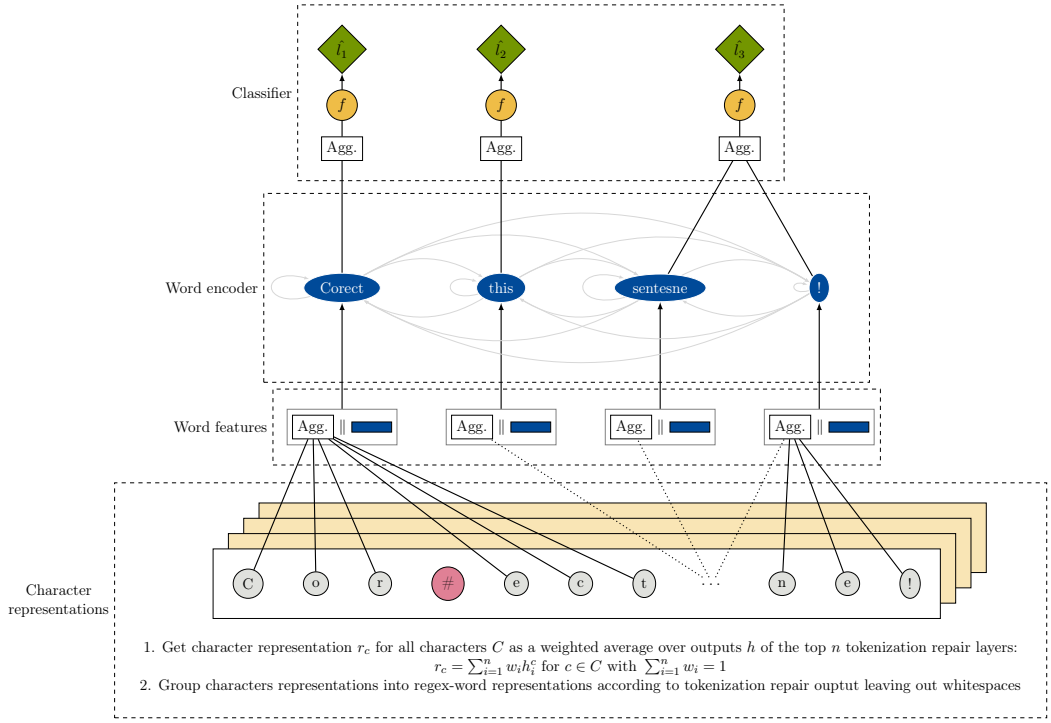

Figure 9: Tokenization repair plus spelling error detection: This approach uses a fixed tokenization repair model to repair and simultaneously extract character representation from an input text. We compute a weighted average of the character representations over the topmost tokenization repair layers and further average them into regex-word representations according to the repaired input text. We then add word features to the regex-word representations and encode them with a Transformer encoder. The outputs of this word-level encoder are averaged to whitespace-word representations and passed through a shared MLP classifier.

4.2.2 Models for spelling error correction

All of our models for spelling error correction are Transformer-based encoder-decoder models within the Seq2Seq framework. The main reason for this is that we want to be able to perform open vocabulary spelling correction instead of predicting words from a fixed vocabulary. In favor of this versatility we sacrifice the fast inference speeds achieved by methods relying on parallel encoders like Neuspell's BERT model or GECToR. However, later we also present ways that turn out to bridge the gap in speed between encoder-decoder and encoder-only models that make use of the special

properties of the spelling correction problem and our spelling error detection models. We treat spelling error correction as a multi-class classification problem over the set of possible next tokens and optimize all models using a categorical cross-entropy loss.

Transformer This model use the original Transformer encoder-decoder architecture unchanged. We turn a misspelled text into a sequence of subword tokens, encode it, and predict the correctly spelled output sequence one subword after another. Its architecture is shown in Figure 10.

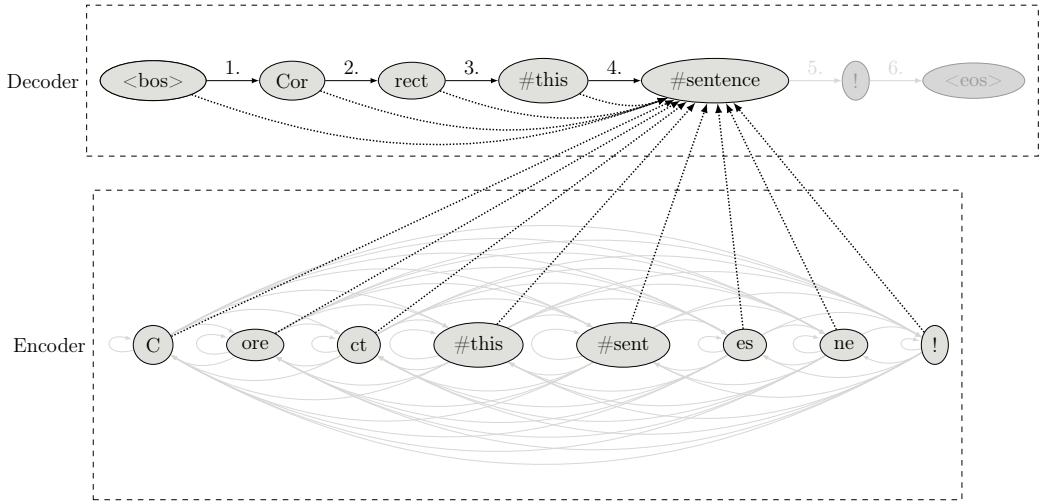

Figure 10: Transformer for spelling error correction: A Transformer encoder-decoder model used for transducing subword sequences with spelling errors into subword sequences without spelling errors. The decoder has access to all subword representations generated by the encoder and its own previous outputs.

Transformer word This model is equivalent to the Transformer model from above regarding its architectural components. However, we differ in the way we apply the decoder part of the model. Instead of predicting the full spelling corrected output sequence at once based on the full input context, here we restrict the decoder to only having access to single whitespace-word contexts and only train it to correct the corresponding whitespace word. This requires that the model incorporates all the relevant information from other subwords into the subword representations of each whitespace word during the encoding step, because the decoder will only have access

to those. But this change also enables us to correct all whitespace words in parallel by sharing the decoder among them since the correction of words that come later in a sequence does not need to wait anymore for all previous corrections to be finished. We consider this model to be a compromise between an encoder-decoder model that autoregressively corrects the whole input sequence and an encoder-only model that predicts corrections from a fixed vocabulary in parallel. Its architecture is shown in Figure 11.

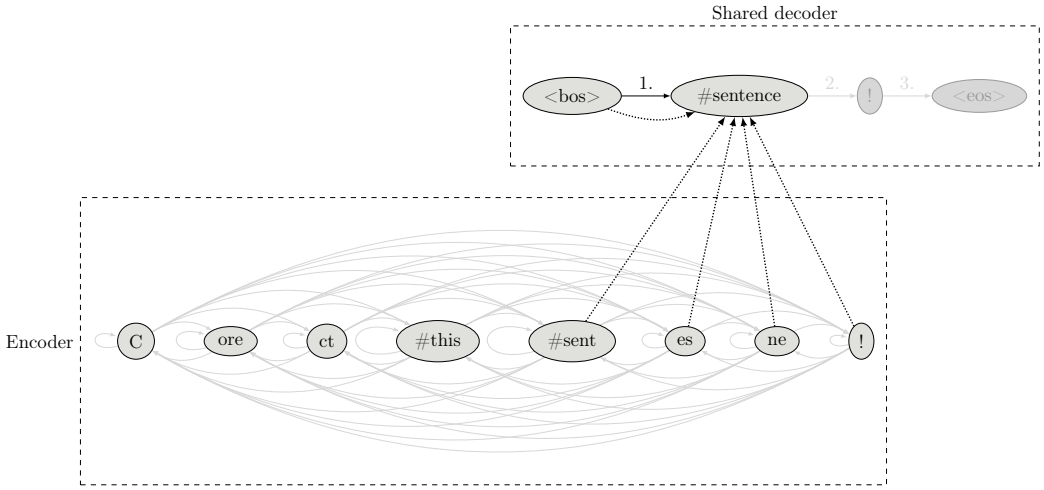

Figure 11: Word Transformer for spelling error correction: A Transformer encoder-decoder model used for transducing every whitespace-word in a text to its correction separately. The decoder has access to the encoder's subword representation of a single whitespace-word and its own previous outputs for that whitespace-word. For clarity, we only show the decoder as we would use it to correct the *(#sent, es, ne, !)* whitespace word, but the same is done in parallel for the whitespace words *(C, ore, ct)* and *(#this)*.

Tokenization repair plus spelling error detection and correction (TR++) This model extends the TR+ model with a shared decoder like in the Transformer word model. The context made available to the shared decoder are both the regex-word representations given by the Transformer word encoder and the averaged character representations from the tokenization repair backbone. Therefore, we change the standard Transformer decoder into a multi-context decoder, such that it is able to

perform not one but two multi-head cross-attention mechanisms per layer, for the regex word and character contexts respectively. Since the model tackles both the spelling error detection and correction tasks at once, we optimize it using the sum of their individual losses $\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{detection}} + \mathcal{L}_{\text{correction}}$.

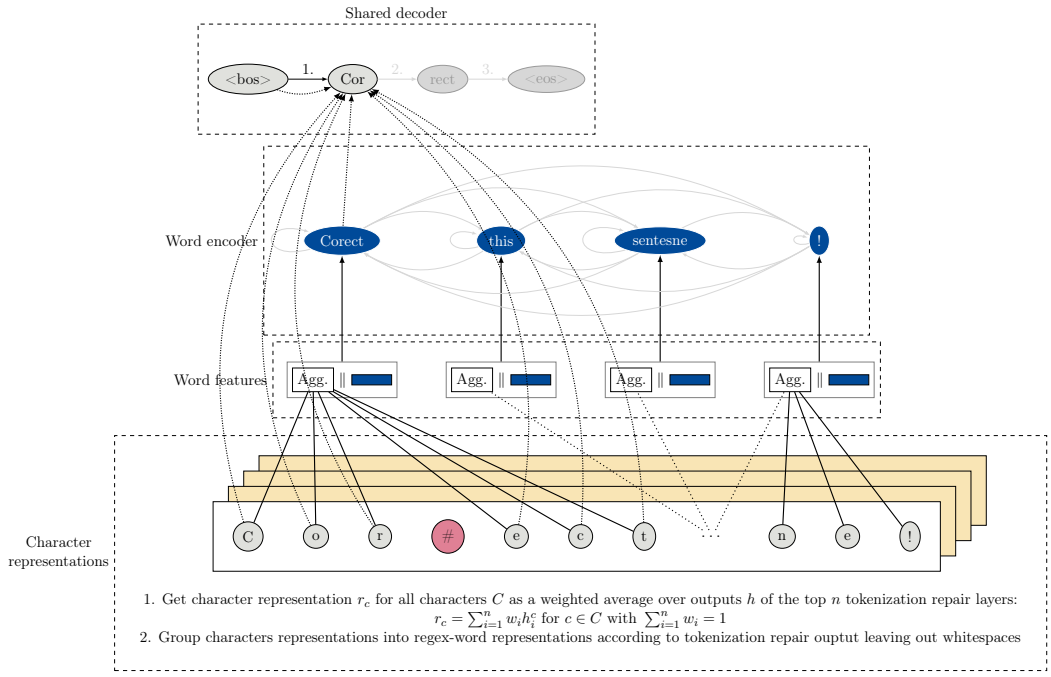

Figure 12: Tokenization repair plus spelling error detection and correction: The model extends the TR+ model (see Figure 9) with a shared decoder for correcting spelling errors analogously to the Transformer word model. The difference is that this model receives both the regex-word representations from the word encoder and the averaged character representations from the tokenization repair backbone as context for every whitespace word. The classification head for spelling error detection from TR+ is still in use but not shown here for clarity.

Transformer with tokenization repair Again, this model uses the original Transformer encoder-decoder architecture unchanged. But we tokenize its input on the character level and train it to correct sequences that contain whitespacing and spelling errors.

Table 1 gives an overview over all models with their parameter counts and layer specifications.

Task	Model	#Parameters	Inputs (Size)	Outputs	Layers
SEDW	transformer	18.0M	Subwords (10,000; BPE)	Binary classification	6 encoder layers
SEDW	gnn	23.5M	Word graph with subwords (10,000; BPE)	Binary classification	6 message passing layers
SEDW	transformer^{+}	18.5M	Subwords (10,000; BPE) and word features	Binary classification	6 encoder layers
SEDW	gnn^{+}	23.6M	Word graph with subwords (10,000; BPE) and word features	Binary classification	6 message passing layers
TR, SEDW	tokenization repair^{+}	32.1M (19.0M fixed)	Characters (99)	Tokenization repair, Binary classification	6 char encoder layers, 6 word encoder layers
SEC	transformer	41.8M	Subwords (10,000; BPE)	Sequence-level next token classification (10,000; BPE)	6 encoder layers, 6 decoder layers
SEC	transformer word	41.8M	Subwords (10,000; BPE)	Word-level next token classification (10,000; BPE)	6 encoder layers, 6 decoder layers
TR & SEC	transformer with tokenization repair	49.3M	Characters (99)	Sequence-level next token classification (10,000; BPE)	12 encoder layers, 6 decoder layers
TR, SEDW, SEC	tokenization repair^{++}	62.5M (19.0M fixed)	Characters (99)	Tokenization repair, Binary classification, Word-level next token classification (10,000; BPE)	6 char encoder layers, 6 word encoder layers, 6 multi-context decoder layers

Table 1: Model overview

4.3 Training

Before we start training we preprocess our cleaned Wikidump and Bookcorpus training paragraphs. The preprocessing mainly involves misspelling injection, tokenization, and subsampling or discarding of too long sequences (>512 tokens). An overview over our generated datasets is shown in Table 2.

We use the same preprocessed dataset for training SEDW and SEC models, only with different labels. For the TR+ and TR & SEC datasets we add whitespacing errors[22] on top of the already injected misspellings. They are the datasets we use to

[22]We remove all spaces or add spaces everywhere with 10% probability each, for the rest of the sequences existing spaces are deleted with 50% probability and new spaces are inserted with 10%

train the TR+/TR++ and Transformer with tokenization repair models.

For all models except for TR+, TR++, and the Transformer with tokenization repair model we will train two versions: One version trained on the SEDW/SEC dataset with the full set of misspellings as shown in Table 2 and one version on a SEDW/SEC dataset generated with a reduced set of misspellings (no misspellings from BEA-2019 and Neuspell, see subsection 4.1.2). The former versions of the models are used for evaluation on our own benchmarks, whereas the latter versions of the models are used to evaluate on the Neuspell benchmarks. Because the Neuspell benchmarks are based on sentences but our training data consists of paragraphs, we additionally finetune the latter models for one epoch on the sentence-level training datasets from Neuspell.

Dataset/Task	Source	#Samples	#Tokens	$\overline{\text{Sample length}}^*$	$\overline{\overline{\text{Sample length}}}^\dagger$
SEDW/SEC	Wikidump/Bookcorpus	102.3M	6.0B	58	29
TR+	Wikidump/Bookcorpus	102.5M	17.4B	170	95
TR & SEC	Wikidump/Bookcorpus	91,2M	11,5B	126	65
SEDW/SEC	Neuspell	4.0M	143.8M	35	34

* We denote the mean with a single line.
† We denote the median with two lines.

Table 2: Dataset overview: The number of samples and tokens in the datasets built from Wikidump and Bookcorpus deviate mainly due to differences in tokenization and in the way we treat too long sequences. Mean and median sample length are measured in tokens.

Unless stated otherwise we use the following training components and hyperparameters to train all of our models:

- **Optimizer**: We use AdamW (Loshchilov and Hutter, 2019) with a weight decay of 0.01.
- **Learning rate**: We warmup the learning rate linearly from 0 to 10^{-4} within the first 5% of training. At 75% and 90% of training we reduce the learning rate by a factor of 0.1.
- **Batch size**: We use bucket sampling as described below for batching. We set a maximum of 32,768 and 98,304 tokens per batch for SEC models and

probability after every character.

SEDS/SEDW models respectively.

- **Epochs**: We train for 1 epoch (iteration over the training data).
- **Precision**: We use mixed precision (Micikevicius et al., 2018) to save memory and speed up training.

For finetuning models on the Neuspell training dataset we change our training setup in the following way:

- **Learning rate**: We warmup the learning rate linearly from 0 to 10^{-5} within the first 5% of finetuning. Afterwards we linearly decay the learning rate from 10^{-5} to 10^{-7} throughout the rest of finetuning. The learning rate is updated after each optimizer step.
- **Batch size**: We still use bucket sampling as described above for batching, but set a maximum of 4,096 tokens per batch for all models.
- **Epochs**: We finetune for 1 epoch (iteration over the finetuning data).

We train in a distributed fashion with data parallelism. Data parallelism means that every training process, which usually corresponds to one GPU, has the same model parameters but only runs forward passes on a subset of the training data. To make sure that all models stay in sync, gradients are averaged across and communicated to all processes before each process updates its local model parameters.
In general, we choose the number of GPUs for training such that we do not get out of memory errors. Depending on the model and task we train with 8 to 20 NVIDIA GeForce RTX 2080 Ti GPUs. All models except for the TR+ and TR++ models can be trained in less than two days.
Finetuning on the Neuspell dataset is done with a single NVIDIA GeForce RTX 2080 Ti for all models and takes not more than a few hours.

Bucket sampling Instead of splitting our input dataset into batches using a fixed batch size, we generate batches by grouping samples with similar lengths (measured in tokens) into buckets and sample from them without replacement until we hit a

maximum token limit. This is useful to speed up training because it reduces the amount of padding required to batch sequences of different lengths into one tensor before running a forward pass.

4.4 Inference

To improve the runtime and performance of our Seq2Seq spelling error correction models during inference we will use the detection outputs of our SEDW models:

- For SEC models with a shared whitespace-word decoder we discard all whitespace-word contexts for which no spelling error was detected.

  Example

  Given the text *This tetx has an eror!* and spelling error detections $(0, 1, 0, 0, 1)$ we will only correct the second and last whitespace words *tetx* and *eror!*.

- For SEC models with a standard Transformer decoder we only correct the whitespace words for which a spelling error is predicted by stopping once we have decoded a full whitespace word. All parts of the input for which no spelling error is predicted stay as is.

  Example

  Given the text *This tetx has an eror!* and spelling error detections $(0, 1, 0, 0, 1)$ we will first correct starting from the prefix *This* until we decoded a full whitespace-word. Assuming the whitespace-word was correctly decoded to be *text*, we then correct starting from the prefix *This text has an* until we again decode a full whitespace-word or, as in this case, hit the end of the sentence.

Because it limits the number of words for which the SEC models can predict corrections, we expect this procedure to result in fewer false positives (words that are changed by the spelling corrector even though they are not misspelled) but also in

more false negatives (words that are not corrected even though they are misspelled) for our SEC models. In the ideal case the SEDW models will be fast enough, such that the additional time spent on running them is small compared to the time savings we get by skipping unnecessary corrections, and produce good enough detections, such that the performance of our SEC models deteriorates not too much.
Furthermore, during inference we also split sequences that are too long to handle for our models at once using a sliding window approach and recombine the individual results of all windows afterwards.

5 Experiments

This chapter presents the results of our models in comparison with various baselines on spell checking benchmarks.

5.1 Benchmarks

Our own benchmarks are built from the test paragraphs of Wikidump and Bookcorpus and, in case for the realistic benchmarks, our misspellings test set. Because the Neuspell (Jayanthi, Pruthi, and Neubig, 2020) benchmarks come in tokenized form, but our spell checkers work on untokenized texts, we develop a rule-based approach to map them back to regular text with proper whitespacing. We also do not use the *bea322* and *bea4660* benchmarks for the SEDS and SEDW tasks because they only contain lowercase sequences. For SEC we convert the corrections of all models to lowercase before evaluating on them.

SEDW and SEC benchmarks We randomly sample 10,000 sequences from our Wikidump test paragraphs and corrupt them with both of our Artificial and Realistic methods exclusively with $p_{\text{corrupt}} = 0.2$. We do the same with Bookcorpus test paragraphs.
Table 3 gives an overview over the resulting benchmarks.

SEDS benchmarks We adopt the same procedure from above to generate SEDS benchmarks, but we only corrupt half of all sequences and set $p_{\text{corrupt}} = 0.05$ to make

it harder for the models to detect misspelled sequences. An overview over the SEDS benchmarks is shown in Table 4.

Combined benchmarks The combined benchmarks are downsampled versions of the SEC benchmarks to evaluate slow-running models. We randomly sample 200 sequences from each of the four Neuspell SEC benchmarks to build the combined Neuspell benchmark. We do the same for our four benchmarks to obtain the combined Wikibook benchmark.

Whitespace benchmarks We randomly sample 200 correct sequences from the groundtruths of each of our four benchmarks. We then introduce misspellings into the correct sequences with a mixture of the Artificial and Realistic methods. First with $p_{\text{corrupt}} = 0.05$, the result of which we call *low*, and then with $p_{\text{corrupt}} = 0.2$, the result of which we call *high*. For both *low* and *high* we introduce whitespace errors on top, first with $p_{\text{ins}} = 0.025, p_{\text{del}} = 0.1$ and then with $p_{\text{ins}} = 0.1, p_{\text{del}} = 0.4$.[1]
In the end we obtain four benchmarks, *high-high*, *high-low*, *low-high*, *low-low*, where the first part of the name indicates amount of spelling errors and the second part the amount of whitespacing errors.

Runtime benchmark We create a distinct runtime benchmark used to measure model runtimes. We randomly sample 200 text sequences from each of the four Neuspell SEC benchmarks and from each of our own SEC benchmarks to obtain a total of 1600 samples. This amounts to about 231K of text. We also create a whitespace-corrupted version of this benchmark to evaluate the runtimes of models that can deal with whitespacing errors in text.

[1] p_{ins} and p_{del} are the probabilities of inserting a space between two non-space characters and deleting an existing space respectively.

Benchmark	#Sequences	#Words	Sequence length*	Word errors†	Real-word errors‡	Nonword errors‡
bookcorpus artificial	10,000	407,347	193.8	83,124 (20.4%)	19,346 (23.3%)	63,778 (76.7%)
bookcorpus realistic	10,000	407,074	194.3	82,855 (20.4%)	22,868 (27.6%)	59,987 (72.4%)
neuspell bea322	322	5,275	75.9	323 (6.1%)	13 (4.0%)	310 (96.0%)
neuspell bea4660	4,660	136,475	143.4	5,714 (4.2%)	547 (9.6%)	5,167 (90.4%)
neuspell bea60k	63,044	997,600	75.5	70,064 (7.0%)	1,970 (2.8%)	68,094 (97.2%)
neuspell jfleg	1,601	33,414	105.6	2,041 (6.1%)	374 (18.3%)	1,667 (81.7%)
wikidump artificial	10,000	365,829	196.0	73,925 (20.2%)	14,783 (20.0%)	59,142 (80.0%)
wikidump realistic	10,000	365,753	196.2	73,390 (20.1%)	18,760 (25.6%)	54,630 (74.4%)

* Average sequence length measured in characters
† Error percentage with respect to the number of words
‡ Error percentage with respect to the number of word errors

Table 3: Word-level spelling error detection and spelling error correction benchmarks

Benchmark	#Sequences	#Words	Sequence length*	Word errors†	Real-word errors‡	Nonword errors‡	Sequence errors
bookcorpus artificial	10,000	407,126	193.7	11,797 (2.9%)	2,727 (23.1%)	9,070 (76.9%)	4,967 (49.7%)
bookcorpus realistic	10,000	407,083	193.7	11,816 (2.9%)	3,151 (26.7%)	8,665 (73.3%)	4,971 (49.7%)
neuspell bea60k	63,044	997,604	75.6	35,033 (3.5%)	936 (2.7%)	34,097 (97.3%)	31,596 (50.1%)
neuspell jfleg	1,601	33,411	105.7	1,024 (3.1%)	181 (17.7%)	843 (82.3%)	795 (49.7%)
wikidump artificial	10,000	365,755	195.9	11,541 (3.2%)	2,073 (18.0%)	9,468 (82.0%)	5,011 (50.1%)
wikidump realistic	10,000	365,750	195.9	11,584 (3.2%)	2,604 (22.5%)	8,980 (77.5%)	5,014 (50.1%)

* Average sequence length measured in characters
† Error percentage with respect to the number of words
‡ Error percentage with respect to the number of word errors

Table 4: Sequence-level spelling error detection benchmarks

5.2 Baselines

We compare our approaches with other spell checking techniques ranging from classical dictionary based to recent Neural Network based methods. For all spell checking tasks we include a *do nothing* baseline that keeps the input text unchanged. This baseline serves as an indicator for how difficult a benchmark or task is and as a reference point to show relative improvements (or deteriorations) achieved by applying spell checking techniques.

Spelling error correction baselines

Aspell GNU Aspell (Atkinson, 2009) is a widely used spell checking tool for Linux and Windows systems. It works by suggesting a list of possible replacements for misspelled words. This list is ordered by a score that combines phonetic and edit

distance between the input and replacement word. For our baseline we split the input text into words by regex and replace every word that is not a punctuation mark with the top scoring suggestion from Aspell.

Jamspell Jamspell (Ozinov, 2022) provides a faster, modified version of the SymSpell spelling correction algorithm developed by Garbe (2012). To incorporate context it uses a word level n-gram language model to score candidate corrections.

LanguageTool LanguageTool (LanguageTool, 2022) is a company developing proofreading software. They provide a free, open source version of their software[2] on which we base our baseline implementation. To run the LanguageTool server locally we use an open source Docker image[3]. We also use the optional n-gram dataset[4] to be able to correct words depending on their context.

Close to dictionary For this baseline we split the input text into words by regex and replace every word that is not a punctuation mark with the closest word from a dictionary in terms of edit distance. In case of a tie between multiple words we choose the word with the highest frequency in our training data.

Neuspell Bert This is the overall best model from Jayanthi, Pruthi, and Neubig (2020) as described in section 2.2.1. The official implementation[5] works by outputting one corrected token for every input token in a text. Since the model tokenizes its input text using regular expressions and not on whitespaces, we can not obtain a valid output text by joining all output tokens with whitespaces.

Example

Tihs isn't split on whitspaces! will be split into
(Tihs, is, n't, split, on, whitspaces, !) and potentially corrected to

[2]https://github.com/languagetool-org/languagetool
[3]https://github.com/Erikvl87/docker-languagetool
[4]https://storage.googleapis.com/books/ngrams/books/datasetsv2.html
[5]https://github.com/neuspell/neuspell

(This, is, n't, split, on, whitespaces, !). Joining the output with whitespaces gives us *This is n't split on whitespaces !*.

Therefore, we extend the official implementation: We use the fact that we have both access to the original input text and the input tokens as given by the model's tokenizer. If we find the positions at which we must join the input tokens with whitespaces to get the original input text we can join the output tokens in the same way to get the output text with proper whitespacing.

GECToR Bert and GECToR XLNet These are the Bert-based and XLNet-based models proposed by Grammarly in Omelianchuk et al. (2020) as described in section 2.2.2. We include them in our baselines to evaluate how modern state-of-the-art GEC models perform for spelling error correction. We modify the official implementation[6] slightly: Before inference we sort the input texts by their length to achieve faster runtimes. Because our benchmarks contain untokenized text, but GECToR follows the format from Bryant et al. (2019) and expects its inputs to be tokenized with spaCy[7], we tokenize our benchmarks before feeding them to GECToR and recombine its outputs again afterwards.
Since our spelling correction benchmarks consist of word-aligned pairs of misspelled and correct sequences, we evaluate GECToR with only one correction iteration to prevent it from changing the sequences too much and to make it more comparable to all of the other single-round spelling correctors. All other relevant inference parameters are left at their defaults.

NLMSpell NLMSpell is the best model from Hertel (2019) as described in section 2.2.1. We have access to the authors Docker setup for running NLMSpell and use it unchanged.

GPT-3 This baseline uses GPT-3 (Brown et al., 2020) as described in section 2.2.1. At the time of writing there is no open source version of GPT-3 available,

[6] https://github.com/grammarly/gector
[7] https://spacy.io

which is why we base our implementation on the OpenAI Beta API[8]. In particular, we use the best available model for the Edit API, called *text-davinci-edit-001* (there are no official specifications from OpenAI for the models available in their API, but according to EleutherAI *davinci* comes closest to the 175B parameter version of GPT-3[9]). The Edit API itself receives an input and an associated prompt, edits the input according to the prompt with GPT-3 and returns it. We pass misspelled texts as input and set the prompt to *Fix the spelling mistakes.*

Google Google Docs is a free to use online service for writing documents developed by Google (2022). It provides functionality to check a document for spelling and grammatical errors. We use Google Docs as a spelling correction baseline by pasting our benchmarks into a document and applying all suggested spelling corrections until there are none left. We do this both with the option to check for grammatical errors enabled and disabled.

Because of their slow runtimes or manual effort we evaluate the NLMSpell, GPT-3, and Google baselines only on our combined and subsampled spelling correction benchmarks with 800 sequences.

Word-level spelling error detection baselines

For SEDW we make use of the fact that any SEC algorithm can be used to detect spelling errors as described in section 1.2. We reuse the baselines for correcting errors from above and convert their output as follows:

> **Given**: Input text S and corrected text S'
>
> **Goal**: Find label sequence $L = (l_1, \ldots, l_n)$ with $l_i \in \{0, 1\}$ and n being the number of words in S
>
> **Procedure**:

[8] https://beta.openai.com
[9] https://blog.eleuther.ai/gpt3-model-sizes

1. Find the edit operations transforming S into S', restricting possible edit operations on whitespaces to insertions and deletions.
2. Find the word boundaries $((s_1, e_1), \ldots, (s_n, e_n))$ where s_i is the position of the first character and e_i is the position of the last character of the i^{th} word in S.
3. Initialize a set $E = \emptyset$ that will contain the indices of all edited words in S. For every edit operation check whether the edited position p in S falls within the boundaries (s_i, e_i) of an input word, that means $s_i \leq p \leq e_i$. If this is the case we consider the i^{th} word in S to be edited by the spelling corrector: $E \leftarrow E \cup \{i\}$. If p falls exactly between the boundaries of two words, that means $e_i < p < s_{i+1}$, p can only be the position of a whitespace that gets deleted by the spelling corrector. We consider such an operation to affect both the i^{th} and $(i+1)^{\text{th}}$ word: $E \leftarrow E \cup \{i, i+1\}$.
4. Generate the SEDW label sequence $L = (l_1, \ldots, l_n)$:
   Set $l_i = \begin{cases} 1 & \text{if } i \in E \\ 0 & \text{else} \end{cases}$ for $i \in \{1, \ldots, n\}$.

Sequence-level spelling error detection baselines

We again reuse the SEC baselines for SEDS and convert their output as follows:

Given: Input text S and corrected text S'

Goal: Find label $l \in \{0, 1\}$

Procedure:

1. Set $l = \begin{cases} 1 & \text{if } S \neq S' \\ 0 & \text{else} \end{cases}$.

5.3 Evaluation metrics

We now define metrics for each task to evaluate the baselines and our models on benchmarks. A benchmark is given as a sequence of potentially misspelled texts $B = (B_1, \ldots, B_n)$ where n is the number of samples in the benchmark. The groundtruth for a benchmark is given as a sequence $G = (G_1, \ldots, G_n)$ and the predictions of our baselines and models as a sequence $P = (P_1, \ldots, P_n)$. Depending on the task, the elements in G and P are of the same format as described in section 1.2.

Spelling error correction metrics

Correction $\mathbf{F_1}$ For our correction F_1 metric we follow the spelling correction metric from Hertel, 2019. For all $i \in \{1, \ldots, n\}$ we compute four index sets using input text B_i, groundtruth text G_i and corrected text P_i:

misspelled$_i$: This set contains the indices of all words in G_i that are misspelled in B_i.

restored$_i$: This set contains the indices of all words in G_i that are present in P_i.

changed$_i$: This set contains the indices of all words in B_i that are changed in P_i.

correct$_i$: This set contains the indices of all words in B_i that are properly corrected in P_i.

From these four sets we compute true positives $\mathrm{TP}_i = \mathrm{misspelled}_i \cap \mathrm{restored}_i$, false negatives $\mathrm{FN}_i = \mathrm{misspelled}_i \setminus \mathrm{restored}_i$, and false positives $\mathrm{FP}_i = \mathrm{changed}_i \setminus \mathrm{correct}_i$.

Summing them over all samples in the benchmark yields

$$\begin{aligned} \text{TP} &= \sum_{i=1}^{n} TP_i, \\ \text{FP} &= \sum_{i=1}^{n} FP_i, \\ \text{and FN} &= \sum_{i=1}^{n} FN_i, \end{aligned}$$

with which we calculate

$$\begin{aligned} \text{Precision} &= \frac{\text{TP}}{\text{TP} + \text{FP}}, \\ \text{Recall} &= \frac{\text{TP}}{\text{TP} + \text{FN}}, \\ \text{and } \text{F}_1 &= \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}. \end{aligned}$$

Mean normalized edit distance Given two strings a and b, we define the normalized edit distance between them to be

$$\text{ned}(a, b) = \frac{\text{editdistance}(a, b)}{\max\{|a|, |b|\}}.$$

We normalize the regular edit distance metric for two reasons:

1. Normalizing makes samples of different lengths comparable. Being 4 edits away from the groundtruth when the input text is 10 characters long should be worse than being 4 edits away from the groundtruth when the input text is 200 characters long. Here we assume that the number of spelling errors in an input text scales approximately linearly with its length.
2. Since editdistance(a, b) is upper bounded by $\max\{|a|, |b|\}$ and lower bounded by 0, ned(a, b) will always be between 0 and 1. Here 0 means that a and b match exactly and 1 means that they have no characters in common.

The mean normalized edit distance (MNED) over benchmark groundtruths G and

predictions P is then defined as

$$\text{mned} = \frac{1}{n}\sum_{i=1}^{n} \text{ned}(G_i, P_i).$$

Word-level spelling error detection metrics

We define four helper functions

$$\text{eq}(g,p) = \sum_{i=1}^{|g|} \begin{cases} 1 & \text{if } g_i = p_i \\ 0 & \text{else,} \end{cases}$$

$$\text{tp}(g,p) = \sum_{i=1}^{|g|} \begin{cases} 1 & \text{if } g_i = p_i = 1 \\ 0 & \text{else,} \end{cases}$$

$$\text{fp}(g,p) = \sum_{i=1}^{|g|} \begin{cases} 1 & \text{if } p_i = 1 \text{ and } g_i = 0 \\ 0 & \text{else,} \end{cases}$$

$$\text{fn}(g,p) = \sum_{i=1}^{|g|} \begin{cases} 1 & \text{if } p_i = 0 \text{ and } g_i = 1 \\ 0 & \text{else,} \end{cases}$$

that all receive two equally long sequences $g, g_i \in \{0,1\}$ and $p, p_i \in \{0,1\}$ of zeros and ones as input.

Word accuracy Given groundtruths G and predictions P we define word accuracy to be

$$\text{wordacc} = \frac{\sum_{i=1}^{n} \text{eq}(G_i, P_i)}{\sum_{i=1}^{n} |G_i|},$$

that is the fraction of all words for which we make the correct prediction.

Detection $\mathbf{F_1}$ Given groundtruths G and predictions P we define word accuracy

$$\begin{aligned} \text{TP} &= \sum_{i=1}^{n} \text{tp}(G_i, P_i) \\ \text{FP} &= \sum_{i=1}^{n} \text{fp}(G_i, P_i), \\ \text{and FN} &= \sum_{i=1}^{n} \text{fn}(G_i, P_i) \end{aligned}$$

and calculate Precision, Recall and F_1 as above.

Sequence-level spelling error detection metrics

Sequence accuracy For groundtruths G and predictions P sequence accuracy is defined as

$$\text{seqacc} = \frac{\text{eq}(G, P)}{n},$$

that is the fraction of all sequences for which we made the correct prediction.

Detection $\mathbf{F_1}$ Given groundtruths G and predictions P we define

$$\begin{aligned} \text{TP} &= \text{tp}(G, P) \\ \text{FP} &= \text{fp}(G, P) \\ \text{and FN} &= \text{fn}(G, P) \end{aligned}$$

and calculate Precision, Recall and F_1 as above.

5.4 Results

Here we first present the result of our models and the baselines on benchmarks and then their runtimes.

5.4.1 Benchmarks

We first evaluate our models on the SEDS task. Table 5 and 6 show the results on our benchmarks.

We can see that the TR++ model outperforms all other methods both in terms of F_1 score and sequence accuracy, reaching almost 90% sequence accuracy on the Bookcorpus benchmarks. We note that all of our models seem to perform worse on the Wikidump benchmarks with about a 3% absolute decrease in terms of sequence accuracy compared to the Bookcorpus benchmarks.

	bookcorpus artificial			bookcorpus realistic			wikidump artificial			wikidump realistic		
	F_1	Precision	Recall	F_1	Precision	Recall	F_1	Precision	Recall	F_1	Precision	Recall
aspell	79.75	69.84	92.95	79.25	69.63	91.95	74.94	61.19	96.67	74.77	61.12	96.27
jamspell	83.71	79.58	88.28	82.05	79.05	85.29	79.96	74.53	86.25	79.35	74.29	85.16
languagetool	82.60	75.83	90.70	82.12	75.67	89.78	76.58	65.30	92.58	75.63	64.86	90.69
gector bert	68.90	64.29	74.23	70.47	65.09	76.83	56.09	64.63	49.55	58.58	66.00	52.65
gector xlnet	69.12	65.11	73.67	70.74	65.93	76.32	56.10	64.98	49.35	57.92	65.97	51.62
neuspell bert	80.68	77.56	84.05	83.33	78.53	88.75	72.22	73.48	71.00	75.28	74.77	75.79
transformer	89.74	83.92	96.44	89.23	83.80	95.41	85.94	83.19	88.88	85.80	83.18	88.59
gnn	89.54	83.47	96.56	88.69	83.27	94.85	85.97	83.56	88.53	85.69	83.51	87.99
transformer$^+$	89.00	81.98	97.34	88.83	81.96	96.96	86.12	82.68	89.86	86.16	82.72	89.91
gnn$^+$	89.73	83.42	97.06	89.40	83.38	96.36	86.32	83.45	89.38	86.37	83.52	89.43
tokenization repair$^+$	89.44	82.89	97.12	89.42	82.93	97.02	86.34	83.83	89.00	86.69	83.98	89.59
tokenization repair^{++}	**90.38**	84.40	97.26	**90.36**	84.44	97.16	**86.72**	84.27	89.30	**86.79**	84.36	89.37

Table 5: Sequence-level spelling error detection: F_1

Table 7 and 8 show the results on the Neuspell benchmarks. Here both Transformer$^+$ and GNN$^+$ perform well on the BEA60k benchmark mainly due to their higher recall compared to the regular Transformer and GNN. However, on the JFLEG benchmark Jamspell and the simple out-of-dictionary baseline perform best.

The benchmark results for the SEDW task can be seen in Tables 9, 10, 11 and 12. In the word accuracy tables, in addition to the word accuracy itself, we show the detection rates of the models on real-word and nonword errors as percentages and the absolute number of real-word and nonword errors for each benchmark. On our benchmarks TR++ again achieves the best results. On the Neuspell BEA60k benchmark our GNN$^+$ model is at the top, whereas on the JFLEG benchmark

	bookcorpus artificial	bookcorpus realistic	wikidump artificial	wikidump realistic
do nothing	50.33	50.29	49.89	49.86
out of dictionary	78.66	78.39	71.18	70.97
aspell	76.56	76.06	67.61	67.43
jamspell	82.93	81.45	78.34	77.78
languagetool	81.02	80.57	71.63	70.69
gector bert	66.72	68.00	61.13	62.66
gector xlnet	67.31	68.62	61.29	62.39
neuspell bert	80.00	82.35	72.63	75.04
transformer	89.05	88.55	85.43	85.30
gnn	88.79	87.97	85.52	85.27
transformer^{+}	88.05	87.88	85.49	85.52
gnn^{+}	88.96	88.64	85.80	85.85
tokenization repair^{+}	88.61	88.59	85.89	86.21
tokenization repair^{++}	**89.71**	**89.69**	**86.29**	**86.36**

Table 6: Sequence-level spelling error detection: Sequence accuracy

Neuspell BERT and Jamspell perform best. Overall, these benchmarks deliver a pretty similar picture than the SEDS benchmarks.

For the MNED metric on our SEC benchmarks we show relative improvements over a *do nothing* baseline as well as the absolute values. The MNED of *do nothing* gives us the distance between the benchmark inputs and groundtruths.
We also show for each SEC benchmark the pipelines of our best SEDW method on the corresponding benchmark with both the Transformer and Transformer word models.

The results for the default SEC benchmarks can be seen in Tables 14, 13, 16 and 15. They show that, according to the F_1 metric, both for our benchmarks and the Neuspell benchmarks three out of four times a pipeline consisting of a SEDW and a SEC model performs best. We see this as the confirmation that SEDW methods indeed help in improving SEC models.

The benchmark results on the combined benchmarks are shown in Table 18 and 17.

	neuspell bea60k			neuspell jfleg		
	F_1	Precision	Recall	F_1	Precision	Recall
out of dictionary	90.18	83.57	97.93	90.71	93.11	88.43
aspell	89.29	81.76	98.34	89.35	91.22	87.55
jamspell	90.81	90.21	91.43	**90.97**	96.34	86.16
languagetool	89.79	83.45	97.17	90.29	91.70	88.93
gector bert	75.49	66.71	86.92	71.89	61.81	85.91
gector xlnet	75.36	66.37	87.17	71.95	62.63	84.53
neuspell bert	89.69	84.79	95.20	88.97	85.42	92.83
transformer	90.21	88.59	91.89	89.32	89.72	88.93
gnn	90.88	87.92	94.04	90.14	88.88	91.45
transformer$^+$	91.65	87.37	96.37	88.92	86.63	91.32
gnn$^+$	**91.85**	87.80	96.28	89.66	87.26	92.20

Table 7: Sequence-level spelling error detection Neuspell: F_1

On the combined Neuspell benchmarks GPT-3 performs best by far, followed by our methods. However, this is no surprise because one half of this benchmark consists of ambiguous sequences where the immense language modeling capabilities by GPT-3 are extremely helpful.

We also show that decoding with beam search rather than greedy search for our models might not be worth it as it only leads to marginal improvements most of the time.

Our final set of benchmarks are SEC benchmarks with whitespacing errors shown in Table 19 and 20. The best model here is the Transformer with tokenization repair achieving the highest scores on three out four benchmarks. The results of the other models and pipelines perform very similar on all of the benchmarks. The *eo medium* model in these tables is the medium-sized tokenization repair model from Walter (2021).

	neuspell bea60k	neuspell jfleg
do nothing	49.88	50.34
out of dictionary	89.31	91.01
aspell	88.17	89.63
jamspell	90.73	**91.51**
languagetool	88.93	90.51
gector bert	71.71	66.65
gector xlnet	71.43	67.27
neuspell bert	89.04	88.57
transformer	90.01	89.44
gnn	90.54	90.07
transformer$^+$	91.20	88.69
gnn$^+$	**91.43**	89.44

Table 8: Sequence-level spelling error detection Neuspell: Sequence accuracy

5.4.2 Runtimes

Table 21 shows the runtimes of our models on the runtime benchmarks. As a first observation we can see that no method is so slow that it is impossible to use it for practical purposes. The slowest model overall is the Transformer with tokenization repair, closely followed by the regular Transformer encoder-decoder model, achieving still 2.8kB/s and 3.2kB/s of throughput respectively.

Since the Transformer and GNN models for SEDS/SEDW can predict spelling errors in parallel, they are much faster in terms of absolute running times than the Seq2Seq models for SEC. The models that are enriched with word features are also only marginally slower than those without. However, the Transformer models are still nearly twice as fast as their graph-based counterparts. We believe the slower runtimes come mainly from two issues: First, for a GNN we have to explicitly create an input graph before we can run the model, which can take up to a few milliseconds. And second, the Transformer encoder layers can be very efficiently implemented in practice using e.g. only a single matrix multiplication to calculate the attention scores between

	bookcorpus artificial			bookcorpus realistic			wikidump artificial			wikidump realistic		
	F_1	Precision	Recall	F_1	Precision	Recall	F_1	Precision	Recall	F_1	Precision	Recall
aspell	86.45	90.22	82.99	81.90	86.87	77.47	82.40	79.33	85.71	76.41	73.29	79.81
jamspell	83.70	95.62	74.42	76.71	93.85	64.86	84.29	93.30	76.86	76.92	90.33	66.98
languagetool	86.58	91.83	81.90	83.15	89.60	77.56	84.38	84.94	83.83	79.76	80.60	78.94
gector bert	52.43	73.85	40.64	60.02	74.70	50.16	50.18	77.23	37.16	60.47	78.55	49.16
gector xlnet	58.01	76.73	46.63	63.38	77.17	53.77	50.18	79.34	36.70	59.78	80.85	47.42
neuspell bert	76.36	94.40	64.11	91.50	94.95	88.30	71.84	92.73	58.63	89.49	93.74	85.61
transformer	96.46	97.52	95.43	92.56	97.04	88.47	95.08	97.39	92.88	93.22	96.94	89.76
gnn	96.51	97.51	95.54	92.57	96.98	88.54	95.30	97.44	93.26	93.36	96.95	90.04
transformer$^+$	96.62	97.01	96.22	95.36	96.56	94.20	95.52	97.20	93.90	95.61	96.81	94.45
gnn$^+$	96.62	97.41	95.84	95.02	97.01	93.10	95.44	97.48	93.49	95.33	97.12	93.60
tokenization repair$^+$	96.80	97.37	96.24	95.69	97.05	94.38	95.65	97.57	93.80	95.98	97.22	94.76
tokenization repair^{++}	**97.04**	97.64	96.46	**96.15**	97.26	95.07	**95.80**	97.67	94.00	**96.27**	97.37	95.19

Table 9: Word-level spelling error detection: F_1

	bookcorpus artificial			bookcorpus realistic			wikidump artificial			wikidump realistic		
	Accuracy	Real-word 19,346	Nonword 63,778	Accuracy	Real-word 22,868	Nonword 59,987	Accuracy	Real-word 14,783	Nonword 59,142	Accuracy	Real-word 18,760	Nonword 54,630
do nothing	76.31	0.0	0.0	76.39	0.0	0.0	76.78	0.0	0.0	76.95	0.0	0.0
out of dictionary	92.32	0.0	96.4	91.41	0.0	99.8	91.51	0.0	97.0	89.61	0.0	99.3
aspell	93.84	32.4	98.3	91.92	35.2	93.6	91.50	36.1	98.1	88.64	37.8	94.2
jamspell	93.13	42.8	84.0	90.70	29.1	78.5	93.35	45.7	84.7	90.74	27.7	80.4
languagetool	93.99	35.0	96.1	92.58	38.7	92.4	92.79	37.9	95.3	90.76	41.1	91.9
gector bert	82.53	48.6	38.2	84.22	54.5	48.5	82.87	56.3	32.4	85.19	58.1	46.1
gector xlnet	84.01	54.8	44.1	85.33	58.6	51.9	83.08	59.2	31.1	85.29	58.7	43.6
neuspell bert	90.60	70.4	62.2	96.13	71.8	94.6	89.33	72.4	55.2	95.37	69.5	91.2
transformer	98.34	91.7	96.6	96.64	76.9	92.9	97.77	91.4	93.2	96.99	78.7	93.6
gnn	98.37	91.7	96.7	96.64	76.2	93.2	97.87	92.0	93.6	97.05	78.3	94.1
transformer$^+$	98.40	92.8	97.3	97.84	80.5	99.4	97.96	92.5	94.3	98.00	81.4	98.9
gnn$^+$	98.41	92.2	97.0	97.69	77.8	98.9	97.93	92.2	93.8	97.89	79.3	98.5
tokenization repair$^+$	98.49	93.1	97.2	97.99	81.1	99.4	98.02	92.9	94.0	98.17	82.2	99.1
tokenization repair^{++}	**98.61**	94.1	97.2	**98.20**	83.8	99.4	**98.08**	93.7	94.1	**98.30**	84.0	99.0

Table 10: Word-level spelling error detection: Word accuracy

all tokens in a sequence, which is not the case for the GNN.

For SEC our fastest single model is Transformer word. We can see that using any of our SEDW models before the SEC models to filter for misspelled words gives a significant boost in inference speed. Our fastest pipeline, Transformer and Transformer word, even runs faster than the BERT model by Neuspell which is an encoder-only model.

	neuspell bea60k			neuspell jfleg		
	F_1	Precision	Recall	F_1	Precision	Recall
out of dictionary	91.61	86.70	97.11	87.48	94.23	81.63
aspell	90.47	84.14	97.83	86.60	91.73	82.02
jamspell	90.68	91.73	89.65	88.36	96.85	81.23
languagetool	88.55	81.89	96.39	85.34	86.28	84.42
gector bert	67.08	58.41	78.78	57.74	49.53	69.23
gector xlnet	66.76	57.89	78.85	57.04	49.15	67.96
neuspell bert	88.65	84.23	93.56	**88.37**	86.88	89.91
transformer	90.02	89.81	90.22	86.97	89.91	84.22
gnn	90.94	89.26	92.69	87.92	89.79	86.13
transformer$^+$	91.88	88.69	95.31	87.40	87.83	86.97
gnn$^+$	**92.02**	88.86	95.42	88.23	88.91	87.56

Table 11: Word-level spelling error detection Neuspell: F_1

	neuspell bea60k			neuspell jfleg		
	Accuracy	Real-word	Nonword	Accuracy	Real-word	Nonword
		1,970	68,094		374	1,667
do nothing	92.02	0.0	0.0	93.23	0.0	0.0
out of dictionary	98.58	0.0	99.9	98.42	0.3	99.9
aspell	98.36	71.8	98.6	98.28	8.8	98.4
jamspell	98.53	41.4	91.0	**98.55**	18.2	95.4
languagetool	98.01	64.7	97.3	98.03	23.8	98.0
gector bert	93.83	60.9	79.3	93.14	53.2	72.8
gector xlnet	93.74	59.5	79.4	93.07	52.9	71.3
neuspell bert	98.09	60.4	94.5	98.40	56.4	97.4
transformer	98.40	50.4	91.4	98.29	38.0	94.6
gnn	98.53	51.4	93.9	98.40	36.1	97.4
transformer$^+$	98.66	46.5	96.7	98.30	38.2	97.9
gnn$^+$	**98.68**	47.6	96.8	98.42	40.4	98.1

Table 12: Word-level spelling error detection Neuspell: Word accuracy

	bookcorpus artificial		bookcorpus realistic		wikidump artificial		wikidump realistic	
	Improvement	MNED	Improvement	MNED	Improvement	MNED	Improvement	MNED
do nothing	-	0.0671	-	0.0693	-	0.0561	-	0.0550
close to dictionary	-18.8%	0.0545	-7.1%	0.0643	+24.8%	0.0700	+41.3%	0.0777
aspell	+13.3%	0.0760	+43.2%	0.0992	+81.6%	0.1018	+113.2%	0.1172
jamspell	-29.4%	0.0474	-18.1%	0.0568	-13.7%	0.0484	-3.6%	0.0530
languagetool	-11.5%	0.0594	+6.9%	0.0741	+13.4%	0.0636	+42.9%	0.0785
gector bert	+15.6%	0.0776	+17.9%	0.0817	+7.4%	0.0602	+8.3%	0.0595
gector xlnet	+10.3%	0.0740	+11.7%	0.0774	+2.0%	0.0572	+2.9%	0.0566
neuspell bert	-29.1%	0.0476	-34.2%	0.0456	-16.5%	0.0468	-19.1%	0.0445
transformer	-75.2%	0.0167	-51.4%	0.0336	-67.9%	0.0180	-56.3%	0.0240
transformer word	-75.6%	0.0163	-49.5%	0.0350	-69.8%	0.0169	-56.5%	0.0239
transformer^{+} → transformer	-75.6%	0.0164	**-52.4%**	0.0330	-68.8%	0.0175	-58.9%	0.0226
transformer^{+} → transformer word	-76.2%	0.0160	-50.6%	0.0342	-70.4%	0.0166	-59.2%	0.0224
tokenization repair^{++}	**-77.2%**	0.0153	-52.1%	0.0332	**-70.8%**	0.0164	**-60.3%**	0.0218

Table 13: Spelling error correction: Mean normalized edit distance

	bookcorpus artificial			bookcorpus realistic			wikidump artificial			wikidump realistic		
	F_1	Precision	Recall	F_1	Precision	Recall	F_1	Precision	Recall	F_1	Precision	Recall
aspell	35.96	37.53	34.53	19.24	20.41	18.20	36.71	35.34	38.18	20.49	19.65	21.41
jamspell	47.70	54.48	42.41	48.03	58.76	40.61	49.52	54.81	45.16	46.57	54.69	40.55
languagetool	46.09	48.87	43.61	38.22	41.07	35.74	48.81	49.14	48.49	38.15	38.49	37.81
gector bert	27.06	40.45	20.33	27.91	37.39	22.26	29.18	47.79	21.00	30.10	42.57	23.28
gector xlnet	34.65	49.46	26.67	33.71	44.60	27.10	35.12	59.03	25.00	35.97	52.52	27.35
neuspell bert	56.79	70.21	47.69	61.82	64.15	59.66	56.01	72.29	45.71	57.67	60.41	55.17
transformer	87.47	88.04	86.92	72.66	74.00	71.37	87.97	89.27	86.70	74.87	76.02	73.75
transformer word	87.48	87.91	87.05	71.00	72.29	69.76	**88.26**	89.26	87.29	73.59	74.50	72.70
transformer^{+} → transformer	**87.64**	89.50	85.86	**72.69**	75.45	70.13	87.91	91.16	84.88	**75.10**	77.89	72.51
transformer^{+} → transformer word	87.62	89.32	85.99	71.10	73.60	68.77	88.15	91.17	85.31	73.82	76.28	71.51
tokenization repair^{++}	87.47	88.90	86.09	71.06	73.05	69.17	87.93	90.70	85.32	73.48	75.43	71.64

Table 14: Spelling error correction: Correction F_1

	neuspell bea322		neuspell bea4660		neuspell bea60k		neuspell jfleg	
	Improvement	MNED	Improvement	MNED	Improvement	MNED	Improvement	MNED
do nothing	-	0.0232	-	0.0118	-	0.0293	-	0.0156
close to dictionary	+33.9%	0.0311	+14.1%	0.0134	-23.9%	0.0223	-49.4%	0.0079
aspell	+44.6%	0.0336	+64.7%	0.0194	-9.6%	0.0265	-25.8%	0.0116
jamspell	-18.7%	0.0189	-48.9%	0.0060	-44.6%	0.0162	-63.3%	0.0057
languagetool	+17.4%	0.0273	+45.3%	0.0171	-21.0%	0.0231	-35.0%	0.0101
gector bert	+8.8%	0.0253	+13.0%	0.0133	-11.7%	0.0258	+90.6%	0.0297
gector xlnet	+4.2%	0.0242	+3.9%	0.0122	-9.6%	0.0264	+77.3%	0.0276
neuspell bert	-3.9%	0.0223	-61.6%	0.0045	-32.4%	0.0198	-55.0%	0.0070
transformer	-22.9%	0.0179	-62.1%	0.0045	-52.8%	0.0138	**-69.2%**	0.0048
transformer word	-33.7%	0.0154	-65.8%	0.0040	-53.4%	0.0136	-68.0%	0.0050
gnn$^+$ → transformer	-31.5%	0.0159	-65.1%	0.0041	-56.6%	0.0127	-68.3%	0.0049
gnn$^+$ → transformer word	**-36.1%**	0.0149	**-68.6%**	0.0037	**-57.6%**	0.0124	-68.1%	0.0050

Table 15: Spelling error correction Neuspell: Mean normalized edit distance

	neuspell bea322			neuspell bea4660			neuspell bea60k			neuspell jfleg		
	F_1	Precision	Recall	F_1	Precision	Recall	F_1	Precision	Recall	F_1	Precision	Recall
aspell	27.23	25.13	29.72	15.82	15.02	16.70	55.76	51.86	60.30	66.91	70.83	63.40
jamspell	53.74	55.23	52.32	71.70	70.53	72.91	69.87	70.67	69.08	81.54	89.32	75.01
languagetool	28.96	28.24	29.72	25.33	23.27	27.77	60.30	55.95	65.37	71.31	72.97	69.72
gector bert	55.23	48.70	63.78	67.30	55.46	85.56	60.44	53.62	69.23	48.00	42.72	54.78
gector xlnet	55.49	49.88	62.54	70.34	58.78	87.54	60.31	53.33	69.39	48.29	42.97	55.12
neuspell bert	65.36	59.54	72.45	85.14	78.77	92.61	74.73	71.00	78.87	83.03	81.60	84.52
transformer	61.94	58.15	66.25	82.32	77.79	87.42	75.37	73.98	76.82	**85.92**	86.80	85.06
transformer word	66.86	63.26	70.90	84.84	80.44	89.74	74.85	72.30	77.58	85.27	85.88	84.66
gnn$^+$ → transformer	64.25	62.65	65.94	83.41	79.96	87.17	**77.43**	78.74	76.16	85.48	88.55	82.61
gnn$^+$ → transformer word	**68.48**	66.76	70.28	**85.75**	82.27	89.55	77.43	78.09	76.78	85.18	88.03	82.51

Table 16: Spelling error correction Neuspell: Correction F_1

	combined neuspell		combined wikibook	
	Improvement	MNED	Improvement	MNED
do nothing	-	0.0194	-	0.0613
gpt3	**-69.5%**	0.0059	+82.3%	0.1118
nlmspell	-38.8%	0.0119	-56.2%	0.0268
google	-5.7%	0.0183	-31.4%	0.0421
$\text{google}_{\text{w/o grammar}}$	-5.8%	0.0183	-31.4%	0.0421
transformer	-47.8%	0.0101	-63.0%	0.0227
$\text{transformer}_{\text{beam}}$	-48.2%	0.0101	**-64.8%**	0.0216
transformer word	-53.1%	0.0091	-62.6%	0.0229
transformer $\text{word}_{\text{beam}}$	-53.3%	0.0091	-62.4%	0.0230
transformer $\rightarrow$ transformer	-50.4%	0.0096	-63.0%	0.0227
gnn $\rightarrow$ transformer	-49.8%	0.0097	-63.4%	0.0224
$\text{transformer}^{+} \rightarrow$ transformer	-51.7%	0.0094	-64.6%	0.0217
$\text{gnn}^{+} \rightarrow$ transformer	-52.1%	0.0093	-62.5%	0.0230
transformer $\rightarrow$ transformer word	-54.3%	0.0089	-62.8%	0.0228
gnn $\rightarrow$ transformer word	-57.2%	0.0083	-63.1%	0.0226
$\text{transformer}^{+} \rightarrow$ transformer word	-56.0%	0.0085	-64.5%	0.0218
$\text{gnn}^{+} \rightarrow$ transformer word	-56.4%	0.0085	-62.6%	0.0229
$\text{tokenization repair}^{++}$	-	-	-60.3%	0.0243
$\text{tokenization repair}^{++}_{\text{beam}}$	-	-	-63.8%	0.0222

Table 17: Spelling error correction Combined: Mean normalized edit distance

	combined neuspell			combined wikibook		
	F_1	Precision	Recall	F_1	Precision	Recall
gpt3	**89.50**	89.40	89.60	74.13	88.87	63.59
nlmspell	72.68	68.39	77.54	78.86	80.38	77.39
google	71.52	64.04	80.97	58.31	73.01	48.54
$\text{google}_{\text{w/o grammar}}$	71.61	64.05	81.19	58.74	72.78	49.24
transformer	77.03	74.01	80.31	79.64	80.54	78.76
$\text{transformer}_{\text{beam}}$	77.07	74.08	80.31	**80.28**	81.21	79.37
transformer word	78.92	75.71	82.41	79.21	79.94	78.50
$\text{transformer word}_{\text{beam}}$	78.88	75.63	82.41	79.25	79.99	78.53
transformer $\rightarrow$ transformer	77.83	77.79	77.88	78.59	82.73	74.85
gnn $\rightarrow$ transformer	78.27	77.67	78.87	78.64	82.80	74.88
$\text{transformer}^+ \rightarrow$ transformer	78.42	77.66	79.20	80.02	82.88	77.35
$\text{gnn}^+ \rightarrow$ transformer	78.73	77.84	79.65	79.11	81.57	76.79
transformer $\rightarrow$ transformer word	80.22	80.58	79.87	77.97	81.74	74.53
gnn $\rightarrow$ transformer word	80.97	80.53	81.42	78.08	81.78	74.71
$\text{transformer}^+ \rightarrow$ transformer word	80.64	79.98	81.31	79.39	81.89	77.04
$\text{gnn}^+ \rightarrow$ transformer word	81.10	80.35	81.86	78.43	80.45	76.51
$\text{tokenization repair}^{++}$	-	-	-	79.11	81.27	77.07
$\text{tokenization repair}^{++}_{\text{beam}}$	-	-	-	79.37	81.54	77.32

Table 18: Spelling error correction Combined: Correction F_1

	whitespace high-high		whitespace high-low		whitespace low-high		whitespace low-low	
	Improvement	MNED	Improvement	MNED	Improvement	MNED	Improvement	MNED
do nothing	-	0.1885	-	0.0900	-	0.1496	-	0.0464
transformer with tokenization repair	-84.0%	0.0301	-69.2%	0.0277	-91.8%	0.0123	-77.8%	0.0103
$\text{transformer with tokenization repair}_{\text{beam}}$	**-84.7%**	0.0288	**-70.9%**	0.0262	**-92.2%**	0.0116	**-79.0%**	0.0098
$\text{tokenization repair}^{++}$	-78.9%	0.0399	-66.9%	0.0297	-91.1%	0.0133	-78.2%	0.0101
$\text{tokenization repair}^{++}_{\text{beam}}$	-79.1%	0.0395	-67.4%	0.0293	-91.1%	0.0133	-78.5%	0.0100
eo medium $\rightarrow \text{transformer}^+ \rightarrow$ transformer	-77.9%	0.0416	-65.7%	0.0309	-90.8%	0.0137	-76.7%	0.0108
eo medium $\rightarrow \text{transformer}^+ \rightarrow$ transformer word	-78.1%	0.0412	-65.7%	0.0308	-91.3%	0.0130	-77.9%	0.0103
$\text{tokenization repair}^+ \rightarrow$ transformer	-77.5%	0.0424	-64.4%	0.0320	-89.8%	0.0153	-73.7%	0.0122
$\text{tokenization repair}^+ \rightarrow$ transformer word	-77.6%	0.0422	-64.3%	0.0321	-90.4%	0.0143	-75.2%	0.0115

Table 19: Spelling error correction Whitespace: Mean normalized edit distance

	whitespace high-high			whitespace high-low			whitespace low-high			whitespace low-low		
	F_1	Precision	Recall	F_1	Precision	Recall	F_1	Precision	Recall	F_1	Precision	Recall
$\text{transformer with tokenization repair}_{\text{beam}}$	**89.28**	87.54	91.10	**85.93**	84.85	87.04	**94.44**	93.27	95.63	91.77	89.64	94.00
$\text{tokenization repair}^{++}$	85.90	84.14	87.73	83.64	83.34	83.95	94.08	93.10	95.09	92.81	91.92	93.71
$\text{tokenization repair}^{++}_{\text{beam}}$	86.02	84.29	87.83	83.85	83.56	84.14	94.08	93.09	95.09	**92.88**	91.99	93.80
eo medium $\rightarrow \text{transformer}^+ \rightarrow$ transformer	85.71	83.84	87.66	83.98	83.88	84.09	94.03	93.05	95.02	92.67	91.72	93.63
eo medium $\rightarrow \text{transformer}^+ \rightarrow$ transformer word	85.70	83.89	87.58	83.84	83.62	84.06	93.96	92.95	95.00	92.63	91.69	93.59
$\text{tokenization repair}^+ \rightarrow$ transformer	85.89	84.09	87.77	84.22	84.21	84.23	94.06	93.10	95.05	92.74	91.94	93.55
$\text{tokenization repair}^+ \rightarrow$ transformer word	85.85	84.09	87.69	84.05	83.90	84.19	94.02	93.04	95.03	92.73	91.95	93.53

Table 20: Spelling error correction Whitespace: Correction F_1

Task	Model/Pipeline	Runtime in s	kB/s
TR*	eo large	7.9	29.9
TR*	eo medium	6.1	38.5
TR*	eo small	5.8	40.4
SEDS/SEDW†	tokenization repair^{+}/tokenization repair^{++}	13.1	18.0
SEDS/SEDW†	gnn^{+}	9.7	24.3
SEDS/SEDW†	gnn	9.2	25.7
SEDS/SEDW†	transformer^{+}	5.6	42.3
SEDS/SEDW†	transformer	4.8	49.5
SEC	transformer	73.5	3.2
SEDW → SEC	transformer^{+} → transformer	47.2	5.0
SEDW → SEC	gnn^{+} → transformer	46.4	5.1
SEC	transformer word	37.2	6.4
SEC	neuspell bert	21.8	10.8
SEDW → SEC	gnn^{+} → transformer word	13.7	17.3
SEDW → SEC	transformer^{+} → transformer word	13.5	17.5
TR & SEC	transformer with tokenization repair	83.1	2.8
TR → SEDW → SEC	eo medium → gnn^{+} → transformer	60.4	3.9
TR → SEDW → SEC	tokenization repair^{+} → transformer	57.6	4.1
TR → SEDW → SEC	eo medium → transformer^{+} → transformer	56.2	4.2
TR → SEDW → SEC	eo medium → gnn^{+} → transformer word	28.0	8.4
TR → SEDW → SEC	tokenization repair^{+} → transformer word	24.7	9.5
TR → SEDW → SEC	tokenization repair^{++}	24.4	9.7
TR → SEDW → SEC	eo medium → transformer^{+} → transformer word	23.6	10.0

* Ported models from https://github.com/ad-freiburg/trt, shown here for reference

† The overhead of converting word level detections into sequence level detections is negligible

Table 21: Runtimes: Model runtimes on our runtime benchmark (see section 5.1). The pipelines for tasks starting with TR are run on the whitespace-corrupted version of our runtime benchmark. For all models we sort the benchmark inputs by length before inference and reorder the outputs afterwards. We measure the runtimes on a system equipped with a NVIDIA GeForce GTX 1080 Ti GPU and an Intel Core i7-9750H CPU.

6 Conclusion and future work

Deep learning models dominate classical methods Our evaluations show that deep learning models outperform classical dictionary and edit distance based methods, as well as methods using n-gram language models on virtually all benchmarks, often by large margins.

GNNs are competitive to Transformers for spelling error detection Representing text as a word graph and processing it with a GNN can give similar and in some cases better results compared to Transformer models. However, Transformer models are still faster.

Adding word features helps spelling error detection In our experiments we find that spelling error detection models with access to word features consistently outperform their respective counterparts without word features on most of the benchmarks. The word features especially seem to help in achieving higher detection rates.

Spelling error detection can improve spelling error correction in runtime and performance In our experiments running a spelling error detection model before spelling error correction to filter out correctly spelled words always led to faster inference and most of the time to better performance judging by our metrics.

Sequence-to-sequence approaches to spelling error correction can perform well Previous work by Hertel (2019) found that formulating spelling error correction

as a machine translation task can be hard because the model e.g. has to deal with many ambiguities when translating from misspelled to correct text. However, our models for spelling error correction which are exclusively Seq2Seq models show, that translation models are able to reach high levels of performance when trained carefully on a large dataset.

We think there are a lot of interesting aspects and open questions that could be subject in future work:

- **Make more word features or information available to the models**: We presented 13 word features that we have shown to improve the performance of spelling error detection models. Adding either more features or information from other sources could help improving them further.
- **Data generation techniques**: We generated pairs of misspelled and correct sequences for training using two relatively simple methods: Applying random character transformations and replacing correct words with misspellings from confusion sets. An interesting direction of future work could be to compare multiple different data generation techniques and determine the ones that work better than ours.
- **Word graph structure**: Our GNN-based spelling error detection models show equal if not better performance compared to regular Transformer-encoders. Finding out to what part this is based on the explicit modeling of natural language text as graph itself and finding graph structures that might perform even better is a task left for future work.
- **Full end-to-end training of TR+ models**: We have shown that we can use pretrained tokenization repair models for fixed feature extraction and successfully detect spelling errors on top of them. It would be of interest whether we can also train the whole model in an end-to-end fashion and whether this benefits performance.
- **Language model pretraining**: We have seen some examples of ambiguous

and hard to detect spelling errors where methods that use large pretrained language models like BERT, XLNet or GPT-3 tend to perform better than our models. Pretraining our models with a language modeling task and checking whether this improves the capabilities of our models when dealing with such errors is an interesting task for future work.

- **Scale**: Our models mainly reside in the small to medium-sized regime when compared to other modern deep learning architectures. Although we think keeping models as small as possible is desirable for fast inference speed it would be of interest to see how the performance of our models would improve by scaling them up in size.
- **Multilingual models**: One of the most interesting aspects from our perspective is the possibility of generalizing our methods to work on multiple languages with a single model. Since our spelling correction methods all use open vocabularies it should be easy to get them to run on multilingual data by training multilingual tokenizers. For our spelling detection models one would need to either design features that work across languages (e.g. multilingual dictionaries) or drop them entirely and use the Transformer and GNN models without features. The tokenization-repair-based models should be portable to multiple languages as well, as long as these languages split their word boundaries with whitespaces.

Bibliography

Ahmadi, Sina (Sept. 21, 2018). *Attention-Based Encoder-Decoder Networks for Spelling and Grammatical Error Correction*. arXiv: 1810.00660 [cs].

Atkinson, Kevin (2009). *GNU Aspell*. URL: http://aspell.net/ (visited on 04/12/2022).

Awasthi, Abhijeet et al. (Nov. 2019). "Parallel Iterative Edit Models for Local Sequence Transduction". In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. EMNLP-IJCNLP 2019. Hong Kong, China: Association for Computational Linguistics, pp. 4260–4270. DOI: 10.18653/v1/D19-1435.

Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E. Hinton (July 21, 2016). "Layer Normalization".

Bast, Hannah, Matthias Hertel, and Mostafa M. Mohamed (Oct. 15, 2020). "Tokenization Repair in the Presence of Spelling Errors".

Belinkov, Yonatan and Yonatan Bisk (Feb. 24, 2018). "Synthetic and Natural Noise Both Break Neural Machine Translation".

Boytsov, Leonid (May 2011). "Indexing Methods for Approximate Dictionary Searching: Comparative Analysis". In: *ACM Journal of Experimental Algorithmics* 16. ISSN: 1084-6654, 1084-6654. DOI: 10.1145/1963190.1963191.

Bronstein, Michael M. et al. (May 2, 2021). "Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges".

Brown, Tom B. et al. (July 22, 2020). "Language Models Are Few-Shot Learners".

Bryant, Christopher et al. (Aug. 2019). "The BEA-2019 Shared Task on Grammatical Error Correction". In: *Proceedings of the Fourteenth Workshop on Innovative Use of NLP for Building Educational Applications*. Florence, Italy: Association for Computational Linguistics, pp. 52–75. DOI: 10.18653/v1/W19-4406.

Chelba, Ciprian et al. (Mar. 4, 2014). "One Billion Word Benchmark for Measuring Progress in Statistical Language Modeling".

Chollampatt, Shamil and Hwee Tou Ng (Jan. 26, 2018). "A Multilayer Convolutional Encoder-Decoder Neural Network for Grammatical Error Correction".

Chowdhery, Aakanksha et al. (Apr. 19, 2022). "PaLM: Scaling Language Modeling with Pathways".

Damerau, Fred J. (Mar. 1964). "A Technique for Computer Detection and Correction of Spelling Errors". In: *Communications of the ACM* 7.3, pp. 171–176. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/363958.363994.

Devlin, Jacob et al. (May 24, 2019). "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding".

Gao, Ji et al. (May 23, 2018). "Black-Box Generation of Adversarial Text Sequences to Evade Deep Learning Classifiers".

Gao, Mengyi, Canran Xu, and Peng Shi (Sept. 29, 2021). "Hierarchical Character Tagger for Short Text Spelling Error Correction".

Garbe, Wolf (June 2012). *SymSpell*.

Ge, Tao, Furu Wei, and Ming Zhou (July 2018). "Fluency Boost Learning and Inference for Neural Grammatical Error Correction". In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. ACL 2018. Melbourne, Australia: Association for Computational Linguistics, pp. 1055–1065. DOI: 10.18653/v1/P18-1097.

Golding, Andrew R. and Dan Roth (Oct. 31, 1998). "A Winnow-Based Approach to Context-Sensitive Spelling Correction".

Google (2022). *Google Docs*. URL: https://docs.google.com (visited on 05/17/2022).

Grundkiewicz, Roman and Marcin Junczys-Dowmunt (June 2018). "Near Human-Level Performance in Grammatical Error Correction with Hybrid Machine Translation". In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*. NAACL-HLT 2018. New Orleans, Louisiana: Association for Computational Linguistics, pp. 284–290. DOI: 10.18653/v1/N18-2046.

Grundkiewicz, Roman, Marcin Junczys-Dowmunt, and Kenneth Heafield (Aug. 2019). "Neural Grammatical Error Correction Systems with Unsupervised Pre-training on Synthetic Data". In: *Proceedings of the Fourteenth Workshop on Innovative Use of NLP for Building Educational Applications*. Florence, Italy: Association for Computational Linguistics, pp. 252–263. DOI: 10.18653/v1/W19-4427.

Hamilton, William L. (2020). "Graph Representation Learning". In: *Synthesis Lectures on Artificial Intelligence and Machine Learning* 14.3, pp. 1–159.

Hendrycks, Dan and Kevin Gimpel (July 8, 2020). "Gaussian Error Linear Units (GELUs)".

Hertel, Matthias (Dec. 6, 2019). *Neural Language Models for Spelling Correction*.

Hládek, Daniel, Ján Staš, and Matúš Pleva (2020). "Survey of Automatic Spelling Correction". In: *Electronicsweek* 9.10, p. 1670.

Hu, Ziniu et al. (Mar. 2, 2020). "Heterogeneous Graph Transformer".

Hugging Face (Apr. 12, 2022). *Huggingface/Tokenizers*. Hugging Face.

Jayanthi, Sai Muralidhar, Danish Pruthi, and Graham Neubig (Oct. 2020). "NeuSpell: A Neural Spelling Correction Toolkit". In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online: Association for Computational Linguistics, pp. 158–164. DOI: 10.18653/v1/2020.emnlp-demos.21.

Joshi, Chaitanya (2020). "Transformers Are Graph Neural Networks". In: *The Gradient*.

Junczys-Dowmunt, Marcin et al. (June 2018). "Approaching Neural Grammatical Error Correction as a Low-Resource Machine Translation Task". In: *Proceedings*

of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers). New Orleans, Louisiana: Association for Computational Linguistics, pp. 595–606. DOI: 10.18653/v1/N18-1055.

Kukich, Karen (Dec. 1, 1992). "Techniques for Automatically Correcting Words in Text". In: *ACM Computing Surveys* 24.4, pp. 377–439. ISSN: 0360-0300. DOI: 10.1145/146370.146380.

LanguageTool (2022). *LanguageTool - Online Grammar, Style & Spell Checker*. LanguageTool. URL: https://languagetool.org/ (visited on 04/26/2022).

Levenshtein, Vladimir (Feb. 1966). "Binary Codes Capable of Correcting Deletions, Insertions and Reversals." In: *Soviet Physics Doklady* 10, pp. 707–710.

Li, Hao et al. (Nov. 1, 2018). "Spelling Error Correction Using a Nested RNN Model and Pseudo Training Data".

Lichtarge, Jared et al. (June 2019). "Corpora Generation for Grammatical Error Correction". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. NAACL-HLT 2019. Minneapolis, Minnesota: Association for Computational Linguistics, pp. 3291–3301. DOI: 10.18653/v1/N19-1333.

Loshchilov, Ilya and Frank Hutter (Jan. 4, 2019). "Decoupled Weight Decay Regularization".

Mays, Eric, Fred J. Damerau, and Robert L. Mercer (1991). "Context Based Spelling Correction". In: *Information Processing & Management* 27.5, pp. 517–522. ISSN: 0306-4573. DOI: 10.1016/0306-4573(91)90066-U.

Micikevicius, Paulius et al. (Feb. 15, 2018). "Mixed Precision Training".

Napoles, Courtney, Keisuke Sakaguchi, and Joel Tetreault (Feb. 13, 2017). "JFLEG: A Fluency Corpus and Benchmark for Grammatical Error Correction".

Omelianchuk, Kostiantyn et al. (July 2020). "GECToR – Grammatical Error Correction: Tag, Not Rewrite". In: *Proceedings of the Fifteenth Workshop on Innovative*

Use of NLP for Building Educational Applications. Seattle, WA, USA → Online: Association for Computational Linguistics, pp. 163–170. DOI: 10.18653/v1/2020.bea-1.16.

Ozinov, Filipp (Apr. 13, 2022). *JamSpell*.

Pandu Nayak (Mar. 29, 2021). *The ABCs of Spelling in Google Search*. Google. URL: https://blog.google/products/search/abcs-spelling-google-search/ (visited on 04/25/2022).

Paszke, Adam et al. (2019). "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., pp. 8024–8035.

Piktus, Aleksandra et al. (June 2019). "Misspelling Oblivious Word Embeddings". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, pp. 3226–3234. DOI: 10.18653/v1/N19-1326.

Sennrich, Rico, Barry Haddow, and Alexandra Birch (June 10, 2016). "Neural Machine Translation of Rare Words with Subword Units".

Stahlberg, Felix and Shankar Kumar (Apr. 2021). "Synthetic Data Generation for Grammatical Error Correction with Tagged Corruption Models". In: *Proceedings of the 16th Workshop on Innovative Use of NLP for Building Educational Applications*. BEA-EACL 2021. Online: Association for Computational Linguistics, pp. 37–47.

Tran, Hieu et al. (May 28, 2021). "Hierarchical Transformer Encoders for Vietnamese Spelling Correction".

Vaswani, Ashish et al. (2017). "Attention Is All You Need". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc.

Veličković, Petar et al. (Feb. 4, 2018). "Graph Attention Networks".

Wagner, Robert A. and Roy Lowrance (Apr. 1975). "An Extension of the String-to-String Correction Problem". In: *Journal of the ACM* 22.2, pp. 177–183. ISSN: 0004-5411, 1557-735X. DOI: 10.1145/321879.321880.

Walter, Sebastian (May 28, 2021). *Tokenization Repair Using Transformers*. URL: https://ad-blog.cs.uni-freiburg.de/post/tokenization-repair-using-transformers/ (visited on 05/22/2022).

Wang, Minjie et al. (2019). "Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks".

Wang, Xiao et al. (Jan. 20, 2021). "Heterogeneous Graph Attention Network".

Yang, Yilin, Liang Huang, and Mingbo Ma (Oct. 2018). "Breaking the Beam Search Curse: A Study of (Re-)Scoring Methods and Stopping Criteria for Neural Machine Translation". In: *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. EMNLP 2018. Brussels, Belgium: Association for Computational Linguistics, pp. 3054–3059. DOI: 10.18653/v1/D18-1342.

Yang, Zhilin et al. (Jan. 2, 2020). "XLNet: Generalized Autoregressive Pretraining for Language Understanding".

Yasunaga, Michihiro, Jure Leskovec, and Percy Liang (Oct. 7, 2021). "LM-Critic: Language Models for Unsupervised Grammatical Error Correction".

Zhou, Yingbo, Utkarsh Porwal, and Roberto Konow (May 17, 2019). "Spelling Correction as a Foreign Language".