COM6115: Lab Class 1

Segmenting Words in Chinese Text

In this lab you will get a better understanding of UTF-8 encoding (which we've covered in the lectures) and build a simple word segmenter for traditional Chinese text.

Introduction

Chinese has a *logographic* writing system in which individual symbols, or *characters*, may represent whole words, or even phrases. However, some words are represented by *sequences of several characters*, and many characters that can stand alone as one word can also be part of the character sequences representing other words. Furthermore, traditional Chinese writing has no explicit marking of when a word ends (as we do with spaces in English). Thus, a Chinese sentence is written as a consecutive string of characters without spaces. For example:

> 中文句子由连续的一系列单词组成。
> (TRANSLATION: "A Chinese sentence is formed by a consecutive string of words")

The above sentence would usually be segmented into words as follows:

中文	句子	由	连续	的	一系列	单词	组成
Chinese	sentence	cause/with	consecutive	adjective marker	a series of	word	formed

Human readers can easily identify how the characters group into words, based on their meaning. This task is more difficult for a machine, however, but is required before various other tasks can be done, e.g. counting word occurrences or building language models. Hence, this task – known as *word segmentation* – is an important step in processing of Chinese text.

Greedy match algorithm

The *greedy match* algorithm, also called the *maximum match* algorithm, is a simple technique for Chinese word segmentation.

The algorithm starts at the beginning of a sentence and tries to find the longest word starting at that position, by checking against a predefined word list (dictionary). If no match is found in the word list, the algorithm simply treats the next (single) character as the next word.

It then moves forward to the end of that word, and repeats this process until it reaches the end of the sentence.

Analysis shows that most Chinese words are 5 characters or shorter, so we can set this as the maximum word length, to limit the strings that must be considered during search.

For example, consider segmenting the sentence:

中文句子由连续的一系列单词组成

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

The algorithm starts at the left, i.e. position 0. The group of length 5 here (“中文句子由”) is not in the dictionary, nor those of length 4 (“中文句子”) or 3 (“中文句”), but that of length 2, i.e. “中文” (“Chinese”), *is* in the dictionary, and so is accepted as the first word. The algorithm then advances to the end of this word, i.e. position 2, and repeats until there is no more input.

Python and Unicode

Within a program, Unicode characters and strings can be manipulated directly, without thought as to the underlying encoding form. However, when Unicode characters are stored in files or displayed on a terminal they must be encoded as *one or more bytes*.

Some encoding forms (such as ASCII and Latin-2) use a single byte, and can only support a small subset of Unicode, enough for a single language. Other encoding forms (such as UTF-8) use multiple bytes, so they can represent the full range of Unicode symbols. We will be using **UTF-8** to encode our Chinese text.

Fortunately, Python (3.x) is able to handle Unicode text, but you may need to specify the encoding of a file when a filestream is opened for reading from or writing to that file.

We can do this using the “`encoding`” keyword argument of the open command, as in the following example:

```
with open("filename.txt", "r", encoding = "utf-8") as myfile:
    for line in myfile:
        ... # process your text as you would normally
```

Depending on your operating system and language settings, UTF-8 may be the default encoding on your system, but it is good practice to be explicit about the encoding in use (especially if you intend your code to be run on other systems).

More information about Python encoding and Unicode can be found in the Python 3 documentation.

Exercises

1 Get the data

Download the file `chinese_segmentation_resources.zip` from the lab class folder on Blackboard and unzip it to get the following files:

```
chinesetext.utf8
chinesetext_goldstandard.utf8
chinesetrad_wordlist.utf8

eval_chinese_segmentation.py
chinese_segmentation_STARTER_CODE.py
```

The file `chinesetext.utf8` contains traditional Chinese text with one sentence per line[1].

The file `chinesetext_goldstandard.utf8` contains a version of the same text, but with 'gold-standard' (human-added) word boundaries marked by the addition of spaces.

The file `chinesetrad_wordlist.utf8` is a list of about 17,000 words in Traditional Chinese, with one word on each line.

In this exercise, you will first explore the data and practise applying UTF-8 decoding, then implement the greedy match algorithm described above to try to convert `chinesetext.utf8` into something like `chinesetext_goldstandard.utf8`.

2 Viewing the Text and Exploring Unicode Encoding of Chinese

Your system may be unable to read the file `chinesetext.utf8` depending on how you attempt to do so. If you can't simply open it, you might try opening the file in a web browser and selecting the translate option to see an English translation.

Alternatively, if you are using Spyder, you should be able to open it directly by selecting the "All files" option rather than the "Supported Files" option in the Open File dialogue box.

[1] The text comes from the Sinica Treebank corpus, as made available in NLTK.

To better understand UTF-8 encoding try the following.

First, read in the file `chinesetext.utf8` **as a binary file** and display the first few bytes of it as a string of hexadecimal codes:

```
import binascii
with open("chinesetext.utf8", 'rb') as f:
    filecontent = f.read()
chunk1 = filecontent[:11]
print(chunk1)
```

You should see the string: `b'\xe4\xb8\x80\n\xe5\x8f\x8b\xe6\x83\x85\n'` .

Try to make sense of each character in this string. What do these 11 characters correspond to in the input file? Why are there 11?

Ask a demonstrator if you cannot understand the role of each.

Recall from Lecture 3 the table showing how Unicode code points are mapped onto encoding forms:

Scalar Value	UTF-16	UTF-8 Bit Distribution			
		1st Byte	2nd Byte	3rd Byte	4th Byte
000000000xxxxxxx	000000000xxxxxxx	0xxxxxxx			
00000yyyyyxxxxxx	00000yyyyyxxxxxx	110yyyyy	10xxxxxx		
zzzzyyyyyyxxxxxx	zzzzyyyyyyxxxxxx	1110zzzz	10yyyyyy	10xxxxxx	
uuuuuzzzzyyyyyyxxxxxx	110110wwwwzzzzyy+ 11011yyyyxxxxxx	11110uuu[a]	10uuzzzz	10yyyyyy	10xxxxxx

Use this table to work out the Unicode code point in hexadecimal for the first Chinese character in the above UTF-8 string.

You should get **x4E00**.

> **Hint:** Convert the first 3 UTF-8 bytes above to binary and then match them against row three in the table to derive the binary scalar value for the Unicode code point; then convert that number to hexadecimal.

Do the same for the next two encoded Chinese characters in the string. Confirm your answers by checking in:
https://www.utf8-chartable.de/unicode-utf8-table.pl?start=19968&utf8=string-literal.

When you are comfortable with this, try the following:

```
print(chunk1.decode('utf8'))
```

Confirm visually that the characters you see are the first two lines of the file `chinesetext.utf8`.

By way of contrast, you might try repeating the above steps with a file of English text (which you can create using a text editor).

3 Word Segmentation

Rename the code file `chinese_segmentation_STARTER_CODE.py`, and then add your own code, so as to load the dictionary of Chinese words (`chinesetrad_wordlist.utf8`), and to use that in your own implementation of the *greedy match algorithm* described above. Write the results to an output file `MYRESULTS.utf8` (encoded as UTF-8), with spaces added for word boundaries.

Hint: Your algorithm structure might be something like the following:[2]

1. Set **Position** to 0 (the beginning of the sentence)
2. Set **Length** to **MaxLength**
3. While **Length** > 1:
   (a) Select characters from **Position** to (**Position** + **Length** - 1)
   (b) Are selected characters listed in the Dictionary as a word?
   YES: Output selected characters + space; set **Position** to **Position** + **Length**; go to 2
   NO: Set **Length** to (**Length** - 1)
4. If **Length** = 1:
   - Output character at **Position** + space
   - Set **Position** to **Position** + 1
   - Go to 2
4. When sentence is exhausted:
   - Output "newline"
   - Go to next sentence

As noted above, you can set the maximum word length (**MaxLength**) to 5.

[2] This is not an actual program but *pseudocode*.

4 Next Steps

You can 'score' the performance of your system by using the evaluation script supplied, calling this (e.g.) as in the following example:

```
> python eval_chinese_segmentation.py chinesetext_goldstandard.utf8 MYRESULTS.utf8
```

This will compare your output against the human-written "gold standard" segmentation.

Are there any recurring errors you notice when comparing the algorithm's output with the "gold standard"?

Can you think of anything which could be done to improve on the *greedy match* algorithm, or of any other approaches to segmenting the text which could be tried?