A Perl program for sentence splitting using rules

Paul Clough {cloughie@dcs.shef.ac.uk}
University of Sheffield

April, 2001

Abstract

Over the past few years, there have been many approaches to segmenting text (in particular English) into sentences. Various academics have argued the linguistic composition of sentences; others have used data-driven approaches to derive heuristics that capture certain characterstics of a sentence. In this report I do not claim to have created a new approach to sentence splitting, but simply used existing methods to create a simple Perl program to do the task. The reason for my investigation into sentence splitting was the need for a Perl sentence splitter and the inability of an existing Perl CPAN module to perform the task with the accuracy I required. I have also attempted to address the problem of evaluating a sentence splitter by using the British National Corpus (BNC) version 1.0 and the Brown corpus (from the PTB3) as my "gold standards". I use observations from the BNC to validate the heuristics used in the sentence splitting algorithm and to empirically investigate some characteristics of sentences. The simple splitting algorithm described in this report achieved correct disambiguation of 98.59% of '.'s, 98.99% of '!'s and 99.24% of '?'s on approximately 750,000 sentences (approximately 13.5 million words) using files from the A sub-section of the BNC. The program also achieved correct disambiguation of 97.61% of '.'s, 97.80% of '!'s and 98.87% of '?'s on approximately 52,000 sentences in the Brown corpus (approximately 1 million words).

Contents

1 Introduction

My interest in sentence splitting was intially brought to light when I wanted to calculate average sentence length as part of an authorship style analysis. I needed a simple sentence splitter that could be called from a Perl program. The first port of call I tried was a web search[1] which brought to my attention a message from Tony Rose posted to the `corpora` dicussion group regarding the heuristics for a simple sentence splitter. Replies to this request came from familiar figures in NLP including Ted Dunning, Michael Barlow and Gregory Grefenstette. The final outcome of this request was a Perl CPAN[2] module called `Text::Sentence` written by Tony Rose and Ave Wrigley from Canon Research Centre Europe (CRE).

On using the Perl CPAN module, I found a number of limitations because of its naive approach to the problem (e.g. sentence breaking after honorifics). Therefore I decided to investigate further sentence boundary detection and to implement my own algorithm to improve on the existing CPAN module. The result is this report and a simple rule-based Perl program evaluated on the British National Corpus (BNC) and the Brown corpus (from the Penn Treebank release 3). The Perl CPAN now offers a further sentence splitting module `Lingua::EN::Sentence` which is not evaluated in this report.

Accurate sentence splitting is an important building block of many NLP systems. For example, most Part of Speech taggers such as Brill's [1] require input in the form of one sentence per line. The problem has, by and large, been solved with solutions offering high degrees of accuracy across a variety of texts. One of the themes to come from the `corpora` discussion on sentence splitting was that building a generic rule-based sentence splitter was relatively easy and effective, although results on texts from a specific domain could be poor. The thoughts seemed to be that to produce a sentence splitter with high accuracy, samples of texts from the domain should be analysed to detect idiosyncracies that would cause a general-purpose sentence splitter to fail. Modifications could then be carried out on the sentence splitter to deal with these irregularities and improve accuracy. However my aim was soley to build a simple, general-purpose sentence splitter that anyone could take and modify to suit their own requirements. Of course one can use an adaptive algorithm that can be trained on a sample corpus using features within the text, but I wanted to create a program that could be used immediately on a document and without the need for training examples. One reason for analysing the BNC is that the corpus contains a wide range of domains e.g. narratives, newspapers, books, etc. and by building a splitter based upon heuristics derived from the BNC it is possible that the splitter would then be suitable for a wide range of textual domains. Evaluation on the Brown corpus (what appears to be a popular corpus for sentence splitter evaluation) makes my results comparable with other evaluations on this corpus.

[1] Google - http://www.google.com
[2] Comprehensive Perl Archive Network - http://www.perl.com/CPAN-local/index.html

The aim of creating a Perl sentence splitting program was not to achieve just high accuracy (which is still of course a desireable characteristic), but to create a program simple enough to be modified by potential users and test heuristics suggested by a range of people for applicability to BNC and Brown texts. I also wanted a program that would work with reasonable accuracy without the requirement of prior processing on the input text such as Part of Speech tagging.

There are areas that I do not consider in this report that would be useful to inviestigate at a later stage including the pre-processing of text e.g. removing HTML, SGML, XML tags, reading Postscript, Microsoft Word, Adobe PDF files etc.

2 Background

Various approaches have been adopted for tackling the problem of sentence boundary identification (or sentence splitting). These range from trainable methods such as Maximum Entropy models [10] and [8], statistical classification trees [11] and neural networks [9], to simpler rule-based approaches such as [4] and [7]. Early work by Mikheev [7] was used to build the simple rule-based splitter developed in this paper. Finally, it is also worth mentioning Stevenson et al. [12] who describe the problems of sentence boundary detection in text from automatic speech recognition systems. They make clear the distinction between finding sentence boundaries in texts with no puncutation or capitalisation versus those with. It is the latter they call sentence splitting (or punctuation disambiguation) and that is the problem investigated in this report.

Segmenting English text into sentences "a string of words satisfying the grammatical rules of a language" cf. Wordnet 1.6) is in most cases a trivial task. Both Manning [6] and Mikheev [7] describe the base-line segmentation algorithm as simply looking for something ending in either a period, exclamation mark or question mark `[.!?]`. Mikheev adds to the base-line algorithm by suggesting that the `[.!?]` should be followed by whitespace, quotes or parenthesis and the next word begin with a capital letter. In a regular expression, this would look something like `[.!?][ ()"]+[A-Z]`.

This assumption gets you quite a long way and Riley [11] comments that in the Brown corpus about 90% of periods are sentence boundaries (10% at the end of abbreviations and about 0.5% as both). From my own analysis, I find approximately 88.3% of periods are sentence boundaries in the Brown corpus and approximately 89% in the BNC. So this seems a reasonable assumption to make. However, the main exception to this is that not all periods mark the end of a sentence, they can also be used with abbreviations. For example in the sentence "The train left Nottm. at 3 p.m.", the location Nottingham has been abbreviated to Nottm. and periods used to express 3 o'clock in the afternoon not morning (p.m.). Periods occur in the middle and end of the sentence.

One can think of sentence boundary segmentation as a classification problem. The task

is to determine whether an instance of `[.!?]` is a sentence boundary or not (also called sentence boundary disambiguation). Further issues arise when segmenting text into sentences including: 1) how to handle punctuation such as ':', ';' and '–' which can also be used to indicate sentence breaks, 2) how to deal with direct quotes and characters occuring before a sentence start character or after a `[.!?]`, and 3) sometimes abbreviations occur at the end of a sentence not in the middle (e.g. "Complaints should be sent to Dr. White" - abbreviation occurs in the middle of the sentence and "The train leaves at 3 p.m. It will stop at Ipswich." - abbreviation occurs at the end of a sentence. Mikheev [7] describes the problems of finding abbreviations and named entities as a fundamental part of text normalisation and sentence boundary identification. He claims to achieve accuracy of 99.3%-99.7% on sentence boundary disambiguation on articles from the Brown Corpus and newswire articles from the New York Times using heuristics derived from corpus analysis.

3 Analysis of the BNC

3.1 Introduction

The BNC [2] is a general-purpose corpus comprising of samples generally no longer than 45,000 words. The corpus is not restricted to any particular genre or register and contains representatives from 10 domains including imaginative, arts, natural science and world affairs. The corpus also contains written and spoken texts, all from the English language. In this report on sentence splitting, I only consider written texts. The 4,124 BNC texts (3,209 written and 915 spoken) are all encoded in SGML and have been Part-of-Speech tagged using the Lancaster CLAWS[3] (Constituent Likelihood Automatic Word-tagging System) tagger [3]. This also includes sentence boundaries which is essential for our analysis. The written section of the BNC contains 5,188,373 sentences and 89,740,544 words. I chose to use the BNC because as to my knowledge no analysis of sentences has yet been undertaken on this corpus.

3.2 Processing files from the BNC

BNC files are annotated in SGML, aiding the extraction and analysis of sentences. At the start of the BNC file is a header that begins with `<bncDoc>`. It is worth pointing out that the actual text of the file starts with the tag `<text` or `<stext`. The latter tag indicates a spoken text and the prior a written one. In this study I concentrate only on written text and therefore this tag is a useful indicator of the file type and basis on which to ignore the file. For example, in `BNCsentences.pl` the condition `if ($line =~ /^<text/)` is used to check for the file type (assuming that the BNC file is read a line at a time). The file header can be ignored and analysis started at the beginning of the text.

[3]For more CLAWS information - http://www.comp.lancs.ac.uk/ucrel/claws/

Sentences in the BNC are indicated using `<s n=val>` where val referred to by n is the sentence number in the file, e.g. `<s n=041>`. BNC annotation assumes that a document consists of a sequential series of sentences (which may also be divided into paragraphs). Sentences can be extracted from a BNC file by continually looking for a `sentence tag-anything-sentence tag OR end of file combination`. For example the following is a sentence in file `A00` of the BNC:

```
<s n=141>
 <w NP0>ACET <w VBZ>is <w AV0>currently <w AJ0-VVG>offering <w NN2>speakers
 <w TO0>to <w VVI>inform<c PUN>, <w VVB>motivate<c PUN>, <w NN1-VVB>train <w CJC>and
 <w NN1-VVB>support<c PUN>.
<s n=142>
```

Using the Perl `BNCsentences.pl` program, the following line of code will extract the above sentence form:

```
while($text =~ /(<s n=[0-9]+>.+?(?:(?=<s n=[0-9]+>)|(?=<\/bncDoc>)))/gs )
```

One experiment I consider is counting how many `[.!?:]` are actually end of sentence markers. This can be found using the condition: `if ($sentence =~ /.*([\.!?:]).*/ )` which says if the sentence ends in [.!?:] then count it, otherwise the count for `other` is incremented. Determining how many overall occurences of these punctuation characters exist overall, however, will also involve counting the number of these within the exctracted sentences. This can be achieved by simply using: `while ($sentence =~ /([\.!?:])/gs)`.

To remove the SGML tags and extract just the plain text, something like this can be used:

```
$sentence =~ s/<.+?>//g
```

It is also worth mentioning that there may also be SGML entity references in the text such as `&pound` (used to represent the £symbol) which you may or may not want to remove (and replace with a space character). These always begin with `&` and can be removed using something like: `$sentence =~ s/&[a-z]+[;]?/ /g`. It is worth remembering that some of these are `&mdash` which are used to separate words (may be important to you if you are counting words and want to consider something like *co-operation* as a single word and not two.

3.3 Analysis of sentence splitting rules on the BNC

3.3.1 Are [.?!:] used often at the end of a sentence?

First, let's look at the assumption that [.!?:] can be used to signal the end of a sentence in written English (cf. Hurford [5]). Table 1 shows the proportion of sentences in the BNC that end with either the punctuation symbol [.!?:] or something else (I call this `other`). Note I assume here that ':' can mark the end of a sentence. Table 1 shows the proportion of [.!?:] that are sentence breaks out of the total number of occurrences.

EOS marker	#sentences ending with EOS marker (% of total)	#EOS markers (% of total)
[.]	4,142,585 (80.1%)	4,637,441 (89.3%)
[!]	77,311 (1.49%)	85,116 (90.8%)
[?]	228,275 (4.41%)	248,098 (92.1%)
[:]	237,761 (4.61%)	271,109 (87.7%)
[other]	485,714 (9.39%)	
TOTAL	5,171,646	

Table 1: End Of Sentence (EOS) character analysis

The total number of sentences in Table 1 does not agree with the BNC reference manual (this quotes the number of `<s>` units as 5,188,373 which equates to a difference less 16,727 or 0.3%). I can only assume that my program is in error. However 0.3% difference overall is not particularly great.

What Table 1 does highlight is that a high number of sentences end with a period (80.1% of all sentences and 89.3% of all periods are full stops). Therefore the assumption that periods are full stops is valid, although on a per text basis, the error may be greatly amplified (i.e. a text with lots of abbreviations not at the end of sentences and before proper names e.g. honorifics). The other interesting points to notice are that almost 9.5% of sentences end with something other than [.:!?]. On further analysis, most of these tend to be either lists where each item is considered a sentence in CLAWS (even though it most likely does not begin with a capital letter), headings (e.g. In the UK, Letters to the Editor), addresses (e.g. Mildmay Hospital, Hackney Road, London E2 7NA Hospice care 071 739 2331), errors in the original text (e.g. "I cannot sleep at night my heart is weak my feet are heavy" - the full stop is missing), direct quotes with no [.!?] end-of-sentence marker (e.g. "Natural and more than natural, beautiful and more than beautiful, strange and endowed with an impulsive life like the soul of its creator") or errors in the CLAWS tagging. For example the following example shows a list where CLAWS has broken not just the items into sentences, but also the item numbers:

```
</item>
<label>
<s n=348>
 <w CRD>4<c PUN>.
<s n=349>
 <w DTQ>What <w VBB>are <w AT0>the <w NN2>conditions <w TO0>to <w VVI>Gift
 <w NN1>Aid<c PUN>? </label>
<item>
<list>
<label>
<s n=350>
<c PUL>(<w ZZ0>i<c PUR>) </label>
<item>
<p>
<s n=351>
<hi r=it> <w NN1>Cash <w AV0>only<c PUN>. </hi>
<s n=352>
```

EOS marker	Average sentence length (words/sentence)
[.]	19.93
[!]	11.50
[?]	11.04
[:]	21.94
[other]	6.10

Table 2: Average sentence lengths in the BNC

The number of `[.!?:]` and `other` sentence boundaries are far less than that for periods, although a high proportion of all '!', '?' and ':' occur at the end of sentences. The result of this observation is that generally the assumption of `[.!?:]` signalling the end-of-sentence is valid.

3.3.2 The average sentence length

The next experiment considers average sentence length. This is calculated by dividing the total number of words in the BNC by the total number of sentences ended with `[.!?:]` and `other`. Of course the mean value does not tell us anything about deviation, but forms the basis for further work on sentence length. The number of words in a sentence can be obtained using the following Perl expression: `@words = split(/\s+/, $sentence);`. This creates an array of words. Of course there is a slight contradiction here with the removal of SGML tags as described previously because words such as *co-operation* would be considered two separate words where previously united as one via a hyphen. The results are shown in Table 2.

The results from Table 2 are as I would have expected where perhaps sentences ending in '?' or '!' signal a question or an exclamation. These appear to be generally shorter than those ending in '.' or ':'. It is worth mentioning that this is only an average across the whole BNC. One might find that newspaper texts have a higher average than children's stories. Therefore these results are useful but may not be applicable to each individual text. The shortest average length is 6.1 words which of the `other` category which probably reflects the fact that many of these sentences are headings or items from a list.

3.3.3 Sentences beginning with a capital letter

How many sentences begin with a capital letter? One grammar book I consulted expressed a sentence in written English as *"a sentence begins with a capital letter and ends with either a full stop (period), question mark (query) or an exclamation mark"* [5]. Whether a sentence begins with a capital can be easily found using a Perl regular expression such as: `if ($sentence =~ /^\s*[A-Z].*/ )`. If the condition is true then a count of starting capitals can be incremented. Table 3 shows the results from this experiment.

EOS marker	#sentences beginning with capital/#total sentences	(% of total)
[.]	4,055,903/4,142,585	(98%)
[!]	76,313/77,311	(99%)
[?]	225,355/228,275	(99%)
[:]	228,374/237,761	(96%)
[other]	399299/485,714	(82%)

Table 3: Sentences beginning with a capital letter

The results from Table 3 are as one might expect given the grammatical definition of sentences taught to learners of the English language. That sentences begin with capitals is a somewhat valid assumption. Those which do not begin with capitals include items of a list and numbers at the beginning of sentences. Assumming that if a consecutive sequence of words ends with `[.?!:]` and begins with a capital letter is a sentence is justified from this experiment.

4 The Rule-Based algorithm

4.1 Introduction

One approach to segmenting a text into sentences is to capture heuristics in the form of rules which can be applied sequentially to a text to determine suitable sentence boundaries. This is the approach Mikheev discusses in [7]. To begin with, let's consider the base-line approach as suggested by Mikheev - [.!?] followed by whitespace followed by a capital letter. As we have seen from Section 3.3.1, it is a reasonable to assume that [.!?:] do occur at the end of sentences. However, there are many occurrences of these punctuation marks which are not sentence terminals but part of abbreviations instead. Periods tend to be the most problematic to deal with, but from the BNC, using the base-line assumption we only have to deal with 10.7% of cases where periods are not at the end of a sentence. Mikheev suggests the following decision tree for period disambiguation:

```
Assume we consider: <previous word>[.]<next word>

if (the previous word an ABBREVIATION) then
  if (the next word is CAPITALIZED) then
    if (the CAPITALIZED word is a PROPER NAME) then
      <undecided>
    else
      <End of sentence>
    endif
  else
    <Internal period>
  endif
else
  if (the next word is CAPITALIZED) then
    <End of sentence>
  else
    <undecided>
  endif
```

```
endif
```

Mikheev descibes each one of these conditions in turn together with some examples. In summary, there are conditions when the periods are undecided or ambiguous, and three cases when the decision is unambiguous (although actually finding abbreviations and capitalised words is still ambiguous). Mikheev decribes four tasks to the problem of sentence boundary disambiguation:

- identify abbreviations and ordinary/common words in the text;
- disambiguate ambiguously capitalised words;
- assign unambiguous sentence boundaries;
- in the case when an abbreviation is followed by a proper noun, disambiguate the sentence boundary.

From a reply on the `corpora` discussion list to Tony Rose's original question, Ted Dunning and Michael Barlow offered the following heuristics for sentence splitting:

- Sentence boundaries occur at one of [.!?];
- In mixed case texts, periods followed by whitespace followed by a lower case letter are not sentence boundaries;
- Periods followed by a digit with no intervening whitespace are not sentence boundaries;
- Periods followed by whitespace and then an upper case, but preceded by any short list of titles are not sentence boundaries;
- Periods internal to a sequence of letters with no adjacent whitespace are not sentence boundaries (e.g. www.dcs.shef.ac.uk);
- Periods followed by certain kinds of punctuation (notably commas and more periods) are probably not sentence boundaries;
- Sentence boundaries can also occur at [“ ’ - – :];

4.2 A description of splitter.pl

To test the heuristics mentioned in Section 4.1 and investigate further the observations made by Mikheev but with respect to the BNC not the Brown corpus, the program `splitter.pl` was written. In this section, I describe the basic assumptions behind the program and observations from the BNC to support these.

The program executes the following algorithm to determine sentence boundaries (expressed in a form of pseudo-code):

```
begin splitter.pl

load in a list of common words (splitter.dict)
load in a list of common abbreviations (splitter.abv)

while (extracting potential sentences using base-line conditions: letter/number-[.!?:]-whitespace-(capital OR EOF))
loop

  if (punctuation is [.]) then

    get previous word before [.] - get everything from [.] to last space
    get next word after [.] - get everything from [.] to next space or EOF
    get sentence length

    \# check for abbreviation
    if ((previous word is all consonants AND not all capitalised AND contains no lower case 'y') OR
        (previous word is a span of single letters followed by periods) OR
        (previous word is a single letter -- except I) OR
        (previous word is a common abbreviation)) then

        \# check whether abbreviation is in middle or end of sentence

        if (next word is common word and pos of  abbreviation > 6 words from sentence start) then
          <sentence_break>
          reset sentence length to 0
        else
          <not_sentence_break>
        end if
    else
        <sentence_break>
        reset sentence length to 0
  else
    <sentence_break>
    reset sentence length to 0
  end if

end loop

end splitter.pl
```

So let me expand upon this pseudo code definition of the algorithm and explain the reasoning behind it. The first steps are to load in a list of common words and abbreviations. The common word list is a copy of the UNIX dictionary `/usr/dict/words`. This contains 25,143 words (one-per-line separated by a line break) with 4,974 names and abbreviations beginning with a capital letter and 20,169 common words (one later experiment could be to find the common words from the BNC rather than using `/usr/dict/words`). The abbreviation list was created by extracting words from 4 different corpora of the form '*.'. This resulted in the following:

- `39` words from CIDE (Cambridge International Dictionary of English);
- `89` words from CELEX [4];
- `89` words from LDOCE (Longman Dictionary of Contemporary English);

[4] CELEX, The Dutch centre for lexical information: http://www.kun.nl/celex/

- 46 words from Wordnet 1.6.

Resulting in 245 merged abbreviations, the list was edited (duplicates and derivations removed) and I added a number from the paper by Grefenstette [4] and my own resulting in 152 abbreviations. Again, it may be interesting to expand these and extract abbreviations from the BNC and other corpora. The rules used to identify abbreviations, however, should cover most forms of abbreviation, the list simply being used to cover exceptional cases that are not captured by the abbreviation heuristics.

In this program, the entire text is first loaded into a Perl variable (`$sentence`). A regular expression is then used to extract all *potential* sentences using the base-line assumption that a sentence can begin with a letter or number followed by [.!?:], whitespace and then a capital or number. This is encapsulated in the following regular expression:

```
$text =~ /([\'\"‘]*[({[]?[a-zA-Z0-9]+.*?)([\.!?:])(?:(?=([([{\"\'‘)}\]<]*[ ]+)[([{\"\'‘)}\] ]*
([A-Z0-9][a-z]*))|(?=([()\"\'‘)}\] ]+)$))/gs
```

Although at first glance this may appear confusing, this expression attempts to deal with intervening punctuation such as brackets and quotations that would otherwise throw off the base-line splitter. Using regular expression bracketing (‘(’ and ‘)’), parts of the sentence can be caught and used in the rest of the algorithm. Starting from the left and counting to the right (not including the brakets before ‘?=’ and ‘?:’) and using the standard Perl notation for paranthesis capturing ($1, $2..), this regular expression captures the following:

- `$1` – the complete sentence including beginning punctuation and brackets.
- `$2` – the punctuation mark - either [.!?:].
- `$3` – the brackets or quotes after the [!?.:]. This is non-grouping i.e. does not consume.
- `$4` – the next word after the [.?!:].This is non-grouping i.e. does not consume.
- `$5` – rather than a next word, it may have been the last sentence in the file. Therefore capture punctuation and brackets before end of file. This is non-grouping i.e. does not consume.

Once the sentence and surrounding text have been captured, the algorithm attempts to establish whether the [.?!:] is a sentence break or not. The first condition is on the punctuation found. If it is [?!:] then we assume this indicates a sentence break. Of course, there may be exceptions to this as highlighted by Table 1.

If the punctuation mark is [.] then further analysis is required. This could be part of an abbreviation or an end-of-sentence marker. If it is an abbreviation then it may also fall at the end of a sentence and not just within the sentence. I use the conditions Mikheev suggests for identifying an abbreviation:

1. The previous word is all consonants and not all capitalised and contain no lower case 'y' – cf., vs. or Dr.
2. The previous word is a span of single letters followed by periods, – e.g., p.m. or A.A.A.
3. The previous word is a single letter (except I) – C.
4. The previous word is in the common abbreviations list – etc. or hon.

There are always exceptions to the rule (e.g. Oz. – abbreviation for Australia), but this will cover most of the more frequent ones. It may be worthwhile exploring further how abbreviations could be identified. If any of the previous conditions are fulfilled, then we assume the period is not an end-of-sentence marker. Of course the problem still arises that perhaps this abbreviation occurs at the end of a sentence. The next condition tries to overcome this by assuming that if the next word after the period is a common word (e.g. "the", "as", "a" etc.) then the end of the sentence has been reached. Obviously the next word could be a common word and a proper name and therefore this assumption would be wrong (e.g. Mr. Pan).

Just considering the first item of the previous list, I looked at common words in the `splitter.dict` file to see how many common words would also be caught and assumed to be abbreviations. I found that out of the 20,169 common words there were 88 which consisted of all consonants. These included words such as rhythm, sly, fly, spy, shy, lynx, lynch, nymph and try. These also included single letters of the alphabet (excluding vowels) and numbers (1st to 10th). Therefore with these removed, the number reduces to 36. All of these contain a 'y' except the word "ppm". I am not sure whether this is a word or an abbreviation (it is part of my abbreviation list). This results in just 1 common word being mis-classified as an abbreviation.

Alternatively, if we consider the names and abrreviations from `splitter.dict` then out of 4,974 words, 228 contains all consonants. From these, 37 are not all uppercase and 21 are abbreviations (the rest are names e.g. "Amy"). Out of these 21, none contain a lower case 'y' implying that this heuristic may be successful for extracting abbreviations and not common words (at least for words in `/usr/dict/words`). One must also consider that this assumption is made upon the basis that many abbreviations do not contain vowels.

If the previous word is not an abbreviation, then the period is considered a full stop marking the end of a sentence. If it is an abbreviation the next condition checks to see whether the abbreviation is likely to be at the end of a sentence or in the middle. If in the middle somewhere then it is not considered as a sentence terminal. To determine this, the condition checks to see whether the next word is a common word e.g. "the", "an" etc. If the next word is common, then it is assumed that the period marks a sentence terminal. For example one could have "He lives at Highfield St. This is a nice street." The abbreviation for street occurs at the end of the sentence and the next word is a common word. However

if we had "Mr. Cook is a amicable person." The abbreviation is identified correctly, but because it is an honorific it is not the end of a sentence even though the following word is a common word (but also a proper name - ambiguity). One solution would be to have more rules to deal with special cases like this, but one aim of this sentence splitter was generality and simplicity. It was shown from Table 2 that the average sentence length even for list items and headings is on average greater than 6 words (6.10 words/sentence). Therefore it is quite reasonable to assume that most periods that are full stops will not occur within the first 6 words. If they do, then these are not considered as sentence terminals, but simply as abbreviations. Of course there will be exceptions to this but on the whole this appears reasonable.

Mikheev goes into more details about finding proper names and abbreviations and uses a number of methods for knowledge-free proper name disambiguation (he deals with multi-word proper name extraction to find expressions such as "Tony Blair" or "Rt. Hon. Tony Blair" and not just single terms). However, to keep this program simple, I do not consider this problem.

5 Evaluation

5.1 Introduction

Consider how we might test the sentence splitter against the BNC (version 1.0) or Brown corpus contained in the Penn Treebank version 3 (PTB3). One could extract the sentences from the BNC/PTB3 and compare them with the sentence splitter output. However this actually becomes an alignment problem and presents a non-trivial task. Therefore the method used in this report is to treat sentence boundary disambiguation as a classification task. The problem is to classify all occurences of [.?!:] as either a sentence break or non-sentence break and compare the results. The alignment problem becomes much easier. All we need to do is sequentially process the original and sentence-split texts and compare classification of the punctuation to determine how many are correctly disambiguated. Punctuation from the original BNC/PTB3 text can be extracted and classified using the SGML sentence tags.

I split the evalutation into four sections:

- Extract raw text from the BNC/PTB3 and classify all occurrences of [.!?:] with `[sentence_break]` or `[non_sentence_break]`. The programs `classifyBNCsentences.pl` and `classifyPTBsentences.pl` do this task.
- Extract from the classified BNC or PTB3 file the raw text (input for the sentence splitter). The `extractclassifiedtext.pl` program does this task for the BNC and `classifyPTBsentences.pl` for the PTB3.
- Run the sentence splitter on the raw text extracted from the BNC and PTB3.

- Compare classification of all occurrence of [.!?:] between classified BNC or PTB3 file and output from sentence splitter. The `compare.pl` program does this.

The above procedure assumes that the sentence splitter will output classes for each [.!?:]. The program `splitter.pl` does this automatically. The `classifyBNCsentences.pl` program simply runs through the raw BNC text and classifies all [.?!:] by adding a tag after each one: `[sentence_break]` or `[non_sentence_break]`. It does this by extracting all the sentences (using the SGML tags indicating sentence breaks) and then assigning a `[sentence_break]` tag to the last [.?!:] in the sentence (if there is one) and `[non_sentence_break]` to every other occurrence. This matches the output from the sentence splitter: `splitter.pl`.

To compare the resulting sentence splitter classifications with the BNC, the `compare.pl` program first extracts all occurrences of [.?!:] followed by the classification tag from both the sentence splitter output file and the classified BNC/PTB3 file and creates two temporary files, each line containing [.!?:] followed by the classification tag, e.g.:

```
.[sentence_break]
:[non_sentence_break]
![non_sentence_break]
   .
   .
   .
```

It is an easy task to then compare these two lists to look for differences between the classifications. The reason one does not simply use a UNIX `grep` command to extract the number of [.?!:] correctly assigned as sentence breaks or non-sentence breaks is that this does not compare sequential versions of [.?!:]. In other words 1 million periods could be assigned sentence breaks, but not the same 1 million that are sentence breaks in the "gold standard" file. Hence the reason for the comparison program.

I tested the following characteristics of the sentence splitter:

1. `T1` - The baseline sentence splitter: anything–[.!?:]–anything;
2. `T2` - The baseline sentence splitter: anything–[.!?:]–capital/number;
3. `T3` - The baseline sentence splitter: capital/number–[.!?:]–capital/number;
4. `T4` - The baseline sentence splitter (T3) and abbreviation list but no common words;
5. `T5` - The baseline sentence splitter (T3) and abbreviation list and common words but no len > 6 condition;
6. `T6` - The baseline sentence splitter (T3) and abbreviation list but assume abreviation is non_sentence_break (no common words);
7. `T7` - The baseline sentence splitter (T3) and abbreviation list, common words and len > 6 condition (final sentence splitter);

EOS marker	#EOS markers as sentence breaks / Total EOS marker (% of total)
[.]	623,016/679,742 (91.65%)
[!]	10,881/12,120 (89.78%)
[?]	29,145/31,719 (91.88%)
[:]	33,642/37,080 (90.73%)

Table 4: Number of [.!?:] as sentence terminals in 'A' part of the BNC

The results for these on both the BNC and Brown can be found in Section 5.2 and Section 5.3 respectively.

5.2 Evaluation using the BNC

5.2.1 Introduction

Using the process described in Section 5.2, I tested the sentence splitter on just the 'A' files from the BNC. The main reason for this was lack of available time for testing. The 'A' section of the BNC consists of 678 written files (0 spoken), 13,640,542 words and 747,547 sentences. These are summarised in Table 4.

The results are similar to those in Table 1 for the whole of the BNC. Periods ('.') form 83.3% of sentence terminals with 1.5% for '!'s, 3.9% for '?'s and 4.5% for ':'s. From Table 4, 91.65% of all periods are sentence terminals with similar results to those for the whole BNC.

5.2.2 Results from the 'A' files of the BNC

Table 5 shows the results of the rule-based sentence splitter on this part of the BNC for test conditions T1 to T7. The results in the most right-hand column in bold typeface highlight the results from the sentence splitter described in Section 4.2.

It is interesting to see that the results from the baseline sentence splitter are reasonable (except for that of ':') with 98.14% of periods disambiguated correctly. T5 gives the best result of 98.70% perhaps reflecting that the length condition is not helpful, but actually makes the results worse. The result for '!' with T1 is 84.51%, but this increases significantly when the condition that whitespace and a capital/number must follow the '!' is added to the baseline splitter. This is also true for '?' which goes from 89.47% to 99.24% correct.

Finally to mention is that although the increase when using the abbreviation and common word heuristics is not particularly great, it does improve the splitter output from the point-of-view of the small-scale test. On the large-scale, the naive splitter does well, but

EOS marker	% classified correctly						
	T1	T2	T3	T4	T5	T6	**T7**
[.]	98.14%	98.30%	98.24%	96.46%	98.70%	97.93%	**98.59%**
[!]	84.51%	98.99%	98.99%	98.99%	98.99%	98.99%	**98.99%**
[?]	89.47%	99.24%	99.24%	99.24%	99.24%	99.24%	**99.24%**
[:]	26.94%	58.00%	58.21%	64.67%	64.71%	64.67%	**64.70%**

Table 5: Number of [.!?:] correctly disambiguated in the 'A' part of the BNC

EOS marker	#EOS markers as sentence breaks / Total EOS marker	% of total
[.]	48,978/55,495	88.3%
[!]	789/1,597	49.4%
[?]	2,340/4,690	49.9%
[:]	0/1986	0%
[other]	1	

Table 6: Number of [.!?:] as sentence terminals in the Brown corpus

a human evaluator would not be impressed to see sentence breaks after every honorific or abbreviation. This highlights the problems with testing on this quantity of text and that has been automatically classified to begin with. Also, without many abbreviations, the naive splitter would perform well (as it does here) and there would not be much improvement when using the abbreviation detection and common word heuristics.

5.3 Evaluation using the Brown corpus

5.3.1 Introduction

The Brown corpus can be found in most versions of the Penn Treebank (PTB). I use PTB version 3.0 in this evaluation. The Brown corpus has been tagged and parsed in version 3.0 and the tagged version is used in this analysis. The corpus consists of 52,108 sentences (Mikheev gets 49,532 sentences ending in '.' and Grefenstette gets 52511 ending in '.' or '?'), 1,033,475 words and 500 files. The Brown corpus is a standard corpus of American English for use with digital computers and fist created in 1964 by W. N. Francis and H. Kucera at Brown University. Table 6 shows the number of [.!?:] that are end-of-sentence (EOS) markers.

From my analysis, 93.99% of sentences end with a period '.' and 88.3% of periods can be found at the end of sentences. 1.51% of sentences end with '!' and 49.4% of all occurrences are at the end of a sentence. This considerably lower than for the BNC. 4.49% of sentences end with '?' and none end with ':'. Also only 1 sentence ends with something other than [.!?:] which perhaps reflects the type of text included in the Brown corpus (i.e. well-formed and overall grammatically correct).

5.3.2 Processing files from the Brown corpus in Penn Treebank 3

In the Penn Treebank version 3.0, there are not rawtexts as in version 2.0. Therefore the program `classifyPTBsentences.pl` runs through the tagged part of the Brown corpus and from this, extracts sentences and removes PTB tags. The sentences in the tagged file are separated by a series of '=' characters. These are used to indicate sentence breaks. An example PTB tagged file (CR01.POS) is shown below (including the header):

```
*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*
*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*
*x*                                                                     *x*
*x*            Copyright (C) 1990 University of Pennsylvania            *x*
*x*                                                                     *x*
*x*    The data in this file are part of a preliminary version of the   *x*
*x*    Penn Treebank Corpus.  Any research using this corpus or based   *x*
*x*    on it should acknowledge that fact, as well as the preliminary   *x*
*x*    nature of the corpus.                                            *x*
*x*                                                                     *x*
*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*
*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*



==================================

[ It/PRP ]
was/VBD among/IN
[ these/DT ]

[ that/WP ]
Hinkle/NNP identified/VBN
[ a/DT photograph/NN ]
of/IN
[ Barco/NNP ]
!/. !/.
==================================
```

When extracting rawtext, the header is removed and sentences extracted in the same way as for `classifyBNCsentences.pl`. The program will also classify all the [.!?:] punctuation marks with the `[sentence_break]` and `[non_sentence_break]` labels depending upon whether the punctuation mark is at the end or middle of the sentence. The rawtext extraction is somewhat naive and does not deal with joining hyphenated words. It is worth mentioning that all punctuation in the PTB tagged file is tagged as, for example, `!/ .` representing '!' and `./ .` representing '.'.

The rawtext can be fed into the sentence splitter and the classified file and output from the splitter used as input to `compare.pl` to evaluate accuracy of the splitter.

5.3.3 Results from the Brown corpus evaluation

Table 7 shows the results of the rule-based sentence splitter on the Brown corpus for test conditions T1 to T7. Again, the results in the most right-hand column highlight the results from the sentence splitter described in Section 4.2.

EOS marker	% classified correctly						
	T1	T2	T3	T4	T5	T6	**T7**
[.]	92.18%	92.95%	93.01%	93.01%	97.61%	98.52%	**97.61%**
[!]	49.41%	97.81%	97.81%	97.81%	97.81%	97.81%	**97.80%**
[?]	49.89%	98.87%	98.87%	98.87%	97.81%	97.81%	**98.87%**
[:]	9.67%	40.89%	40.89%	40.89%	63.54%	64.50%	**63.54%**

Table 7: Number of [.!?:] correctly disambiguated in the Brown corpus

The results for '.' disambiguation using the baseline splitter are high again with 92.18% correct. However, results for '!' and '?' are much lower. The main reason for this is that there are many examples in the Brown corpus where '!' and '?' are repeated twice (in fact almost all of them!). For example:

```
==================================
``/`` Is/VBZ
[ that/DT ]
so/RB ''/'' ?/. ?/.
==================================
```

This means that the baseline splitter will get about 50% wrong which is verified by the resulting 49.14% and 49.89% correctly classified. Therefore T2 improves the results dramatically by enforcing that the [.!?:] must be followed by whitespace and then a capital/number. Using the abbreviations with common words (T4) does not appear to improve the results for period disambiguation. Only with common words, T5, do the results improve. A final interesting point to notice is thathe condition T6 actually gives the best results when after an abbreviation the period is considered a non-sentence break. However, this could be because there are not enough common words recognised and therefore the condition of sentence break is never reached and it always asummed the next word is a proper name and therefore no sentence break is signalled. The only solution to this is to use a better dictionary or perform accurate proper name disambiguation. From Mikheev [7], he finds 8,366 proper names in the Brown corpus.

6 Conclusions

In this report, I have described a simple splitting algorithm, analysis and evaluation on both the BNC and Brown corpus. The results are not as high as other academics have reported, but the main aim of the sentence splitter was simplicity and ability to easily modify the code. This has been acheived together with reasonable results. I have not created a new algorithm for sentence splitting, but simply reused ideas of a number of researchers working in this area. I have, however, analysed the BNC for sentence characteristics and produced a framework such that I can evaluate the splitter under different conditions using a simple [.!?:] disambiguation approach.

There are areas that need further investigation from this report and I mention just a couple of the most important. First has to be to test and improve the dictionary used in the program. It is only very small and a larger lexicon could be used e.g. Wordnet. One of the simple improvements that could help the program would be the introduction of a morphological analyser. I found many cases where input words were not matched as common words because of simple differences such as tense changes. Morphological analysis would help this. A further improvement could be to use a name entity recogniser to extract names. The approach I use is that finding common words is easier than proper names and therefore make a check for common words after the period. Common words could be extracted statistically from a large corpus such as the BNC.

I have used a number of heuristics to detect abbreviations supplemented with an exceptions list. However, this exceptions list (and indeed the heuristics) need to be investigated further. There may be cases when abbreviations are being missed or common words incorrectly assumed to be abbreviations. Mikheev highlights the importance of accurate abbreviation detection and name entity recognition with respect to period disambiguation. This would be an interesting area to continue further research.

Finally, I have evaluated the sentence splitter only on two corpora. It would be good to evaluate the splitter on more examples such as the Wall Street Journal (WSJ) and other corpora. This would help determine the generality of the splitter and find more examples where the algorithm fails. A rule-based approach will work well I think in most cases, but this will depend upon the input data. Ted Dunning highlighted the need to know about the domain before building a splitter and I think this can be justified by the results in this report. I used the BNC for analysis and the splitter performs better on the BNC. The approaches used to extract the raw text from the tagged corpora has been naive (to say the least) and this is another area that could, again, be improved upon and evaluated with further investigation.

7 Acknowledgements

I would like to thank Ted Dunning for personal email correspondance and both Tony Rose and Ave Wrigley for their interest in my work. I also appreciate the help of Wim Peters with extracting abbreviations from various lexical resources such as Wordnet, CELEX and LDOCE.

References

[1] E. Brill. A simple rule-based part of speech tagger. In *Proceedings of the Third conference on Applied Natural Language Processing (ANLP 3)*, 1992.

[2] L. Burnard. *Users Reference Guide for the British National Corpus.* Oxford University Computing Services, 1995.

[3] R. Garside. The claws word-tagging system. In *The Computational Analysis of English: A Corpus-based Approach.* Longman, 1997.

[4] G. Grefenstette and P. Tapanainen. What is a word, what is a sentence? problems of tokenization. In *Proceedings of the 3rd Conference on Computational Lexicography and Text Research (COMPLEX'94)*, 1994.

[5] J.R. Hurford. *Grammar - A student's guide.* Cambridge University Press, 1994.

[6] C.D. Manning and H. Schütze. *Foundations of statistical natural language processing.* MIT Press, 1999.

[7] A. Mikheev. Periods, capitalized words, etc. *Computational Linguistics*, 16(1):1–32, 1994.

[8] A. Mikheev. Feature lattices for maximum entropy modelling. In *ACL 36*, pages 848–854, 1998.

[9] D.D. Palmer and M.A. Hearst. Adaptive sentence boundary disambiguation. In *Proceedings of the Forth conference on Applied Natural Language Processing (ANLP 4)*, pages 78–83, 1994.

[10] J.C. Reynar and A. Ratnaparkhi. A maximum entropy approach to identifying sentence boundaries. In *Proceedings of the fifth ACL Conference on Applied Natural Language Processing (ANLP'97).* ACL, 1997. Washington D.C.

[11] M.D. Riley. Some applications of tree-based modeling to speech and language indexing. In *Proceedings of the DARPA Speech and Natural Language Workshop*, pages 339–352. Morgan Kaufmann, 1989.

[12] M. Stevenson and R. Gaizauskas. Experiments on sentence boundary detection. In *Proceedings of the Sixth conference on Applied Natural Language Processing (ANLP 6) and First conference of the North American Chparter of the Association for Computational Linguistics*, 2000.

Perl `splitter.pl` source code

```
#!/bin/perl

#*********************************************************************************************
#  SENTENCE SPLITTER

# Author: Paul Clough  {cloughie@dcs.shef.ac.uk}

# Description: This sentence splitter is a very simple splitter that acts as a
# base-line to evaluate further splitters. The simple rule is this:

# All [.!?] are considered as a sentence terminal if followed
# by at least one whitespace and a capital letter (also allow
# for optional brackets and quotes).

# $sentence_break = /[.!?][ ()"]+[A-Z]

# Input: An ASCII text file e.g. blob.txt

# Output: The input ASCII text file but with one sentence per line
# and all [.!?] marked with category [sentence_break] or
#  [non_sentence_break]. The output filename is the input
# filename appended with ".split".

# Known bugs: The program ignores anything after the last sentence of the input stream.
# This means that some text from the splitter at the end of the file could
# appear missing. This has not been a problem to date, but may appear as a
# problem in the future. A further release of splitter will resolve this.

# Uses: The program makes use of two files: 1) splitter.dict - a simple lexicon and
# 2) splitter.abv - a list of abbreviation exceptions that do not match the
# simple abbreviation detection rules.

# Useage: %perl splitter.pl -f <file to split>

# e.g. %perl splitter.pl -f input.txt

# Revision: 12/04/01 Version 1.0

# Any problems or suggestions for improvements, please email me.

#*********************************************************************************************

require 5.004;
require "getopts.pl";

# Use the Linguia::EN::Syllable Perl module to calculate the number of syllables.
# From the CPAN site - about 96% accurate.
use Syllable;

&Getopts('f:');

$dictionary = "splitter.dict";
$abbrv_file = "splitter.abv";
$len = 0;
@COMMON_TERMS = ();
@ABBREVIATIONS = ();
$output_file = "$opt_f.split";

if (!$opt_f) {
die "Useage: perl splitter2.pl -f <file to split>\n\n";
```

```
}

open(OUTPUT_FILE, ">$output_file") || die "cannot create the output file: $!";

print "Working .... reading in dictionary and abbreviation files\n";

# Load in the dictionary and find the common words.
# Here, we assume the words in upper case are simply names and one
# word per line - i.e. in same form as /usr/dict/words
&loadDictionary;

# Same assumptions as for dictionary
&loadAbbreviations;

print "Working .... reading in raw text file\n";

open(FILE, $opt_f) or die "Can't open $opt_f for reading\n";

while (defined ($line = <FILE>)) {
$text.= $line;
}
close(FILE);

print "Working .... splitting sentences\n";
# Remove any carriage returns etc.
$text =~ tr/\n\r/ /;

# for my purposes, remove hex 19 (^Y character)
$text =~ tr/\x19//d;

# make sure there are always spaces following punctuation to enable splitter to work
# properly - covers such cases as believe.I ... where a space has forgotten to be
# inserted.
$text =~ s/(.*)([.!?])(.*)/$1$2 $3/g;
print OUTPUT_FILE "\n";

# sentence ends with [.!?], followed by capital or number. Use base-line splitter and then
# use some heuristics to improve upon this e.g. dealing with Mr. and etc.
# In this rather large regex we allow for quotes, brackets etc.
# $1 = the complete sentence including beginning punctuation and brackets
# $2 = the punctuation mark - either [.!?:]
# $3 = the brackets or quotes after the [!?.:]. This is non-grouping i.e. does not consume.
# $4 = the next word after the [.?!:].This is non-grouping i.e. does not consume.
# $5 = rather than a next word, it may have been the last sentence in the file. Therefore capture
#      punctuation and brackets before end of file. This is non-grouping i.e. does not consume.
while($text =~ /([\'\"`]*[({[]?[a-zA-Z0-9]+.*?)([\.!?:])(?:(?=([([{\"\'`)}\]<]*[ ]+)[([{\"\'`)}\]
]*([A-Z0-9][a-z]*))|(?=([()\"\'`)}\] ]+)$))/gs ) {

$sentence = $1;
$punctuation = $2;
if (defined($3)) {
$stuff_after_period = $3;
} else {
if ($5) {
$stuff_after_period = $5;
} else {
$stuff_after_period = "";
}
}

@words = split(/\s+/, $sentence);
$len+=@words;
```

```
if ($4) { $next_word = $4; } else { $next_word = ""; }

if ($punctuation =~ /[\.]/) {
# consider the word before the period => is it an abbreviation? (then not full-stop)
# Abbreviation if:
#  1) all consonants and not all capitalised (and contain no lower case y e.g.
shy, sly
#  2) a span of single letters followed by periods
#  3) a single letter (except I).
#  4) in the known abbreviations list.
# In above cases, then the period is NOT a full stop.

# perhaps only one word e.g. P.S rather than a whole sentence
$sentence =~ /\s+([a-zA-Z\.]+)$|([a-zA-Z\.]+)$/;

if ($1) { $last_word = $1; } else { $last_word = $2; }

if ( (($last_word !~ /.*[AEIOUaeiou]+.*/)&&
($last_word =~ /.*[a-z]+.*/)&&($last_word !~ /.*[y]+.*/))||
     ($last_word =~ /([a-zA-Z][\.])+/)||
     (($last_word =~ /^[A-Za-z]$/)&&($last_word !~ /^[I]$/))||
     ($abbreviation =~ / $last_word /i) ) {

# We have an abbreviation, but this could come at the middle or end of a
# sentence. Therefore we assume that the abbreviation is not at the end of
# a sentence if the next word is a common word and the abbreviation occurs
# less than 5 words from the start of the sentence.

$next_word = lc $next_word;

if ( ($common_term =~ / $next_word /)&&($len > 6) ) {
# a sentence break
        &print_sentence("$sentence"."$punctuation".
        "[sentence_break]"."$stuff_after_period");
$len = 0;

} else {
      # not a sentence break
&print_non_sentence("$sentence"."$punctuation".
"[non_sentence_break] ");
}
} else {
# a sentence break
&print_sentence("$sentence"."$punctuation"."[sentence_break]".
"$stuff_after_period");
$len = 0;
}

} else {
  # only consider sentences if : comes after at least 6 words from start of
   # sentence
   if (($punctuation =~ /[!?]/)||(($punctuation =~ /[:]/)&&($len > 6))) {
     # a sentence break
     &print_sentence("$sentence"."$punctuation"."[sentence_break]".
     "$stuff_after_period");
     $len = 0;
   } else {
     # not a sentence break
     &print_non_sentence("$sentence"."$punctuation"."[non_sentence_break] ");
   }
}
```

```
}

# finally print any remaining text after the last match to make sure output text is same as
# the input
close(OUTPUT_FILE);
exit;

#*********************************************************************************************

#*********************************************************************************************
# procedures
#*********************************************************************************************

sub loadDictionary {

# Initialise var
$common_term = "";

if (open(DICT, $dictionary)) {

while (defined ($line = <DICT>)) {
chomp($line);
if ($line !~ /^[A-Z]/) {
push(@COMMON_TERMS, "$line");
}

}

close(DICT);

# build up the common word list
foreach $word (@COMMON_TERMS) {
$common_term .= $word.' ';
}
chop $common_term;
} else {
print "cannot open dictionary file $opt_d: $!";
}
}

sub loadAbbreviations {

# Initialise var
$abbrv_term = "";

if (open(ABBRV, $abbrv_file)) {

while (defined ($line = <ABBRV>)) {
chomp($line);
push(@ABBREVIATIONS, "$line");
}

close(ABBRV);

# build up the abbreviations list
foreach $word (@ABBREVIATIONS) {
$abbreviation .= $word.' ';
}
chop $abbreviation;
} else {
print "cannot open dictionary file $opt_d: $!";
}
```

```
}

sub print_sentence {

  my $sentence = shift;
  my $final_sentence = "";

  # deal with anything after the [.!?:][sentence_break] and add this to the end e.g. "
  $sentence =~ /\[sentence_break\](.*)$/;
  $leftover = $1;

  while ($sentence =~ /(.*?)([\.!?:])/gs) {
    $final_sentence.= $1.$2."[non_sentence_break]";
  }

  $final_sentence =~ s/\[non_sentence_break\]$/[sentence_break]/;
  $final_sentence.=$leftover;
  print OUTPUT_FILE "$final_sentence\n\n";
}

sub print_non_sentence{

  my $sentence = shift;
  my $final_sentence = "";

  while ($sentence =~ /(.*?)([\.!?:])/gs) {
    $final_sentence.= $1.$2."[non_sentence_break]";
  }

  print OUTPUT_FILE "$final_sentence ";
}

#********************************************************************************************
```