IIT Madras
BSc Degree

Lesson-6.2

Text processing

The following paragraph is an excerpt from a talk given by Guido. The full text can be found here.

> In reality, programming languages are how programmers express and communicate *ideas* — and the audience for those ideas is other programmers, not computers. The reason: the computer can take care of itself, but programmers are always working with other programmers, and poorly communicated ideas can cause expensive flops. In fact, ideas expressed in a programming language also often reach the end users of the program — people who will never read or even know about the program, but who nevertheless are affected by it.

Text processing plays an important role in analyzing text data. Given a piece of text, the following are some of the basic questions that we can ask:

- How many sentences are there in the text?
- How many words are there in the text?
- How many of them are unique?
- Which word appears the most number of times?

Are these meaningful questions to ask? Do they lead us anywhere? Yes, they do! Consider the task of classifying articles. Some sample categories could be: lifestyle, science and technology, literature, films. If we want to understand what category an article falls under, one way to go about it is to read the entire article. We can do it for one or two articles, but what if we have to do this for hundreds of them? A better solution would be to computationally process each article, find the top five most common words and use that to get an idea of what the text is about.

Let's get started. The first task is to store the text as a string:

```python
text = "In reality, programming languages are how programmers express and communicate ideas — and the audience for those ideas is other programmers, not computers. The reason: the computer can take care of itself, but programmers are always working with other programmers, and poorly communicated ideas can cause expensive flops. In fact, ideas expressed in a programming language also often reach the end users of the program — people who will never read or even know about the program, but who nevertheless are affected by it."
```

Number of sentences

Sentences could end with one of the following tokens: full stop, exclamation mark or question mark. For simplicity, let us assume that all sentences in our text ends with a full stop. We can split the string using full stop as a delimiter to get a list of sentences:

```python
sentences = text.split('.')
# Prints one sentence in each line
for sentence in sentences:
    print(sentence)
print(f'There are {len(sentences)} sentences in this text.')
```

Notice that there are only three sentences, but we get the output to be four in the last line. On closer inspection, we see that `sentences[-1]` is not a sentence but an empty string. This is because, when a string is split using a delimiter which is present in the string, two substrings get generated, one to the left of the delimiter and the other to its right. As the full stop is the last character in the text, the substring to its right is an empty string. One way to correct this is to remove all empty strings in `sentences`:

```python
while '' in sentences:
    sentences.remove('')
print(f'There are {len(sentences)} sentences in this text.')
```

One problem solved!

Number of words

To get the number of words, we can split each sentence by space:

```python
words = [ ]
for sentence in sentences:
    words_ = sentence.split(' ')    # words_ contains words in sentence
    words.extend(words_)            # words is the collection of all words
print(f'There are {len(words)} words in this text')
```

We get the number of words to be 86. Is that correct? wordcounter.net claims that there are 82 words in this text. Something is wrong with our code. Let us print each word along with its index in separate lines and see what we have:

```
for index, word in enumerate(words):
    print(index, word)
```

Observing the output, we notice the following offenders:

```
11 —
23
49
67 —
```

Indices 11 and 67 are em dashes (—) while 23 and 49 correspond to empty strings. Since we have two different characters to remove, let us clean up the list in the following way:

```
proc_words = [ ]
for word in words:
    if not(word == '' or word == '—'):
        proc_words.append(word)
print(f'There are {len(proc_words)} words in this text')
```

And we have 82 words as expected. One more problem solved!

Number of Unique Words

You might be wondering why this lesson has come under chapter-6 if there are no dictionaries floating around. This section will assuage that worry.

```
uniq_words = dict()
for word in proc_words:
    if word not in uniq_words:
        uniq_words[word] = 0
    uniq_words[word] += 1
print(f'There are {len(uniq_words)} unique words in this text')
```

Let us now test if our code is working as expected. Upon manual inspection, the word "programmers" occurs four times in the text. What does our dict have to say?

```
print(uniq_words['programmers'])
```

We get `2` as the output, another wrong answer! Programming doesn't seem like magic after all. We are making mistakes far too often. Note that this is not the exception, but the norm. The nice part of making mistakes is that they are almost always an opportunity to learn something. An error in the code is hidden knowledge, it is some piece of insight that we are yet to unmask. Now, back to the drawing board. Let us search for all entries in the list `proc_words` that have the substring "programmers" in them:

```
for word in proc_words:
    if 'programmers' in word:
        print(word)
```

This gives the following output:

```
programmers
programmers,
programmers
programmers,
```

So, the problem is with the special character: comma. To confirm this:

```
assert uniq_words['programmers'] + uniq_words['programmers,'] == 4
```

Another problem is introduced by the capitalization of words, usually at the beginning of sentences. Now that the problems have been identified, let us go ahead and fix them. This means going back to the list of words and then generating `proc_words` in the right way:

```
1  proc_words = [ ]
2  for word_ in words:
3      word = word_.lower()
4      if not(word == '' or word == '-'):
5          if not word_.isalnum():
6              word = word_[:-1]
7          proc_words.append(word)
8  print(f'There are {len(proc_words)} words in this text')
```

Several things are happening here. In line-3, every word is converted to lower case. In line-4, em dashes and empty strings are being ignored. Line-5 checks if a word contains a special character. If it does, then it is unburdened of that dangling character in line-6. Here we assume that special characters usually appear at the end of the word. In this text, there are two cases: "programmers," and "reason:". All processed words are finally added to `proc_words` in line-7. Now that we have cleaned up `proc_words`, we can go back and generate `unique_words`:

```
uniq_words = dict()
for word in proc_words:
    if word not in uniq_words:
        uniq_words[word] = 0
    uniq_words[word] += 1
print(f'There are {len(uniq_words)} unique words in this text')
```

Let us print all the words and their counts:

```
for word, freq in uniq_words.items():
    print(word, freq)
```

Lovely! There are 58 unique words in the text. As a test, we can also see if the sum of the counts gives back the total number of words:

```
total = 0
for word in uniq_words:
    total += uniq_words[word]
assert total == len(proc_words)
```

As the code doesn't raise any `AssertionError`, we are correct!

Frequent Words

Finally, let us calculate the top three most frequently occurring words:

```
first_word = second_word = third_word = ''
first_val = second_val = third_val = 0
for word, freq in uniq_words.items():
    if freq > first_val:
        first_val, second_val, third_val = freq, first_val, second_val
        first_word, second_word, third_word = word, first_word, second_word
    elif freq > second_val and freq < first_val:
        second_val, third_val = freq, second_val
        second_word, third_word = word, second_word
    elif freq > third_val and freq < second_val:
        third_val = freq
        third_word = word
print(first_word, first_val)
print(second_word, second_val)
print(third_word, third_val)
```

We see that "programmers" is the second most frequent word. First and third most frequent words are "the" and "in" respectively. Such common words are called stop-words. If they are removed from the text, "programmers" becomes the most frequent non-trivial word. So, without reading this text, one can guess that it should be something about programmers, thanks to Python!

Summary

The main takeaway from this lesson is the kind of mistakes we made and the way we fixed each one of them. In almost every problem, we started off with a solution, then tested it. We figured out that something was wrong, so we went back and tried to fix the problem.