GECKO+: A GRAMMATICAL AND DISCOURSE ERROR CORRECTION TOOL FOR ENGLISH

Eduardo Calò
IDMC
Université de Lorraine
Nancy, France
eduardo.calo6@etu.univ-lorraine.fr

Léo Jacqmin
IDMC
Université de Lorraine
Nancy, France
leo.jacqmin8@etu.univ-lorraine.fr

Thibo Rosemplatt
IDMC
Université de Lorraine
Nancy, France
thibo.rosemplatt3@etu.univ-lorraine.fr

January 10, 2021

ABSTRACT

In this paper, we introduce GECko+, a web-based writing assistance tool for English that corrects errors both at the sentence and at the discourse-level. Based on two existing state-of-the-art models for grammar error correction (GEC) and sentence ordering, this application is the first of its kind to go beyond sentence-level corrections. These two models work in conjunction to correct short paragraphs of text containing a combination of interdependent errors. We present the implementation details along with a case study to demonstrate the capabilities of this tool.

***K*eywords** grammatical error correction · sentence ordering · language assistant tool · BERT

1 Introduction

While most people can write, few would boast they never produce spelling and grammar mistakes, let alone systematically write coherent prose and express ideas clearly. Natural language processing (NLP) techniques have the potential to help in that regard. In particular, such technologies can have a beneficial impact on two issues related to the way we write.

First, NLP techniques can help us alleviate language-related discrimination. This type of discrimination particularly occurs in the professional world where job applications are rejected simply due to the quality of one's writing. Additionally, errorful writing is poorly perceived in social contexts and is often synonymous with barriers.

Second, those who are already proficient in writing can benefit from these techniques to improve the quality of their prose. This aspect applies to journalists, business-persons, college students, and teachers alike. These individuals are often required to write lengthy reports. The frequency with which these reports are produced is such that topological or consistency errors can occur. As a result, their message may not be delivered as intended.

To address these issues, we are introducing a novel digital writing assistance tool called GECko+. Recently, Transformer-based models have proven to be particularly effective for a broad range of NLP applications, and grammar error correction is no exception to this. Consequently, digital writing assistance tools have become more performant than ever. However, currently available tools focus solely on sentence-level mistakes and do not take discourse coherence into account. Our novel approach to grammar error correction tackles errors both at the sentence-level and at the discourse-level. To our knowledge, this approach was previously unheard of.

We have developed this system using existing state-of-the-art models. To correct spelling and grammar mistakes, we use GECToR [28], a grammatical error correction model developed by the well-known Grammarly[1]. For tackling discourse fallacies, we make use of a sentence ordering model[2] [29] based on Google's BERT[3]. We have created a web interface that users can access to correct paragraphs of text in English. The source code, together with the up-to-date link to the web interface, is publicly available on GitHub[4]. Our web application was then containerized with Docker[5] and deployed with Microsoft Azure[6].

The rest of the report is organized as follows. In Section 2, we will review what is considered a grammatical error along with related work in the areas of grammatical error correction and discourse analysis. Then, in Section 3, we will discuss in detail how the project has been implemented, presenting the technical choices that have been made. Afterward, in Section 4, we will study several use cases of our tool to help uncover its functioning and what are its potential shortcomings. Finally, in Section 5, we will draw the conclusions, describing the major difficulties we encountered, the current status of our application, and eventual improvements that could be made in the future.

2 Context

In this section, we present the theoretical framework surrounding grammatical errors and review how this problem has been approached in the literature.

2.1 Theoretical Background

Grammatical error is an umbrella term used in prescriptive grammar to describe an instance of faulty, unconventional, or controversial usage of a given language. This broad term generally refers not only to proper grammatical errors but also to other concepts, such as discourse fallacies, misspellings, and typographical errors.

Errors are found in language at all levels, i.e., both in speech and in written texts. The instances of faulty usage in the two linguistic modalities diverge considerably. Examples of errors in spoken language may include, among others, creation of improvised words and accidental addition or omission of linguistic material, mainly provoked by the dynamicity of the speech acts. However, our focus is on addressing the specificities of errors in written texts. Moreover, although all the issues we are going to introduce are valid for all languages, the focal point of our work, both in theoretical and practical implications, is English. Thus, all the examples and works cited will refer to English grammatical errors.

Grammatical errors have been traditionally identified at five different levels: lexical, syntactic, semantic, discourse, and pragmatic [20]. Lexical errors include misspellings that result in non-existent words, such as *type* vs. **tipe*, where **tipe* is not within the English lexical inventory. Syntactic errors include instances of faulty use where the grammatical categories of the words do not fit their contexts, such as subject-verb agreement errors, as in *She always *know/knows her place*, or verb tense errors, as in *The church *is/was rebuilt in 1948* [36]. Semantic errors are those which cause semantic anomalies, such as collocation errors, as in **big/long conversation* [19]. Discourse errors go beyond the span of a single sentence: they violate the inherent coherence relations between sentences in an utterance, such as those involving anaphora resolutions, as in *My mum doesn't like apples. *He/She likes pears*. Pragmatic errors arise through the faulty use of a speech act or one of the rules of speaking in given pragmatic contexts, such as dialog acts, as in *A: Do you want a piece of cake or some ice cream? B: Yes*.

An important theoretical distinction that should be underlined is that errors made by speakers of English as a second language (ESL or L2 speakers, i.e., learners) are significantly different from those made by natives (L1 speakers). Although the literature has mostly focused on L2 speakers' errors, L1 speakers are prone to make mistakes as well. Connors and Lunsford [10] and Donahue [14] have studied errors made by native college students in the United States and by ESL learners, concluding that most of the errors made by L1 speakers are negligible in L2 speakers. This happens because language learning is a complex and laborious process and involves several factors that could hamper the speed and quality of learning, such as inference of the learners' mother tongue, incomplete acquisition of a given rule, etc. In L2 cases, it is considered important to distinguish errors from mistakes. The former are defined as resulting from a learner's lack of proper grammatical knowledge, while the latter as performance errors [5], that is to say failures to utilize known rules properly.

[1] `https://github.com/grammarly/gector`
[2] `https://github.com/shrimai/Topological-Sort-for-Sentence-Ordering`
[3] `https://github.com/google-research/bert`
[4] `https://github.com/psawa/gecko-app`
[5] `https://www.docker.com/`
[6] `https://azure.microsoft.com/en-us/`

Within the practical application of our project, we do not discriminate between the aforementioned distinctions, trying to address all forms of spelling, morphosyntactic, lexical, typographical, and (mildly) discourse errors, regardless of the linguistic background of the user.

2.2 Prior Approaches and State of the Art

In this section, we review some of the earlier approaches along with the current state-of-the-art methods of grammatical error correction and discourse analysis.

2.2.1 Sentence Correction

In natural language processing (NLP), grammatical error correction (GEC) encompasses any sort of modifications made to automatically correct an errorful sentence. This includes spelling, punctuation, grammar, and word choice errors. Given a potentially errorful sentence or short piece of text as input, a GEC system is expected to output a corrected version of that text. See Table 1 below as an example.

Table 1: An example of grammatical error correction.

Input: erroneous text	**Output: corrected text**
I hope it is not been involved an inconvenience to you.	I hope it has not involved an inconvenience to you.

Early approaches to GEC consisted of rule-based systems. The first grammar checking tools were based on hard-coded corrections rules. Some systems [24] relied on basic pattern matching and string replacement, while others [7, 30] used hand-written grammar rules for syntactic analysis. Rule-based systems provide the advantage of being easy to implement and are still widely used today. However, due to the highly productive nature of language, such systems are avoided as a general solution.

In the 1990s, large-scale annotated resources became available and this led researchers to adopt data-driven approaches [18, 33, 12]. Machine learning (ML) classifiers were used to correct closed-class errors, i.e., specific error types that were easier to tackle, such as articles and prepositions. Such classifiers were to select the best pick among a set of all possible correction candidates. The training examples were represented as vectors using linguistic features such as part-of-speech (POS) tags, neighboring words, and dependency trees. The downside of machine learning-based approaches is that they treat specific error types as independent from one another. However, an errorful sentence may contain an intricate combination of errors that can be interdependent. This is illustrated in Example (1) below.

(1) *Robert thinks that aubergines is delicious.*
A system that combines independently-trained classifiers will correct *aubergines is* with *aubergine are*.

More recently, machine translation (MT) techniques have been used to solve this problem. GEC can be formulated as the automatic translation of a grammatically incorrect sentence into a correct equivalent. Such systems rely on correction mappings learned from parallel examples. Initially, statistical machine translation (SMT) was the dominant approach [4, 37, 26]. MT systems require large corpora to be trained effectively. Given that parallel corpora containing errorful sentences and their correction are difficult to acquire, a common technique is to generate data by automatically transforming well-formed sentences into ungrammatical ones [17].

Nowadays, neural approaches have been increasingly adopted for GEC [9, 15, 1]. These systems rely on the encoder-decoder framework [8] to project an input sentence into a vector representation, which is then decoded to output the corrected sentence. The current state-of-the-art approach to GEC was introduced with Grammarly's GECToR [28]. Rather than treating the problem as a sequence-to-sequence task, they treat it as a sequence tagging task, relying on a Transformer encoder. We discuss this model in more detail in Section 3.1.1.

2.2.2 Discourse Correction

We have reviewed several approaches to correct sentences individually. However, language does not simply consist of individual, independent sentences that are added one after the other, but rather forms a coherent whole composed of interconnected sentences. This coherent whole is commonly referred to as discourse. The area of NLP concerned with how sentences fit together is called discourse coherence or discourse analysis [16].

A typical user of digital writing assistance tools is looking to craft a well-written, coherent piece of text, be it an essay, a report, or an article. Ideally, such tools should be able to handle paragraphs of text and correct any discourse fallacies in addition to sentence-level errors. For this reason, we deemed it important to take discourse coherence into account for our tool. As far as we are aware, no writing assistance tool currently available goes beyond sentence-level errors. This is no surprise given that dealing with discourse is more difficult than processing individual sentences. Discourse coherence remains an open problem for the NLP community.

There are multiple factors that make a discourse coherent, and discourse can be analysed both at the local and global level. A local analysis may include coherence relations between nearby sentences [34], entity-based coherence which tracks salient entities [2], as well as topical coherence [3]. At the global level, we may be concerned whether a piece of text follows the appropriate writing conventions. All in all, discourse analysis encompasses many different aspects and can be very fine-grained. To reduce both complexities in computation and implementation, we decided to focus on one aspect of discourse coherence with sentence ordering.

The goal of sentence ordering is to arrange sentences of a given text in the correct order, i.e., in a coherent manner. An example of this is shown in Table 2.

Table 2: An example of sentence ordering. The correct order is **S1**, **S2**, **S3**, **S4**. **S2** coming before **S3** is a more coherent order since the entity *Central Park* is first introduced in **S2**.

S1: I used to love spending time with my daughter.
S3: In the park, we took long walks, talking about anything and everything.
S2: We would go to Central Park on Sundays.
S4: I miss these days, watching the dogs running around.

Early approaches to sentence ordering used probabilistic transition models. For example, Barzilay and Lee [3] modeled content by representing topics as states in an hidden Markov model (HMM). Barzilay and Lapata [2] later introduced another similar method based on entities. With recent neural approaches, this task has typically been formulated as a sequence prediction problem [23, 11, 21]. Relying on the encoder-decoder framework, the decoder uses the document representation to output the index of the sentences in the right order. In contrast, Prabhumoye et al. [29] have introduced a novel way to solve this task by treating it as a constraint solving problem, achieving a better capacity to capture coherence in documents. Their system, which we will describe in more detail in Section 3.1.2, serves as the basis for discourse correction in our application.

3 Implementation

This section deals with the methodology we have used to approach the task of grammatical error correction, the models we have exploited, and the pipeline we have adopted. Moreover, we explain how the web application has been developed, organized, and deployed.

3.1 Models

In this section, we present the details of the neural models for sentence and discourse correction we use within our application.

3.1.1 Sentence Correction

To address errors at the level of the sentence, we make use of GECToR, a model that achieves state-of-the-art results for the GEC task[7]. This novel approach diverges from neural machine translation (NMT) methodologies common in GEC nowadays: instead of directly generating the sentence in its correct form starting from an errorful one, the problem is treated as a sequence tagging task. The sequence of words is tagged with custom transformations describing corrections to be made, using a human-readable tagset[8], which also substantially accelerates training.

[7] `http://nlpprogress.com/english/grammatical_error_correction.html`

[8] The tagset is composed of 5000 tags. This number was chosen as a compromise between performance and speed: were the tagset too big, the model would be unwieldy and slow; were it too small, the errors' coverage would not be enough. The majority of the tags are of four basic types: $KEEP, $APPEND, $DELETE, and $REPLACE.

Preprocessing Starting from a parallel corpus made of errorful-correct sentence pairs, for each pair, the preprocessing algorithm generates transformation tags, one tag for each token in the source sequence. To better illustrate the process let us consider the following example:

- Errorful sentence: A ten years old boy go school
- Correct sentence: A ten-year-old boy goes to school.

The algorithm tags the sentence in three steps. First of all, each token in the source sequence is aligned with one or more tokens from the target sequence, minimizing the overall Levenshtein distance [22] of possible transitions between the source tokens and the target tokens:

[A → A], [ten → ten, -], [years → year, -], [old → old], [boy → boy], [go → goes, to], [school → school, .]

Afterward, each mapping is converted into the tag or tags that represent the transformation:

[A → A]: $KEEP, [ten → ten, -]: $KEEP, $MERGE_HYPHEN, [years → year, -]: $NOUN_NUMBER_SINGULAR, $MERGE_HYPHEN, [old → old]: $KEEP, [boy → boy]: $KEEP, [go → goes, to]: $VERB_FORM_VB_VBZ, $APPEND_{to}, [school → school, .]: $KEEP, $APPEND_{.}

Finally, if there are multiple tags for a token, only the first tag that is not $KEEP is kept:

A → $KEEP, ten → $MERGE_HYPHEN, years → $NOUN_NUMBER_SINGULAR, old → $KEEP, boy → $KEEP, go → $VERB_FORM_VB_VBZ, school → $APPEND_{.}

Model Technically, the architecture of the model consists of an encoder made up of a pre-trained BERT-like Transformer, stacked with two linear layers, with softmax layers on the top. The two linear layers are responsible for mistake detection and token-tagging, respectively. The architecture of the model is shown in Figure 1. Training was performed through different stages, namely a pre-training stage using synthetic data [1], and two additional fine-tuning training stages using real data [27, 35, 31, 12, 6].

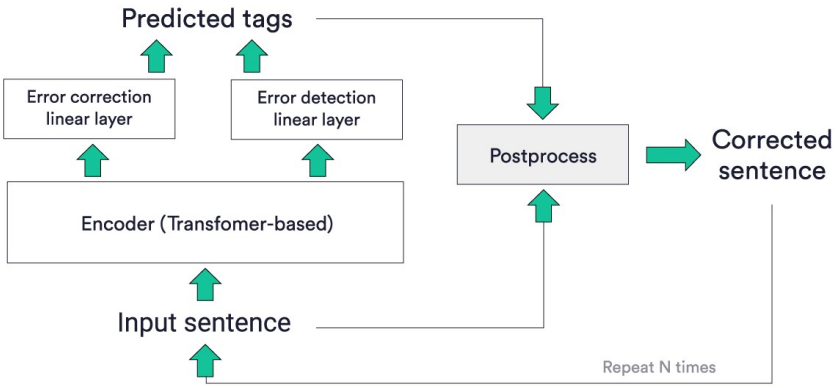

Figure 1: GECToR's architecture.

Inference The model predicts the tag transformations for each token in the input sequence. Clearly, some corrections in a sentence may depend on others, thus, applying the GEC sequence-tagger only once may not be enough to fully correct the sentence. Therefore, an iterative correction approach is implemented: the tagger is applied iteratively to the same sentence for a maximum of five times. Refer to Table 3 below for an example.

Table 3: Iterative correction process. Corrections are highlighted in bold. The third column indicates the cumulative number of corrections after each iteration.

Iteration	Sentence's evolution	# of corrections
Original	A ten years old boy go school	-
1	A ten**-**years old boy **goes** school	2
2	A ten-**year-**old boy goes **to** school	5
3	A ten-year-old boy goes to school**.**	6

3.1.2 Discourse Correction

To tackle discourse errors, we use a state-of-the-art sentence ordering model based on BERT[9] [29]. This novel approach frames the task as a constraint learning problem. Essentially, the model is trained to predict the correct constraint given a pair of sentences. The constraint represents the relative ordering between the two sentences. Given a set of constraints between the sentences of a text, the right order of the sentences is found by topological sorting.

Task Definition More formally, the task can be stated as follows. Starting from an input set $I = \{s_{o_1}, ..., s_{o_n}\}$ of n sentences in a random order, where the random order is $o = [o_1, ..., o_n]$, the aim is to find the right order of the sentences $o^* = [o_1^*, ..., o_n^*]$. The set of constraints C represents the relative ordering between every pair of sentences in I. Hence, we have $|C| = \binom{n}{2}$ constraints. For example, if the number of sentences in I is equal to four, whose correct order is $s_1 < s_2 < s_3 < s_4$, then we have a set $C = \{s_1 < s_2, s_1 < s_3, s_1 < s_4, s_2 < s_3, s_2 < s_4, s_3 < s_4\}$ of six constraints. To note that by using this method, not all the permutations of an order are to be considered, thus reducing the complexity of the model.

Model Technically, the model that learns the constraints is a BERT pre-trained uncased language model [13], fine-tuned on NIPS dataset [23] using a fully connected perceptron layer. Specifically, this approach leverages the Next Sentence Prediction (NSP) objective of BERT to get a single representation for both sentences s_1 and s_2 of each pair. The input to the BERT model is the sequence of tokens of s_1, followed by the separator token [SEP], followed by the sequence of tokens of s_2.

Topological Sort Afterward, the right order o^* is found using topological sort algorithm on the relative ordering between all the constraints in C. Topological sort [32] is a standard algorithm for linear ordering of the vertices of a directed graph. The sort produces an ordering $\hat{o}$ of the vertices such that for every directed edge $u \rightarrow v$, u comes before v in the ordering $\hat{o}$. A depth-first search-based algorithm that loops through each node of the graph in an arbitrary order is used. The topological sort finds the correct ordering o^* of the sentences in the input text. The sentences represent the nodes of the directed graph and the edges are represented by the ordering between the two sentences. The directions of the edges are the constraints predicted by the classifier. For example, if s_1 comes first than s_2 in the order, then the direction of the edge will go from s_1 to s_2, i.e., $s_1 \rightarrow s_2$.

3.2 GECko+'s Pipeline

The pipeline we decided to adopt for GECko+ is summarized in Figure 2. As the diagram shows, the text given as input by the user gets segmented into sentences. After the segmentation, we obtain a list S of sentences, whose length can range from one to n. For each sentence s_i in S, we apply GECToR, in order to perform sentence-wise error correction. At this moment in the pipeline, S contains n sentences, grammatically corrected, but possibly unordered. If $n = 1$, we directly output the single corrected sentence to the user. Conversely, if S contains more than one element, the sentence ordering model is applied to S. Once the sentences are ordered, the output is displayed to the user.

[9] `http://tts.speech.cs.cmu.edu/sentence_order/nips_bert.tar`

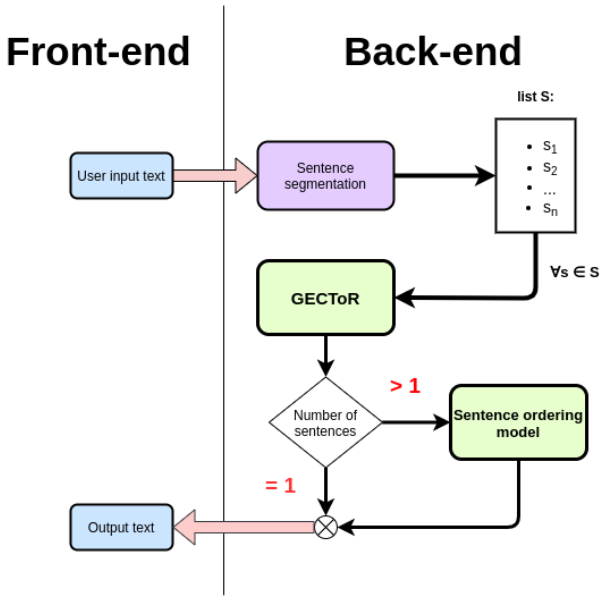

Figure 2: GECko+'s pipeline.

3.3 Software Architecture

We developed GECko+ as a web application. We used the web-development framework *Flask*[10], which allowed us to host a development server that includes a debugger. It also provides a handy HTML template rendering engine. The programming languages used are naturally HTML5 and Python, but also JavaScript (JS), and Sass.

The application stores data that are needed for the web-app, i.e., HTML, SCSS, and JS files, images for the website, and Python Flask files. As our application relies on two neural models, we also store them within the application.

Since the application started to become heavy and convoluted during development, we reorganized the architecture opting for a modular structure. This facilitated development and code reviewing, allowing the execution of the pipeline from the root files, with fewer lines of code. Figure 3 shows that the modular structure also helps separate the files in different folders, depending on the function they hold. As visible in the schema, the Flask endpoint `run.py` is contained in the root folder. `application` is the source folder, containing other Python files called collectively *controllers*, since they are used to control what happens in the back-end. The source folder also contains three sub-folders: `models` gathering all the different ML models, `static` containing the files with which the user will not interact, and `template` containing the HTML files that structure the web-app.

Using a modular structure also facilitates maintenance and upgradability. For instance, in case developers want to upgrade functionalities by adding a new model, it is sufficient to add a folder containing the new ML model in `models`, and to import it from the *controllers*.

[10] `https://github.com/pallets/flask`

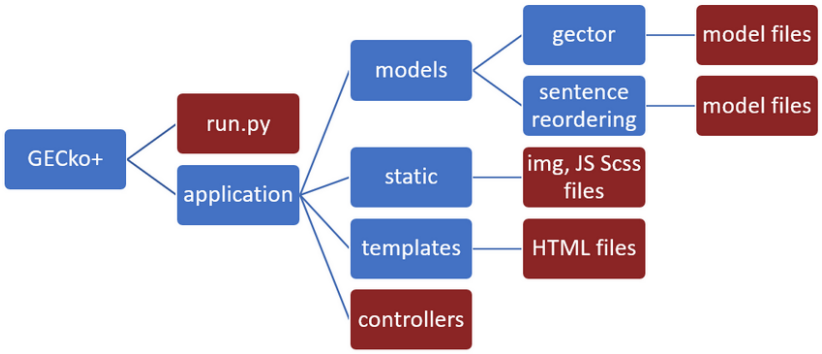

Figure 3: GECko+'s software structure.

3.4 UX Design

Offering a smooth and intuitive user experience is something we always aimed for during the development phase. GECko+, like many other applications offering functionalities based on machine learning, is targeting a wide audience. The average user is likely to be unfamiliar with the technologies used. With this in mind, we felt it was important for the interface to be intuitive and easy to use, so that any user would feel comfortable using the application.

We chose to build a sleek single-page website. This way, we tried to avoid any superfluous information in order to highlight the features. The layout of the website is shown in Figure 4. The interface presents three main sections: (1) the header, with the logo, a tagline, and a navigation bar, (2) the user interaction area, with input and output boxes, a brief introduction and a demo button, and (3) the about section, with the description of our team on the left, and the description of the project on the right.

A feature that we deemed important to have from the beginning is a way to highlight what changes occur to the input text. Initially, we used the sequence comparison library *difflib*[11] to compute the difference between the input and output strings at the character-level. However, we realized that visualizing changes at character-level is not intuitive, as grammatical modifications might not be seen at first glance. Thus, the approach we have finally opted for is to highlight changes token-level. Implementing this feature posed a challenge as there is no one-to-one mapping between input and output tokens. To solve this, we have used GECToR's formulation as a sequence tagging task. First, the system predicts a correction for an input text. Next, using one of GECToR's scripts intended for preprocessing a parallel corpus (see Section 3.1.1), we treat the new prediction as the gold standard corrected text, and we use it to tag the original input text with the modifications needed. We are then able to retrieve the changes made at the token-level.

We use an intuitive color code to highlight changes:

- In the input text box, deletions are underlined in red.
- In the output text box, modifications are underlined in blue, and additions are underlined in green.

This feature ensures that users can quickly visualize the mistakes they may have made and learn from them. An example can be seen in Figure 5.

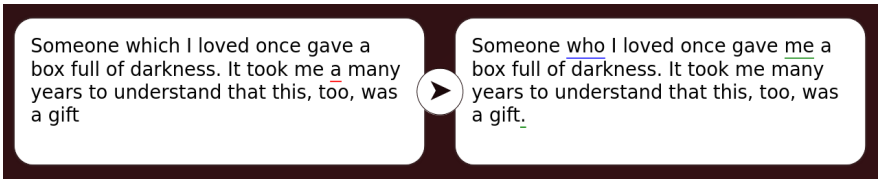

Figure 5: Example of how changes are highlighted.

[11] `https://docs.python.org/3/library/difflib.html`

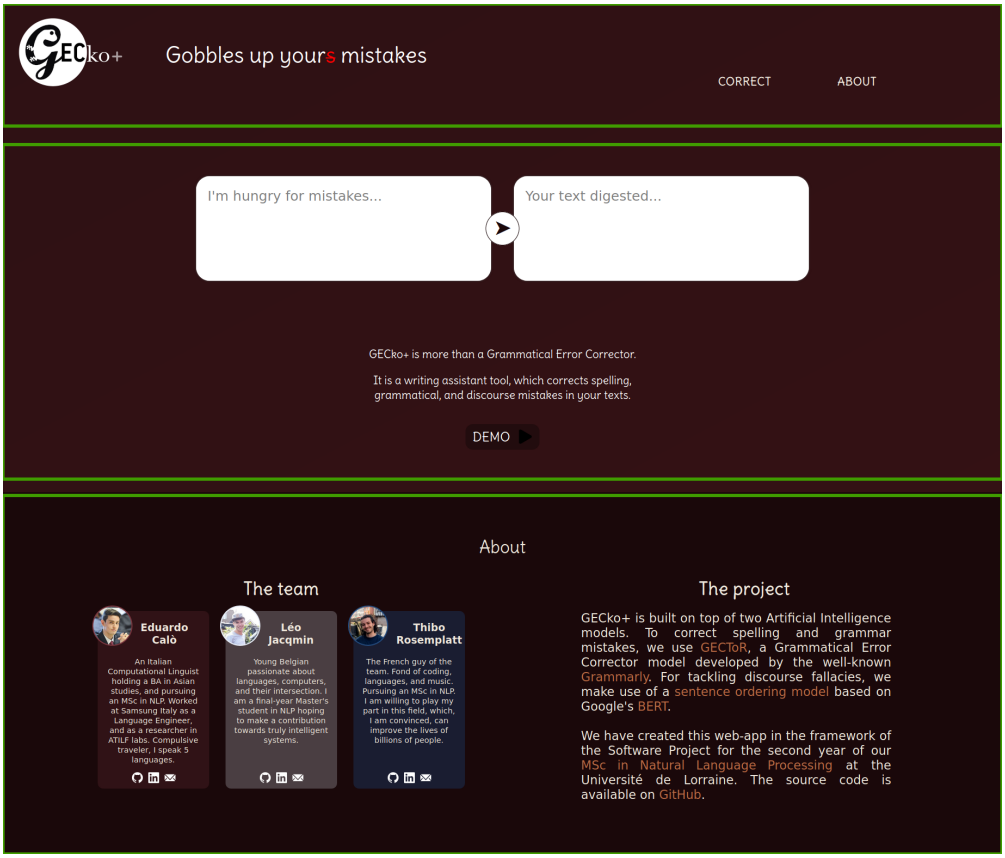

Figure 4: GECko+'s desktop layout.

As part of our commitment to providing a seamless experience, we have implemented keyboard shortcuts. For example, the `Enter` key will launch the correction, and the return to the line will be triggered by pressing `Shift` + `Enter`. These key-bindings have been programmed in JavaScript.

We also put a particular effort into adapting the interface for mobile and tablet devices, as they have become more and more widespread in regard to web browsing. In that matter, we adopted a responsive design that renders a different layout according to the device used. This has been achieved using CSS media queries. Figure 6 shows how the website is displayed on various mobile devices.

Figure 6: Responsive design of GECko+ on different devices.

3.5 Deployment

A container allows a developer to package an application and all of its parts, i.e., the stack the application runs on and the dependencies that are associated with it. Using this technology, we are able to package our web application in a container, which is an isolated environment, along with all the necessary parts it needs to run. In other words, the application is no longer dependent on the underlying host, that is to say the environment on which it runs. This technology allows us to deploy our application with no dependency problem. Docker [25] provides an easy way of doing this. Docker separates applications from infrastructure using container technology, similar to how virtual machines separate the operating system from bare metal. Thanks to this technology, our dockerized application can be deployed anywhere. Moreover, we can easily add new layers to our initial dockerized application. This provides a way of iteratively developing our application without having to recreate the container from scratch. Following are the workflow components used to dockerize our application:

- A Docker image that holds the environment and our application
- A `Dockerfile` that automates image construction
- A `Dockerignore` file that allows ignoring files during the build
- A Docker container that instantiates our Docker image to run the application
- A Docker repository where we can store images and work collaboratively

Once the dockerization is complete, a platform to host it is needed. Initially, we tried to deploy it using Heroku[12]. Heroku is a platform as a service (PaaS) that is fully managed, meaning deployment is simplified. It provides a free-tier service to deploy non-commercial applications and personal projects. Unfortunately, we found out that our application exceeded both space and memory limits of this service. As our application contains two pre-trained models and is quite heavy, we were forced to rely on another more robust service. Among other competitors, we have opted for Microsoft Azure with which we are able to use free student credits. Azure is a cloud computing platform that provides various web services. Among these services is the App Service that we used to deploy our application. Given an App Service Plan, the platform dedicates a set amount of storage and memory to our application. The App Service retrieves the Docker image from our Docker repository and launches the application as a container. The application is then accessible online.

When deploying preliminary versions of GECko+, we used the native serving utility integrated within Flask. This comes as a useful feature during the development phase, since it does not require any extra dependencies to serve the website in one click. However, performances will suffer if it is used in a production environment. The developers of Flask themselves advise against using this built-in feature in production[13]. The main reason why the Flask serving utility is not suited for production is that this feature is only a convenient tool for development, and does not support requests coming from multiple users at once. In addition to that, a production-ready server would be faster at serving static files, such as HTML and CSS. To overcome this limitation, we used *Waitress*[14], a Python Web Server Gateway Interface (WSGI) dedicated to production environments. Using Waitress significantly improved the response time of our queries.

Having a local server integrating a debug utility was still essential for development. This is why we studied various ways to have separated production and development environments within the same version of the application, depending on where the code is run from.

For this purpose, we decided to program the application to have the following behavior:

1. On start, the application uses a production server by default, without the debug mode.
2. Right after setting-up the default configuration, the application looks for an optional configuration file located in the root folder. If the file is found, the parameters contained inside will overwrite the default parameters of the application. If not, the application will remain on its default configuration. Parameters in the configuration file can be of any kind among those offered by Flask.

The application now launches with default settings. Then, we added a configuration file in the repository, indicating to use a development environment and to use the debug mode. This file will be read on launch when working locally. However, those parameters are not wanted in production, so the configuration file must be omitted from the container in order to use the default settings when the app is deployed. To this end, we made use of a `Dockerignore` file that is set

[12] `https://www.heroku.com/`

[13] `https://flask.palletsprojects.com/en/1.1.x/tutorial/deploy/`

[14] `https://github.com/Pylons/waitress`

to ignore the configuration file when building the Docker image. This way, the default parameters are used when the app is containerized, i.e., it uses a production environment without the debug mode.

4 Use Cases

This section presents a list of case studies we have performed using our final tool to evaluate its performances and the actual corrections it can perform in real case usages. The use cases can be seen in Table 4. The table presents the input text on the left column, the text output by the system on the central column, and, on the right column, the analysis we have carried out on the errors the tool is able to correct. The corrections are consistently highlighted with the color code used within the application, which we described in Section 3.4.

Table 4: GECko+'s working use cases.

Input text	**Output text**	**Type of correction**
I luv the apples.	I love apples.	Sentence-level: (1) spelling of *love*, (2) deletion of superfluous determiner *the*.
I do not know how it look like.	I do not know what it looks like.	Sentence-level: (1) lexical choice in grammatical construction **how/what x looks like*, (2) person agreement in *it looks*.
Joe Smith is a old Englishman who lives in London all his life.	Joe Smith is an old Englishman who has lived in London all his life.	Sentence-level: (1) correct orthographic variant *an* before a vowel, (2) present perfect tense should be used in these constructions.
She was recently fired from his job as architect	She was recently fired from her job as an architect.	Sentence-level: (1) agreement of personal pronoun *her* with *She*, (2) addition of determiner *an*, (3) punctuation added.
She refuses to listen to them and is running out of the room when they bring up the subject. They advised Mary having an abortion.	They advised Mary to have an abortion. She refuses to listen to them and runs out of the room when they bring up the subject.	Sentence-level: (1) infinitive vs. gerund in fixed construction *advise x *having/to have y*, (2) present simple tense *runs* should be used in routine actions. Discourse-level: (1) sentences reordered since the entity *Mary* was defined after reference.
This chemical is widly used in the swimming pools market. Chlorine is well known for its sanitizing properties.	Chlorine is well known for its sanitizing properties. This chemical is widely used in the swimming pool market.	Sentence-level: (1) spelling correction of *widely*, (2) noun adjuction in *swimming pool market* should be singular. Discourse-level: (1) sentences reordered to define entity *Chlorine* first.
Today, most NLP tasks rely on neural methods. Natural language processing (NLP) is a science at the crossroad of computer science and linguistics.	Natural language processing (NLP) is a science at the crossroads of computer science and linguistics. Today, most NLP tasks rely on neural methods.	Sentence-level: (1) spelling correction of the idiom *at the crossroads*. Discourse-level: (1) sentences reordered to expand the acronym *NLP* before subsequent reference.
Fool me twice, shame on me. Fool me once, shame on you.	Fool me once, shame on you. Fool me twice, shame on me.	Discourse-level: (1) common English saying reordered.
The wood are lovely, dark, ,and deep. And miles to go before I sleep. But I have promises to keep.	The woods are lovely, dark, and deep. But I have promises to keep. And miles to go before I sleep.	Sentence-level: (1) subject-predicate agreement in *woods are*, (2) punctuation typo in *and*. Discourse-level: (1) famous quote by Robert Frost reordered.
i dont like going there. i consider swimming pool bored	I consider swimming pool boring. I don't like going there.	Sentence-level: (1) uppercasing of personal pronoun *I* twice, (2) semantic error in **bored/boring* couple, (3) apostrophe's insertion in *don't*. Discourse-level: (1) sentences reordered to introduce the entity *swimming pool* before deictically referring to it using *there*.
Finally, spread the topping, bake, and serve. Secondly, place cod fillets on a paper towel. First, preheat the oven to 220 degrees. Next, line a rimmed baking sheet with parchment paper. Then, mix together the bread crumbs and oregano in a bowl. Afterward, season both sides of each fillet with salt and pepper and place on the prepared baking sheet.	First, preheat the oven to 220 degrees. Secondly, place cod fillets on a paper towel. Next, line a rimmed baking sheet with parchment paper. Then, mix together the bread crumbs and oregano in a bowl. Afterward, season both sides of each fillet with salt and pepper and place on the prepared baking sheet. Finally, spread the topping, bake, and serve.	Discourse-level: (1) correct reordering of a long recipe with linking words that highlight the logical steps to follow.

From the analysis, we can see that the two models integrated into our app mesh well together, working in tandem to correct sentences containing a combination of interdependent errors. The system can properly address and correct a wide range of errors at both sentence and discourse-level. Sentence-wise, the tool can deal with orthographic, morphological, syntactic, and lexical errors. Discourse-wise, the application can tackle problems regarding anaphoric reference, deixis, and linking words, correctly reordering several types of texts, from general language to technical texts, and even sayings and quotes.

5 Conclusion

We have described the implementation details of GECko+, a writing assistant tool for English. Unlike other applications of its kind that only deal with errors at the sentence-level, GECko+ is able to provide corrections related to discourse. Thanks to this innovative feature, this web-based application can help users craft coherent and error-free pieces of writing. To achieve this, GECko+ relies on two state-of-the-art Transformer-based models that respectively handle grammar error correction and sentence ordering. Following are some comments on this project: the difficulties we faced while carrying it out, what is currently working and not working as well as the extent to which it does what we wanted it to do, and finally the issues left and possible improvements.

5.1 Difficulties

Pipeline Plugging the two models in the pipeline turned out to be not as effortless as one might think. We learned that each model is meant to be used differently. They all have their own way of being instantiated and manipulated. One has to learn how to install, launch, and use them for predictions. Some models are developed for training and batch-evaluation only, and in that case some coding is needed in order to perform a simple prediction.

In particular, the sentence ordering model [29] is peculiar in its architecture. Throughout the whole pipeline, data is temporarily saved on the hard disk, instead of using the RAM, like most models do. This leads to technical issues that eventually corrupted the output when the application was deployed. For example, when multiple users queried the model at the same time, temporary data was created for each query. However, this temporary data is always saved under the same file name, hence new data would erase the previous one. Consequently, files related to a query were overwritten before this query could get a response, and the output ended up being altered. This issue has been fixed by naming files such that they are identifiable as belonging to a certain query.

Errors' Highlighting As we mentioned in Section 3.4, we were able to leverage GECToR's implementation as a sequence tagging task to develop a functionality that highlights changes made at the token-level. However, this was not so straightforward as GECToR does not necessarily handle each token separately. For example, were a sentence finishing with the word *over* to miss a stop at the end, no $APPEND_{.} tag would be added, but rather the word *over* would be tagged as $REPLACE_{over.}, which is trickier to work with. We were able to work around this issue by explicitly specifying when punctuation had been added. We also added ad-hoc regular expressions to deal with detokenization.

Deployment Having no prior experience with software deployment, we faced somewhat of a steep learning curve when dealing with Docker and Microsoft Azure. Azure in particular turned out to be difficult to handle as its services cover a wide range of areas and the documentation does not always precisely relate to what one is trying to achieve.

Discourse Analysis As we have seen in Section 2.2.2, this is a complex task to address, since the relations holding between sentences composing a text are numerous and diverse. Ideally, we would have had a system that tackles discourse analysis and correction in all its complexity, but, given the various constraints, we decided to simplify the task to a less fine-grained sentence ordering, which can still mildly address some discourse errors occurring in texts.

5.2 Current Status

General Impressions The project has been very rewarding since the beginning. Throughout its whole duration, we acquired knowledge in both theoretical issues and practical applications. First of all, we discovered the nuances of the linguistic concepts of grammatical error and discourse analysis, and studied NLP approaches that have been developed to deal with them. Moreover, during the development of the software, we had the opportunity to deeply understand and exploit very recent state-of-the-art models. Furthermore, we learned how to efficiently split the tasks among us and deliver them within self-imposed deadlines. We improved our coding and code management skills, having to deal with such a vast software. Finally, we became aware of the huge and complex world of DevOps when deploying the app. All in all, having a smooth and responsive working application deployed online, we humbly consider both the execution and the outcome of the project to be successful, easily exceeding our initial expectations.

Errors' Detection Clearly, our tool is not flawless. Mainly, correction failures come from the two pre-trained models (GECToR and sentence ordering) we have included in our application. Although being state-of-the-art models, they are not devoid of defects. Table 5 presents a series of errors that are not detected by our tool. The table has four columns: the input text, the output text by the app, the expected correction, and a brief explanation of what did not work. As

we did in Section 4, we highlight errors using the color code of the application. In the expected results, our manual corrections are in bold.

Moreover, we noticed that the sentence ordering model might give different orders in consecutive predictions of the same text, mainly when the given text is long. This might be due to the search performed by the topological sorting algorithm or to the varying constraints' values assigned to each pair of sentences (see Section 3.1.2).

A solution for correction failures would be re-training or fine-tuning the two models with more real or synthetic data. However, this is not feasible, given the amount of computation power needed to perform these tasks.

Table 5: GECko+'s failures in corrections.

Input text	**Output text**	**Expected result**	**Explanation**
Yesterday the weather was nice. I don't need to take an umbrella.	Yesterday the weather was nice. I don't need to take an umbrella.	Yesterday the weather was nice. I **didn't** need to take an umbrella.	No correction is made since the sentence ordering model just reorders sentences and does not address actual discourse. Taken singularly, the sentences are well-formed, so GECToR rightfully does not perform corrections.
Her computer is the one who overheats all the time.	Her computer is the one who overheats all the time.	Her computer is the one **that** overheats all the time.	GECToR loses track of the pronominal dependencies and does not detect that the relative pronoun is referred to the initial *Her computer*.
My boss is misrespectful to her employees.	My boss is misrespectful to her employees.	My boss is **disrespectful** to her employees.	GECToR does not detect the morphological error made by forming the negative adjective using the prefix *mis-*, instead of the appropriate *dis-*.
I described John the landscape.	I described John the landscape.	I described **the landscape to John**.	GECToR fails in swapping the oblique object with the direct object and in adding the particle *to* to the oblique object.
Me and my dad went fishing.	Me and my dad and went fishing.	**My dad and I** went fishing.	GECToR totally fails with this sentence: it does not recognize that *me* is an object pronoun and should not be used as a subject, thus it also fails in placing the correct subject pronoun *I* in the second place in the noun phrase. Moreover, it adds a superfluous *and*.
lack of ressources could lead to a precocous demise.	lack of resources could lead to a precocous demise.	**Lack** of resources could lead to a **precocious** demise.	GECToR succeeds in correcting the typo in *resources*, but fails in uppercasing the initial grapheme of *lack*, and correcting the misspelling in *precocious*.
Joe Smith divorced a year ago. His two daughters now lives with his ex-wife. The older girl has recently become pregnant and would like to keeping the baby. Joe think that she is too young for the responsibility of bringing up a child as a single parent.	Joe Smith got divorced a year ago. The older girl has recently become pregnant and would like to keep the baby. Joe thinks that she is too young for the responsibility of bringing up a child as a single parent. His two daughters now live with his ex-wife.	Joe Smith got divorced a year ago. **His two daughters now live with his ex-wife.** The older girl has recently become pregnant and would like to keep the baby. Joe thinks that she is too young for the responsibility of bringing up a child as a single parent.	The sentences are reordered despite the input text being in the right order. One possible explanation for this is that the sentence ordering model does not associate *His two daughters* with *The older girl*. Replacing *The older girl* with *The older daughter* results in the proper order being kept in about half of the consecutive predictions (5 out of 10).

Input Text Box Currently, when text is pasted in the input box, it keeps its original formatting, e.g., font-style, text-decoration, color, etc. We introduced a functionality to remove this unwanted formatting, but it resulted in another bug: when selecting a text to paste something over it, the selected text remained and the pasted text was added at the end. Similarly, when intending to paste at a specific position in the text, the pasted text was again added at the end. This bug caused more trouble than having the text formatting remain, so we removed this functionality.

5.3 Future Work

Visualize Sentence Reordering Currently there is no explicit indication of how sentences have been reordered. Ideally, a user should be able to visualize which sentences were swapped. One possible way to do this with minimal clutter would be to highlight corresponding sentences in both text boxes when a user hovers his mouse over a sentence. However, adding this feature might be cumbersome, and given the time, we did not try to further explore it.

Remove Formatting in Input Text As discussed previously in Section 5.2, a string pasted in the input box will keep its original formatting. One useful improvement to add would be to remove the formatting of text pasted in the input box.

Miscellaneous Interesting ideas we have come up with, but decided not to implement for several reasons like time or computational constraints, or because we decided to explore other paths, include insertion of additional languages, improvement of the pre-trained models used in our application, and customizing the user's experience by adding a glossary of personalized self-corrections' entries. Implementation of these features is not excluded in the future.

References

[1] A. Awasthi, S. Sarawagi, R. Goyal, S. Ghosh, and V. Piratla. Parallel iterative edit models for local sequence transduction. *arXiv preprint arXiv:1910.02893v2*, 2020.

[2] R. Barzilay and M. Lapata. Modeling local coherence: An entity-based approach. *Computational Linguistics*, 34 (1):1–34, Mar 2008. ISSN 0891-2017, 1530-9312. doi: 10.1162/coli.2008.34.1.1.

[3] R. Barzilay and L. Lee. Catching the drift: Probabilistic content models, with applications to generation and summarization. In *Proceedings of the Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics: HLT-NAACL 2004*, pages 113–120, Boston, Massachusetts, USA, May 2 - May 7 2004. Association for Computational Linguistics. URL `https://www.aclweb.org/anthology/N04-1015`.

[4] C. Brockett, W. B. Dolan, and M. Gamon. Correcting ESL errors using phrasal SMT techniques. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics*, pages 249–256, Sydney, Australia, July 2006. Association for Computational Linguistics. doi: 10.3115/1220175.1220207. URL `https://www.aclweb.org/anthology/P06-1032`.

[5] H. D. Brown et al. *Principles of language learning and teaching*, volume 4. Longman New York, 2000.

[6] C. Bryant, M. Felice, Ø. E. Andersen, and T. Briscoe. The bea-2019 shared task on grammatical error correction. In *Proceedings of the Fourteenth Workshop on Innovative Use of NLP for Building Educational Applications*, page 52–75. Association for Computational Linguistics, 2019. doi: 10.18653/v1/W19-4406. URL `https://www.aclweb.org/anthology/W19-4406`.

[7] F. R. Bustamante and F. S. León. GramCheck: A grammar and style checker. In *COLING 1996 Volume 1: The 16th International Conference on Computational Linguistics*, 1996. URL `https://www.aclweb.org/anthology/C96-1031`.

[8] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, page 1724–1734. Association for Computational Linguistics, 2014. doi: 10.3115/v1/D14-1179. URL `http://aclweb.org/anthology/D14-1179`.

[9] S. Chollampatt and H. T. Ng. A multilayer convolutional encoder-decoder neural network for grammatical error correction. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[10] R. J. Connors and A. A. Lunsford. Frequency of formal errors in current college writing, or ma and pa kettle do research. *College composition and communication*, 39(4):395–409, 1988.

[11] B. Cui, Y. Li, M. Chen, and Z. Zhang. Deep attentive sentence ordering network. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 4340–4349, Brussels, Belgium, Oct.-Nov. 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1465. URL `https://www.aclweb.org/anthology/D18-1465`.

[12] D. Dahlmeier, H. T. Ng, and E. J. F. Ng. NUS at the HOO 2012 shared task. In *Proceedings of the Seventh Workshop on Building Educational Applications Using NLP*, pages 216–224, Montréal, Canada, June 2012. Association for Computational Linguistics. URL `https://www.aclweb.org/anthology/W12-2025`.

[13] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[14] S. Donahue. Formal errors: Mainstream and esl students. In *Conference of the Two-Year College Association (TYCA)*, 2001.

[15] M. Junczys-Dowmunt, R. Grundkiewicz, S. Guha, and K. Heafield. 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), pages 595–606, New Orleans, Louisiana, June 2018. Association for Computational Linguistics. doi: 10.18653/v1/N18-1055. URL https://www.aclweb.org/anthology/N18-1055.

[16] D. Jurafsky and J. H. Martin. *Speech and Language Processing (2nd Edition)*. Prentice-Hall, Inc., USA, 2009. ISBN 0131873210.

[17] S. Kiyono, J. Suzuki, M. Mita, T. Mizumoto, and K. Inui. An empirical study of incorporating pseudo data into grammatical error correction. 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)*, page 1236–1242. Association for Computational Linguistics, 2019. doi: 10.18653/v1/D19-1119. URL https://www.aclweb.org/anthology/D19-1119.

[18] K. Knight. Automated postediting of documents. In *Proc. 12th National Conference on Artificial Intelligence, Washington, USA, Aug. 1994*, pages 779–784, 1994.

[19] E. Kochmar. Error detection in content word combinations. Technical report, University of Cambridge, Computer Laboratory, 2016.

[20] K. Kukich. Techniques for automatically correcting words in text. In *Proceedings of the 1993 ACM conference on Computer science - CSC '93*, page 515. ACM Press, 1993. ISBN 9780897915588. doi: 10.1145/170791.171147. URL http://portal.acm.org/citation.cfm?doid=170791.171147.

[21] P. Kumar, D. Brahma, H. Karnick, and P. Rai. Deep attentive ranking networks for learning to order sentences. *arXiv preprint arXiv:2001.00056*, 2019.

[22] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics. Doklady*, 10:707–710, 1965.

[23] L. Logeswaran, H. Lee, and D. Radev. Sentence ordering and coherence modeling using recurrent neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32;1, 2018.

[24] N. MacDonald, L. L. Frase, P. Gingrich, and S. A. Keenan. The writer's workbench: Computer aids for text analysis. *Educational Psychologist*, 17:172–179, 1982.

[25] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014 (239):2, 2014.

[26] H. T. Ng, S. M. Wu, T. Briscoe, C. Hadiwinoto, R. H. Susanto, and C. Bryant. The CoNLL-2014 shared task on grammatical error correction. In *Proceedings of the Eighteenth Conference on Computational Natural Language Learning: Shared Task*, pages 1–14, Baltimore, Maryland, June 2014. Association for Computational Linguistics. doi: 10.3115/v1/W14-1701. URL https://www.aclweb.org/anthology/W14-1701.

[27] D. Nicholls. The cambridge learner corpus: Error coding and analysis for lexicography and elt. In *Proceedings of the Corpus Linguistics 2003 conference*, volume 16, pages 572–581, 2003.

[28] K. Omelianchuk, V. Atrasevych, A. Chernodub, and O. Skurzhanskyi. GECToR – grammatical error correction: Tag, not rewrite. In *Proceedings of the Fifteenth Workshop on Innovative Use of NLP for Building Educational Applications*, pages 163–170, Seattle, WA, USA. Online, July 2020. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/2020.bea-1.16.

[29] S. Prabhumoye, R. Salakhutdinov, and A. W. Black. Topological sort for sentence ordering. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 2020.

[30] S. D. Richardson and L. C. Braden-Harder. The experience of developing a large-scale natural language text processing system: Critique. In *Second Conference on Applied Natural Language Processing*, pages 195–202, Austin, Texas, USA, Feb. 1988. Association for Computational Linguistics. doi: 10.3115/974235.974271. URL https://www.aclweb.org/anthology/A88-1027.

[31] T. Tajiri, M. Komachi, and Y. Matsumoto. Tense and aspect error correction for esl learners using global context. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 198–202, 2012.

[32] R. E. Tarjan. Edge-disjoint spanning trees and depth-first search. *Acta Informatica*, 6(2):171–185, 1976.

[33] J. R. Tetreault and M. Chodorow. The ups and downs of preposition error detection in ESL writing. In *Proceedings of the 22nd International Conference on Computational Linguistics (Coling 2008)*, pages 865–872, Manchester, UK, Aug. 2008. Coling 2008 Organizing Committee. URL https://www.aclweb.org/anthology/C08-1109.

[34] S. A. Thompson and W. C. Mann. Rhetorical structure theory: A framework for the analysis of texts. *IPrA Papers in Pragmatics*, 1(1):79–105, Jan 1987. ISSN 2406-419X, 2406-4246. doi: 10.1075/iprapip.1.1.03tho.

[35] H. Yannakoudakis, T. Briscoe, and B. Medlock. A new dataset and method for automatically grading esol texts. In *Proceedings of the 49th annual meeting of the association for computational linguistics: human language technologies*, pages 180–189, 2011.

[36] Z. Yuan. Grammatical error correction in non-native english. Technical report, University of Cambridge, Computer Laboratory, 2017.

[37] Z. Yuan and M. Felice. Constrained grammatical error correction using statistical machine translation. In *Proceedings of the Seventeenth Conference on Computational Natural Language Learning: Shared Task*, pages 52–61, Sofia, Bulgaria, Aug. 2013. Association for Computational Linguistics. URL `https://www.aclweb.org/anthology/W13-3607`.