Technical Report

UCAM-CL-TR-894
ISSN 1476-2986

Number 894

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Issues in preprocessing current datasets for grammatical error correction

Christopher Bryant, Mariano Felice

September 2016

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

ISSN 1476-2986

Issues in Preprocessing Current Datasets for Grammatical Error Correction

Christopher Bryant Mariano Felice

Abstract

In this report, we describe some of the issues encountered when preprocessing two of the largest datasets for Grammatical Error Correction (GEC); namely the public FCE corpus and NUCLE (along with associated CoNLL test sets). In particular, we show that it is not straightforward to convert character level annotations to token level annotations and that sentence segmentation is more complex when annotations change sentence boundaries. These become even more complicated when multiple annotators are involved. We subsequently describe how we handle such cases and consider the pros and cons of different methods.

1 Introduction

Grammatical Error Correction (GEC) is the task of automatically correcting grammatical errors in English as a Second Language (ESL) learner texts. As with many other tasks in Natural Language Processing (NLP), this is achieved by means of large corpora of ESL learner texts. Unfortunately however, there are not only very few sizeable corpora available for this purpose, but there are also additional challenges involved in processing them.

In this report, we describe some of these challenges in the context of two of the largest datasets for GEC; namely the public First Certificate in English (FCE) corpus (Yannakoudakis et al., 2011) and the National University of Singapore Corpus of Learner English (NUCLE) (Dahlmeier et al., 2013) along with associated Conference on Natural Language Learning (CoNLL) test sets (Ng et al., 2013; Ng et al., 2014).

In particular, one issue concerns how to convert character level edit spans to token level edit spans. While human annotations are often made at the character level, GEC systems are often trained and tested at the token level; as such, there are problems if character edit spans do not map exactly to token spans. To give an example, suppose an annotator wanted to capitalise the first letter of the string *the*. To do this, they might have proposed the edit [$t \rightarrow T$]. This character level edit does not correspond to a complete token however, and so additional processing must be done in order to retrieve the intended token level edit: [*the* $\rightarrow$ *The*].

A related issue concerns sentence boundaries. In particular, there is no guarantee that a sentence boundary in an original sentence corresponds to an equivalent sentence boundary in a corrected sentence. Consider the text: "*In the morning. He slept.*". The most likely correction to this string would be to change the first full stop to a comma and then lower case the first letter of *He*; i.e. [*. He* $\rightarrow$ *, he*]. This means, however, that while the original text appears to consist of 2 sentences, the corrected text only contains 1. This poses a problem for sentence-based GEC systems which make the assumption that each input original sentence is a complete sentence.

In the rest of this report, we shall describe several important qualities of the data and also make recommendations on how to deal with any issues.

2 Data

2.1 FCE

The public portion of the FCE corpus is available online.[1] The data is not explicitly organised into training and test data, so we must first merge various folders to recreate it. Specifically, all the folders in the *dataset* directory containing the string *2000* should be merged as training data (1141 scripts in total from the year 2000), while the folder named *0100_2001_6* should be used as test data (97 scripts from the year 2001). These figures correspond to the descriptions of the data in Yannakoudakis et al. (2011) and the readme file bundled with the data.

Each file in the training and testing data contains two essays written by the same author on various topics. This data comes in an XML format that includes a lot of extra metadata, such as the age and nationality of the author. For the purposes of this report, we are only interested in the essay text between `<coded_answer>` tags, which can be further subdivided into paragraphs based on the `<p>` tags.

One feature of the FCE XML structure is that the edit tags are built directly into the essay text, as shown in Figure 1.

```
<p>
  This
    <NS type="AGV">
      <i>are</i>
      <c>is</c>
    </NS>
  a sample annotated FCE paragraph.
</p>
```

Figure 1: Example FCE XML Paragraph Structure

Here, `<NS>` denotes the start of an edit, `<i>` denotes the original string to be changed and `<c>` denotes the correction string. In this example, the error type has been labelled `"AGV"`, which is a verb agreement error. See Nicholls (2003) for a more complete list of error categories used in CLC data.

In all, there are 4 basic underlying types of XML edit within any given `<NS>` tag:

(a)	No `<i>` or `<c>`	: `<NS>`text`</NS>`
(b)	Only `<i>`	: `<NS><i>`text`</i></NS>`
(c)	Only `<c>`	: `<NS><c>`text`</c></NS>`
(d)	Both `<i>` and `<c>`	: `<NS><i>`text`</i><c>`text`</c></NS>`

Specifically, (a) denotes a mistake that an annotator identified but was unable to correct, (b) denotes an unnecessary word in the original text, (c) denotes a missing word from

[1]http://ilexir.co.uk/media/fce-released-dataset.zip

```
<p>
  I will wait at the
    <NS type="RN">
      <i>
        <NS type="S">
          <i>entery</i>
          <c>entry</c>
        </NS>
      </i>
      <c>entrance</c>
    </NS>
  .
</p>
```

Figure 2: Example of a Nested Edit in the public FCE data

FCE	None	`<i>`	`<c>`	`<i><c>`	`<NS>`	`<i><NS>`	`<c><NS>`	`<i><c><NS>`	Total
test	212	458	907	2931	51	4	0	216	4779
train	1445	4714	8985	30728	265	41	0	1893	48071

Table 1: Table showing the counts for the number of edits in the public FCE data sorted by XML structure. For example, there are 2931 edits in the test data containing both an `<i>` and `<c>` tag (i.e. substitution errors) while there are 216 cases where a similar substitution edit also contains a nested error. Here, `<NS>` denotes a nested error somewhere within the outermost error span.

the original text and (d) denotes a replacement of one word for another in the original text. Additionally, the CLC also allows nested edits (i.e. edits within edits), which means that there may be another `<NS>` tag nested within any of the above four types. Note that the nested edit will always occur inside an `<i>` or `<c>` tag if there is one present. An example of a nested edit is shown in Figure 2, where the string *entery* is first identified as a spelling error and corrected to *entry* and second identified as a replacement noun error and changed into *entrance*.

This nesting, while fairly rare, makes processing this data much more complicated, as there is no limit to the amount of nesting that can occur; there may be multiple nested edits of various depths inside another edit. See Table 1 for the counts of all the edits in the FCE data in terms of their XML structure. This table only counts the outermost `<NS>` tags so nested edits are not counted twice (or more). In total, roughly 5% of all errors in the FCE contain nested errors. While there are no instances of nested edits appearing inside a lone `<c>` tag anywhere in the public FCE (column `<c><NS>`), nested edits do rarely appear inside the `<c>` tag of substitution edits (column `<i><c><NS>`) and so it is not impossible that this may occur in other data. That said, it seems strange that annotators found errors within their own corrections, and so it is highly likely that any nested edit within a `<c>` tag is the result of annotator error.

2.2 NUCLE/CoNLL

Although NUCLE is free, it can only be obtained after submitting a signed license form to its authors (Dahlmeier et al., 2013).[2] The CoNLL-2013 and CoNLL-2014 test data,

[2] http://www.comp.nus.edu.sg/ nlp/conll14st.html

```
<P>
This are a sentence.
</P>
<MISTAKE start_par="0" start_off="5" end_par="0" end_off="8">
<TYPE>SVA</TYPE>
<CORRECTION>is</CORRECTION>
</MISTAKE>
```

Figure 3: Example CoNLL SGML Paragraph Structure

on the other hand, is freely available without a license.[3,4] All this data is available in 2 different formats: SGML and M2. The M2 format is a preprocessed version of the SGML data that has been cleaned up and tokenized. Since there is no cleaned up version of the FCE data, and in the interest of fairness, we only work with the SGML data.

Like the FCE data, there are a few additional metadata tags in the SGML files, but, for the purposes of this report, we are only interested in the essay text between `<TEXT>` tags. These essays can similarly be separated into paragraphs based on the `<P>` tags, with the added complication that some essays also contain `<TITLE>` tags which we also treat as a paragraph. Unlike the FCE data, however, edits are separate from the text body and are represented as a list between `<ANNOTATION>` tags, where each individual edit is enclosed by a `<MISTAKE>` tag. As such, Figure 1 would look something like Figure 3 in CoNLL SGML format.

Note that as the edits are separate from the text, the exact location of the edit is instead marked by explicit paragraph and character start and end offset positions. See Ng et al. (2014), Table 1, for the complete list of error categories used in the NUCLE framework. The original list of error categories was defined in Dahlmeier et al. (2013), but a few changes were made for the CoNLL shared tasks.

The same 4 basic edit operations available in the FCE data are also present in the CoNLL datasets (nested edits are not allowed) with the following differences:

1. When an annotator identifies but is unable to correct an error, the selected error span is labelled with the category `Um` (Unclear Meaning) and the `<CORRECTION>` label is (usually) left blank.

2. When an annotator wants to insert a missing word into the original text, they must select an adjacent word and repeat that word in the correction along with the missing word.

Regarding the latter, given the sentence *I want go home* into which an annotator wants to insert *to*, the annotator could either select *want* with the correction *want to* or *go* with the correction *to go*. This was done due to the limitations of the CoNLL annotation platform which required an edit to consist of at least 1 non-whitespace character. One consequence of this is that missing word errors look similar to substitution errors, at least at the character level, and so Table 2 just reports the total number of edits in each CoNLL dataset.

[3]http://www.comp.nus.edu.sg/ nlp/conll13st/release2.3.1.tar.gz
[4]http://www.comp.nus.edu.sg/ nlp/conll14st/conll14st-test-data.tar.gz

CoNLL	Total
Test 2013	3424
Test 2014 (0)	2397
Test 2014 (1)	3331
NUCLE	44912

Table 2: Counts for the total number of edits in several CoNLL SGML files.

Test data for 2014 features twice in this table because this was the first year that the data was annotated by 2 annotators. Note also that these figures represent the maximum number of annotations made by the annotator, including those that were subsequently pruned in the cleaner M2 version of the data. This is why the total edit counts reported in the literature may differ slightly between SGML and M2 files.

3 Preprocessing

The main aim of preprocessing is to extract the original and corrected versions of each paragraph in the input data along with the edits that transform the former into the latter. In the case of the FCE, we have to regenerate both the original and the corrected paragraphs, owing to the in-line edit format, but in the case of CoNLL, we already have the original text so only need to apply the character edits to produce the corrected text. Unsurprisingly, each dataset thus undergoes slightly different preprocessing.

3.1 Preprocessing - FCE

When regenerating the FCE essays, any edited text enclosed by `<i>` tags is used only in the original paragraph while any edited text enclosed by `<c>` tags is used only in the equivalent corrected paragraph. Any text that appears inside an edit but is outside an `<i>` or `<c>` tag is used in both sides, as this often indicates uncorrected erroneous text. Regarding nested edits, we only keep the outermost correction string for the corrected paragraph and ignore the other intermediate corrections. For example, from Figure 2, we would bypass the spelling error [*entery* → *entry*] and only keep the edit [*entery* → *entrance*].

One disadvantage of the FCE in-line edit format is that it is almost impossible to faithfully recreate the original text as it was intended using standard orthography. Consider the following sentence: `I want <NS><c>to</c></NS> go.`

This contains a missing word error, *to*, which an annotator subsequently corrected. Missing or unnecessary words, however, also affect whitespace, which is rarely considered part of an edit. As a result, omitting the *to* in order to recreate the original sentence results in: *I want␣␣go*, where there is double whitespace between *want* and *go*. Such a phenomenon can be more clearly seen when the missing or unnecessary word precedes, for example, a comma or other punctuation mark, as one side of the text will then contain that punctuation mark surrounded by whitespace; e.g. [*a b, c* → *a␣, c*].

Whilst this is a problem if we want to faithfully reproduce the properly formatted original or corrected text, it is less of a problem for the present work where we ultimately want tokenized text. As such, where there is unnecessary whitespace, we can simply replace all sequences of 2 or more adjacent whitespace characters with a single whitespace character, and where there is a floating punctuation mark, we need not do anything as this effectively constitutes some early tokenization.

It is worth noting, however, that this whitespace problem would become more of an issue if the text were annotated by multiple annotators, but that the current XML data format does not support this.

3.2 Preprocessing - CoNLL

Before applying the character edits to the original text in the CoNLL data, we first modify or prune those that meet certain undesirable criteria.

For example, we modify edits tagged `Um` (Unclear Meaning), which represent uncorrected errors, such that the correction string is the same as the original string. This is done because often the correction string of a `Um` edit is left blank by an annotator; this would otherwise be misinterpreted as a deletion by a system, but we do not want to simply delete any string that an annotator was unable to correct. We only keep such edits because they may still be useful for the purposes of error detection.

One error type that we prune completely from the CoNLL data is `Cit` (Citations), which is used to identify poor citation practice. This category has not only been used inconsistently in terms of whether the error is corrected or not, it is also very rare. Furthermore, it is debatable whether citation practice itself qualifies as a grammatical error as this would only really affect formal essay texts; other genres, such as fiction or correspondence, do not require citations.

Other groups of edits removed from the data include those that cross paragraph boundaries, those that select entire paragraphs, and those whose corrections include "...". In most cases, these edits are not grammatical errors, but rather annotator comments that suggest the whole essay or paragraph should be rewritten for various reasons. In the case of the ellipsis, some annotators occasionally used ellipses in the correction string to denote long sequences of unchanged text; a machine, of course, interprets this literally and unhelpfully attempts to replace the original text with ellipses.

Finally, another complication when processing CoNLL data is that a small number of edits overlap with others at the character level. This is normally the result of annotator error however, since nested edits are not allowed in the NUCLE framework. We handled these by simply ignoring any edits that overlapped with any previously seen edits. The new total error counts for each SGML file after edit pruning are shown in Table 3; as this table shows, this preprocessing mainly affects NUCLE.

CoNLL	Before	After
Test 2013	3424	3415
Test 2014 (0)	2397	2397
Test 2014 (1)	3331	3331
NUCLE	44912	43878

Table 3: Counts for the total number of edits in CoNLL SGML files before and after preprocessing.

4 From Characters to Tokens

One feature common to both datasets is that edits are defined at the character level. As most NLP tools require tokenized text however, we need to map these character spans to token spans. While this is straightforward in the majority of cases, there are several exceptions that complicate matters. Consider the following tokens and edits:

1. Token: *WORD.*	Edit: [. → ,]
2. Token: *Forest'view*	Edit: [*Forest'* → *Forest's*]
3. Token: *dancing*	Edit: [*ing* → *ed*]
4. Token: *klever*	Edit: [*kleve* → *clever*]
5. Token: *To*	Edit: [*T* → *to*]

In the first example, the annotator wanted to change a full stop into a comma. Unfortunately however, the tokenizer did not separate it from *WORD*, and instead considers *WORD.* (i.e. including the full stop) a complete token. This might be because the text is in upper case (as some essays in the FCE are) and so the tokenizer perhaps considers it an abbreviation. In any case, the tokenizer's mistake results in a character edit that does not map to a complete token.

A similar thing happens in the second example, where the missing whitespace between *Forest'* and *view* results in these 2 tokens being joined together; it is no surprise that the tokenizer is unable to separate them given that it is unreasonable to expect it to consider all the possible ways a string might be tokenized. In contrast, the human annotator *was* able to carry out a mental tokenization of this string, which regrettably led to the character-token alignment mismatch seen here.

While annotators were typically instructed to only select whole tokens when making corrections, in practice, this did not always happen. The third example is hence a case where the annotator only wanted to change the morphology of a given token, and thus only edited the relevant characters rather than the complete token.

Finally, the last 2 cases are examples of annotator error. Specifically, the annotators accidentally omitted a character from their edit spans but nevertheless provided a complete correction. Consequently, applying the character level edit in the fourth example produces the word *cleverr*, while applying the edit in the fifth example produces the word *too*; in other words, the omitted character is duplicated on the end of the correction string. The last case is especially noteworthy because it unintentionally produces a valid word.

4.1 Automatic Tokenization

In order to detect where character edits do not map to token edits, we first need to tokenize the text. We did this using spaCy[5] v0.101.0, a relatively new NLP library that not only includes a large number of NLP tools, but is also very fast and easy to use. One particular advantage of spaCy is that it keeps track of the character start positions of each token even after tokenization, which makes converting character-level edit spans to token-level edit spans a lot more straightforward.

Before determining whether a given character span mapped exactly to a token span however, we first stripped the leading and trailing whitespace, if any, from the character span; given that annotators sometimes omit characters from the peripheries of their edits, they also sometimes accidentally include unnecessary whitespace at the starts and ends of their edit. The total number of character-to-token mismatches is hence the total number of edits for which the character start and end offsets do not exactly match any of the token start and end character offsets determined by spaCy. The counts for these mismatches are shown in Table 4.

[5]https://spacy.io/

Dataset	**Mismatches**	**Total Edits**	**%**
FCE Test	20	4779	0.42%
FCE Train	250	48071	0.52%
CoNLL 2013	6	3415	0.18%
CoNLL 2014 (0)	8	2397	0.33%
CoNLL 2014 (1)	3	3331	0.09%
NUCLE	2580	43878	5.88%

Table 4: The total number of character-level edits that do not map exactly to token-level edits for various datasets.

The most surprising result from this table is that NUCLE is affected by mismatches more than any other dataset by a large margin. In fact while typically fewer than 0.5% of all edits in other datasets involve an alignment mismatch, this climbs to almost 6% in NUCLE. In particular, there are far more morphological annotations (e.g. [*ing* $\rightarrow$ *ed*]) in NUCLE than any other dataset, which perhaps indicates that some annotators either overlooked the need to annotate tokens or were otherwise unaware of this requirement. In any case, this makes the resolution of this problem much more significant as it affects more than just a tiny handful of edits.

4.2 Just Add Whitespace

One solution to the character-to-token alignment problem is to surround each edit with whitespace. If we make the assumption that annotators only edit complete tokens, as instructed, this implies there is a token boundary before and after each edit. As such, by artificially surrounding each edit with whitespace, we can ensure that the tokenizer tokenizes the text at these positions. This ultimately means that character edit spans always map to token edit spans.

Whilst this approach correctly handles the *WORD.* and *Forest'view* cases mentioned at the start of this section, it fails to handle the other 3 cases. Specifically, using this method, [*dancing* $\rightarrow$ *danced*], [*klever* $\rightarrow$ *clever*] and [*To* $\rightarrow$ *to*] are all respectively realised as [*danc ing* $\rightarrow$ *danc ed*], [*kleve r* $\rightarrow$ *clever r*] and [*T o* $\rightarrow$ *to o*]. Given that these last 3 cases are all instances of annotator error however, we might reasonably consider them an acceptable loss.

Unfortunately however, one side effect of inserting whitespace around each edit is that it fundamentally changes the original text for each annotator. This is not a problem if there is only one annotator, which is true for the vast majority of current datasets, but becomes more of an issue when there are multiple annotators. In particular, this affects the CoNLL-2014 test data which was initially annotated by 2 annotators for the shared task and subsequently annotated by a further 8 annotators in Bryant and Ng (2015).

To give an example of the problem, we can again consider the string *Forest'view*. While one official CoNLL annotator edited the substring [*Forest'* $\rightarrow$ *Forest's*], as previously discussed, the other annotator edited the whole string [*Forest'view* $\rightarrow$ *Forest's view*]. Given that we now insert whitespace around each edit, this means that the original text becomes *Forest'␣view* for the first annotator (i.e. with whitespace), but remains *Forest'view* for the second (i.e. without whitespace). Ultimately, this means we end up with 2 different tokenized versions of the same original text.

This is highly undesirable as we do not want the annotations of one annotator to affect the tokenization and annotations of every other annotator.

4.3 The CoNLL Approach

An alternative to adding whitespace around an edit is to instead grow its character span, if necessary, until it aligns with the nearest token boundary. To give an example, consider the token *dancing*, which an annotator wants to change into *danced* by means of the correction [*ing* $\rightarrow$ *ed*]. The character span of the edit is hence 4 to 7, while the character span of the token is 0 to 7; consequently, we must increase the range of the edit span to agree with the token span.

We can do this by calculating the difference between the edit start span and the token start span to learn the extent of the mismatch; in this case $4 - 0 = 4$. By subtracting this value from the edit start span, we hence obtain a new character edit span that corresponds to a token boundary. While this solves the character alignment problem, a second step is to also update the correction string. Specifically, as we have now increased the size of the edit span by 4 characters, we must also add these 4 characters to the start of the correction string. In this case, this means we add *danc* to the original correction string *ed*, to produce the expanded edit [*dancing* $\rightarrow$ *danced*]. Note that a similar process can be applied to growing the end of an edit span.

One rare complication to the above concerns a token that has been partially annotated more than once. For example, let us say the annotator wanted to change *dancing* into *Danced* by means of 2 edits: [*d* $\rightarrow$ *D*] and [*ing* $\rightarrow$ *ed*]. As neither of these edits maps to a complete token, growing the character span will produce 2 edits for the same token: [*dancing* $\rightarrow$ *Dancing*] and [*dancing* $\rightarrow$ *danced*]. In such cases, it is more complicated to recover the intended edit [*dancing* $\rightarrow$ *Danced*] and so we instead just keep the first edit.

The main advantage of this approach is that it correctly handles all the cases where annotators only edited morphology or capitalization. As this mainly only applies to NUCLE, it should hence be no surprise that this was the approach adopted in the CoNLL shared tasks where NUCLE was the official training corpus. The disadvantage of this approach, however, is that character span expansion can also have unintended consequences. For example, growing [*Forest'* $\rightarrow$ *Forest's*] in the string *Forest'view*, results in the correction [*Forest'view* $\rightarrow$ *Forest'sview*] which, unlike the whitespace-adding method, is not what the annotator intended. That said, at least this approach does not modify the original text in any way and is hence more compatible with multiple annotators.

4.4 Putting Everything Together

Thus far, we have mainly discussed how to align the character span of an original string with an original token. This is not the only alignment problem, however, and we similarly want to align the character span of a corrected string with a corrected token. Although the problem is the same, there is in fact a small difference between the two cases which means we proceed as follows.

Firstly, we make use of the CoNLL character expansion approach to ensure original character spans map to original token spans. This is important because, unlike using the whitespace expansion method, we want the original text to remain the same regardless of the number of annotators. This produces a tokenized original text where the original token edit spans are well defined.

Secondly, instead of processing the corrected text in the same way and then trying to

```
S Back to a hundred years ago , water was never a concern .
A 0 2|||Rloc-|||||REQUIRED|||-NONE-|||0
A 2 3|||Mec|||A|||REQUIRED|||-NONE-|||0
A 12 13|||Mec|||,|||REQUIRED|||-NONE-|||0

S While , today it is different .
A 0 1|||Mec|||while|||REQUIRED|||-NONE-|||0
A 1 2|||Mec|||||REQUIRED|||-NONE-|||0
```

Figure 4: An example of improper sentence alignment in NUCLE v3.2 (M2 format).

align it with the tokenized original text, we instead make a copy of the tokenized original text and simply apply the edits to that to produce the tokenized corrected text. This also saves having to tokenize everything twice and avoids tokenization inconsistencies.

One complication, however, is that the correction strings of the edits are usually not tokenized. As such, if we applied them directly, we would effectively be inserting fragments of untokenized text into tokenized text. To resolve this, we instead tokenize the edit fragments individually before applying them. We can then POS-tag and parse the corrected text without needing to call the tokenizer again.

The output from this processing hence produces parallel original and corrected tokenized texts where all the edit spans always only map to complete tokens.

5 From Paragraphs to Sentences

Another important aspect of processing GEC datasets is sentence segmentation. Up until now, we have been operating at the paragraph level, rather than the sentence level, simply because paragraphs are the minimal unit of text in the input files. While there is no strong empirical reason why we should convert paragraphs into sentences, other than the fact that this has long been a convention in NLP, we nevertheless also attempt to carry out sentence segmentation here.[6] Unfortunately however, this is not straightforward in GEC given that annotators sometimes edit sentence boundaries.

To give an example of the problem, consider Figure 4, which contains an extract from the latest version of NUCLE (v3.2) in M2 format. Given that the annotator wanted to change the full stop at the end of the first sentence into a comma, and hence combine these two sentences, it seems inappropriate to list them separately. Instead, they should be combined such that the original and corrected text constitute at least one complete sentence. If this is not done, a system trained on this data might learn that it is acceptable for some sentences to end with a comma, which seems undesirable.

That said, we are only able to combine the above sentences in this way because we already know the corrected sentence, which is not the case in a realistic grammar checking scenario. In that instance, with only the original sentence to work with, it might be beneficial for a system to entertain the possibility that even a sentence final full stop can change.

Ultimately, whether we should sentence tokenize based on the original, corrected or

[6] Although most parsers require sentence tokenized input before they can parse, spaCy is unique in that it can parse whole documents at once and then use this information to determine sentence boundaries afterwards. The authors of spaCy argue that this is more reliable than other sentence segmentation methods which typically rely on punctuation and upper casing to predict sentence boundaries.

Index	0		1		2		3		4		5		6		7		8		9		10		11		12		13		14		15
Orig		A		B		C		.		D		E		,		F		G		.		H		I		.		J		.	
Cor		A		B		.		D		E		.		F		G		,		H		I		?		J		K		.	

Table 6: Mock paragraph structure containing multiple sentences, where each letter denotes a token in those sentences. The challenge is to determine how the sentences should align.

both texts remains a contentious issue. It might even be most desirable to bypass the problem altogether and simply operate at the paragraph level instead.

5.1 Sentence Alignment

In fact the difficulty of the sentence segmentation problem is further compounded by the fact that such sentences also often occur inside a larger paragraph where other sentence boundaries may change. Ultimately, this becomes an alignment problem. While you could use sentence alignment techniques from the field of Machine Translation to solve this problem (for example, see Smith et al. (2010)), our situation is much simpler in that we: a) do not have to deal with different languages, b) do not have to account for sentence reordering or other effects, and c) already have the exact mapping that "translates" Paragraph A into Paragraph B. As such, we can use a more heuristic approach.

It is easiest to explain this approach by means of an example. Table 6 hence shows an example of a typical paragraph containing several sentences, where the letters A-K represent tokens in those sentences. If there was only one sentence on either the original or corrected side, we would not need to do anything as the sentences would already be minimally aligned. This is not the case here however, and so we must work out the proper alignment.

Firstly, as sentence segmentation of the corrected paragraph is likely to be more robust than for the potentially ungrammatical original paragraph, we choose the corrected paragraph sentence boundaries, as determined by spaCy, as our starting point. In Table 6, this means we consider everything between the range of 0-3 to be the first corrected sentence. We then undo any of the edits that fall exclusively within this range to obtain an equivalent original sentence boundary. In this case, this means we add C back into the sentence to obtain a span of 0-4 for the original. If it is then true that spaCy also detected a sentence boundary at position 4 in the original paragraph, we declare these sentences aligned and move on to the next case, as we do here.

Having aligned the previous sentence, we now know that the next sentence starts at position 4 in the original paragraph and position 3 in the corrected paragraph. As the next sentence boundary in the corrected paragraph is 6, we also know that the next candidate corrected sentence spans tokens 3-6. As before, we again undo the edits in this range to obtain an equivalent original sentence boundary position, which we this time determine to be located at position 7. This time, however, no equivalent sentence boundary was detected in the original sentence at position 7 and so the sentences do not align.

Instead, we look ahead to the next corrected sentence boundary, which is at position 12. Note that although there is a sentence boundary at position 10 in the original paragraph, it similarly does not have an equivalent sentence boundary in the corrected paragraph, so we do not want to align the sentences here either. If we now consider our corrected sentence to span from 3-12, we can again calculate the corresponding original sentence boundary by undoing the edits and determine it to be at position 13. This has also been identified as a sentence boundary in the original paragraph and so we align the sentences.

This second case is more complicated than the first, in that the minimal alignment actually consists of 2 sentences. Additionally, it also involves a full stop becoming a question mark, which shows that sentence final punctuation edits need not necessarily indicate sentence boundary changes. Finally, when we reach the last corrected sentence boundary at position 15, we need not do any more processing, as with only one sentence to align, whatever is left will align by definition.

After each alignment, a final step is to update the token edit spans for each new sentence. Previously, all the edits were defined in terms of tokens in a paragraph, so we must take the new sentence segmentation into account and redefine the spans in term of tokens in a sentence.

5.2 Multiple Annotations

One complication to the above is that sentence segmentation becomes more difficult when a text is annotated by multiple annotators. Since there is no guarantee that all annotators will agree on the same sentence boundaries, the tokenization may be different for each annotator. For this reason, in order to keep everything consistent across all annotators, we only tokenize on sentence boundaries upon which all annotators agreed. The disadvantage of this is that globally optimum sentence segmentation might change with each additional annotator, but there is no way to avoid this unless we back off to paragraphs.

6 Conclusion

We have shown that it is not easy to process the largest publicly available datasets for GEC and that different implementation details can result in slightly different versions of the same corpus. Without a standard way of processing the data, this inconsistency ultimately means different approaches to GEC using the same data may not be strictly comparable.

Additionally, while character edit spans that do not exactly map to token edit spans typically make up less than 0.5% of the data, we found that this rises to almost 6% in the NUCLE corpus. This shows that the problem sometimes affects more than just a negligible amount of edits, which is especially important given that NUCLE was used as the official training corpus of the CoNLL shared tasks. We resolved these problems by expanding the range of the character edit to align with a complete token in the original text, and by adding additional whitespace around any correction string in the corrected text.

Given that sentence fragments are also considered a type of grammatical error, we also showed that sentence segmentation of GEC text is more difficult when annotators change sentence boundaries. This means we cannot rely on the sentence boundaries in the original text alone for sentence segmentation, but must also make use of the corrected sentence boundaries to extract only complete sentences from the data. Of course the corrected sentences are unknown in a realistic grammar checking scenario however, and so it may be easier to model at the paragraph level instead.

References

Christopher Bryant and Hwee Tou Ng. 2015. How far are we from fully automatic high quality grammatical error correction? In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 697–707, Beijing, China, July. Association for Computational Linguistics.

Daniel Dahlmeier, Hwee Tou Ng, and Siew Mei Wu. 2013. Building a large annotated corpus of learner english: The nus corpus of learner english. In *Proceedings of the Eighth Workshop on Innovative Use of NLP for Building Educational Applications*, pages 22–31, Atlanta, Georgia, June. Association for Computational Linguistics.

Hwee Tou Ng, Siew Mei Wu, Yuanbin Wu, Christian Hadiwinoto, and Joel R. Tetreault. 2013. The CoNLL-2013 shared task on grammatical error correction. In *Proceedings of the Seventeenth Conference on Computational Natural Language Learning: Shared Task*, pages 1–12, Sofia, Bulgaria. ACL.

Hwee Tou Ng, Siew Mei Wu, Ted Briscoe, Christian Hadiwinoto, Raymond Hendy Susanto, and Christopher Bryant. 2014. 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, USA. ACL.

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

Jason R. Smith, Chris Quirk, and Kristina Toutanova. 2010. Extracting parallel sentences from comparable corpora using document level alignment. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 403–411, Los Angeles, California, June. Association for Computational Linguistics.

Helen Yannakoudakis, Ted Briscoe, and Ben Medlock. 2011. 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, Portland, Oregon, USA, June. Association for Computational Linguistics.