NLP Lab Session Week 4
September 21, 2017

Regular Expressions and Tokenization

So far, we have depended on the NLTK word_tokenizer for our tokenization. Not only does the NLTK have other tokenizers, but we can custom-build our own tokenizer using regular expressions.

Download the examples: LabWeek4RegExpTokenization.ipynb or LabWeek4RegExpTokenization.py

Save the file in a folder where you keep materials for this class. Open your command prompt or terminal window and use the cd command to change directory to your class materials folder. Type at the prompt:

$ jupyter notebook

Start your nlp session by:

>>> import nltk

Text as Strings

We'll read text from the file emma in the Gutenberg Corpus as before, but just leave it as raw text.

```
>>> file0 = nltk.corpus.gutenberg.fileids( ) [0]
>>> emmatext = nltk.corpus.gutenberg.raw(file0)
```

Remember that this text is a Python string. We'll quickly review some of the operations and functions that are used with strings. (See also section 3.2 in the NLTK book, http://www.nltk.org/book/ch03.html

Strings can be treated as lists of characters. So if we get the length of a string, it is the number of characters, and if we use indexing with square brackets [ ], we get substrings of characters.

```
>>> print(type(emmatext) )              # type is string
>>> print(len(emmatext))                # number of characters in the book
>>> emmatext[:150]                  # display the first 150 characters of the string
>>> print (emmatext[:150])          # print these characters
```

Note that just displaying the text shows the end-of-line characters \n, but the print functions renders them as end-of-line.

If we wanted to do something to every character in the string, we can even loop over the characters; compare printing the first 20 characters as a substring, and then as individual characters.

```
>>> print(emmatext[:20])
>>> for char in emmatext[:20]:
    print char
```

We use the operator '+' to catenate strings together.

```
>>> string1 = 'Monty Python'
>>> string2 = 'Holy Grail'
>>> string1 + string2
>>> string1 + ' and the ' + string2
```

Check out the section on Accessing Substrings to see more examples of using the ':' in indexing to get slices.

Also check out Table 3.2 to see other string functions. For example, we could use the function replace to replace all the new characters '\n' with a space ' '. This is special case of using substitution with regular expressions.

```
>>> emmatext[:150]
>>> newemmatext = emmatext.replace('\n', ' ')
>>> newemmatext[:150]
```

We have defined the variable newemmatext to be the text without newlines.

Regular Expressions for Tokenizing Text

Now we have a pretty good tokenizer that works well on regular text like news stories, nltk.word_tokenize, which is trained on Penn Treebank. There is a demo page about the NLTK tokenizers from Jacob Perkins NLTK demos at

http://text-processing.com/demo/tokenize/

But we are going to look more at the tokenization task as a good way to learn about regular expressions. Furthermore, the NLTK allows us to define a tokenizer using those regular expressions, and we can do that for other types of text. Our example will be to define a tokenizer that is specifically for tweets, and also works well on other types of social media text.

In the NLTK book, section 3.4 is about regular expressions, including Table 3.3 with basic regular expression patterns, and section 3.7 includes Table 3.4 with regular expression symbols.

Let's start our investigation of using regular expressions to tokenize text by looking at some simple patterns first. We'll start with a pattern for words that just finds alphabetic characters. There are several functions in the module re for finding how patterns match text, e.g. re.match

finds any match at the beginning of a string, re.search finds a match anywhere in the string, and re.findall will find the substrings that matched anywhere in the string.

```
>>> import re
>>> shorttext = 'That book is interesting.'
>>>  pword = re.compile('\w+')
>>> re.findall(pword, shorttext)
```

This does fine on the alphabetic words of this simple text, noting that \w includes alphabetic characters, numeric digits and underlines, but it ignores the period at the end of the sentence because it doesn't try to match any other types of characters or patterns.

Let's get an example of text with some special characters to illustrate some other situations. We can reuse the pattern pword with this new text.

```
>>> specialtext = 'That U.S.A. poster-print costs $12.40, but with 10% off.'
>>> re.findall(pword, specialtext)
```

Getting only alphabetic text leaves lots of the string unmatched. Let's start making a more general regular expression to match tokens by matching words that can have an internal hyphen. In this case, we need to put parentheses around the part of the pattern that can be repeated 0 or more times. Unfortunately, findall will then only report the part that matched inside those parentheses, so we'll put an extra pair of parentheses around the whole match.

```
>>> ptoken = re.compile('(\w+(-\w+)*)')
>>> re.findall(ptoken, specialtext)
```

re.findall has reported both the whole matched text and the internal matched text, i.e. it reports the last match of any part of the regular expression in parentheses. We could fix this by looking at the parts of the **re.groups** function to access only the outer match. But let's assume that we only want to look at outer matches and not at any of the internal matches. We can instead make the internal parentheses into non-capturing subgroups. This regular expression matches the same strings, but the findall function doesn't report the subgroups.

```
>>> ptoken = re.compile('(\w+(?:-\w+)*)')
```

Now let's check our pattern on a word with two internal hyphens.

```
>>> re.findall(ptoken, 'end-of-line character')
```

Now we try to make a pattern to match abbreviations that might have a "." inside, like U.S.A. We only allow capitalized letters, and we make a simple pattern that matches alternating capital letters and dots.

```
>>> pabbrev = re.compile('(([A-Z]\.)+)')
>>> re.findall(pabbrev, specialtext)
```

This worked well, so let's combine it with the words pattern to match either words or abbreviations.

```
>>> ptoken = re.compile('(\w+(-\w+)*|([A-Z]\.)+)')
>>> re.findall(ptoken, specialtext)
```

Well, that didn't work because it first found the alphabetic words which found 'U', "S' and "A' as separate words before it could match the abbreviations. So the **order of the matching patterns really matters** if an earlier pattern matches part of what you want to match. We can switch the order of the token patterns to match abbreviations first and then alphabetics.

```
>>> ptoken = re.compile('(([A-Z]\.)+|\w+(-\w+)*)')
>>> re.findall(ptoken, specialtext)
```

That worked much better. Now we'll add an expression to match the currency, with an optional $ so that we can also match numbers with optional decimal parts.

```
>>> ptoken = re.compile(' (([A-Z]\.)+|\w+(-\w+)*|\$?\d+(\.\d+)?)')
>>> re.findall(ptoken, specialtext)
```

We can keep on adding expressions, but the notation is getting awkward. We can make a prettier regular expression that is equivalent to his one by using Python's triple quotes (works for either """ or ''') that allows a string to go across multiple lines without adding a newline character. We can use Python's "r" before the string to get a "raw" string. And we also use the regular expression verbose flag to allow us to put comments at the end of every line, which the re compiler will ignore. But we seem to have to put extra parentheses around each of our disjunctions for the multi-line re to format correctly with findall.

```
ptoken = re.compile(r'''(([A-Z]\.)+)    # abbreviations, e.g. U.S.A.
        | (\w+(-\w+)*)                  # words with internal hyphens
        | (\$?\d+(\.\d+)?)              # currency, like $12.40
        ''', re.X)                      # verbose flag
```

But we still had to put in extra parentheses, which messed up the output a lot in the findall. So NLTK has provided us with an even better way to write these expressions.

Regular Expression Tokenizer using NLTK Tokenizer
(From section 3.7 in the NLTK book.)

NLTK has built a tokenizing function that helps you write tokenizers by giving it the compiled pattern. Regular expressions can also be written down in the "verbose" version, using the (?x) flag that allows the alternatives to be on different lines with comments, and it also alleviates the need to put extra parentheses.

```
pattern = r''' (?x)           # set flag to allow verbose regexps
        (?:[A-Z]\.)+          # abbreviations, e.g. U.S.A.
        | \$?\d+(?:\.\d+)?%?  # currency and percentages, $12.40, 50%
        | \w+(?:-\w+)*        # words with internal hyphens
        | \.\.\.              # ellipsis
        | [][.,;"'?():-_%#']  # separate tokens
        '''
```

As far as I can tell, the nltk function regexp_tokenize applies these regular expressions to text by applying each regular expression in order to get anything that matches as a token. We observed the importance of the order of expressions earlier, but also note that it is important that the expression to separate special characters as individual tokens comes last in the list, so that other expressions, such as the words with internal hyphens, can first get longer tokens that involve individual characters.

We put this pattern into NLTK's function for a regular expression tokenizer. Note the order of the arguments to this function; first give the text and then the pattern.

```
>>> nltk.regexp_tokenize(shorttext, pattern)
>>> nltk.regexp_tokenize(specialtext, pattern)
```

Note that if there are any characters not matched by one of the regular expression patterns, then it is omitted as a token in the result.

We might compare regular expression tokenizer with the built-in word tokenizer of NLTK:

```
>>> nltk.word_tokenize(specialtext)
```

This word tokenizer has chosen to separate the $ in currency and the % sign from the percentage. This choice must depend on what later processing is desired.

Next, we'll try to make a **regular expression tokenizer appropriate for tweet text or other social media text**. Some of the patterns in this tokenizer are taken from tweetmotif, a Python regular expression tokenizer written for tweets by Brendan O'Connor. Here is the original description, http://tweetmotif.com/about and the later inclusion into the ARK tools for social media, http://www.cs.cmu.edu/~ark/TweetNLP/

Here is part of a tokenizer derived from tweetmotif:

```
tweetPattern = r''' (?x)      # set flag to allow verbose regexps
      (?:https?://|www)\S+    # simple URLs
      | (?::-\)|;-\))         # small list of emoticons
      | &(?:amp|lt|gt|quot);  # XML or HTML entity
      | \#\w+                 # hashtags
      | @\w+                  # mentions
      | \d+:\d+               # timelike pattern
      | \d+\.\d+              # number with a decimal
      | (?:\d+,)+?\d{3}(?=(?:[^,]|$))   # number with a comma
      | (?:[A-Z]\.)+                    # simple abbreviations
      | (?:--+)               # multiple dashes
      | \w+(?:-\w+)*          # words with internal hyphens or apostrophes
```

```
    | ['\".?!,:;/]+          # special characters
    '''
```

We need some examples to work with (be sure and get these examples from LabWeek4RegExpTokenization.py and do not try to copy/paste from the pdf).

tweet1 = "@natalieohayre I agree #hc09 needs reform- but not by crooked politicians who r clueless about healthcare! #tcot #fishy NO GOV'T TAKEOVER!"

tweet2 = "To Sen. Roland Burris: Affordable, quality health insurance can't wait http://bit.ly/j63je #hc09 #IL #60660"

tweet3 = "RT @karoli: RT @Seriou: .@whitehouse I will stand w/ Obama on #healthcare, I trust him. #p2 #tlot"

Try the regexp tokenizer on these tweets, for example:

nltk.regexp_tokenize(tweet1,tweetPattern)

We can also observe the built-in TweetTokenizer to the NLTK. This was taken from a later version of the TweetMotif tokenizer.

```
>>> from nltk.tokenize import TweetTokenizer
>>> ttokenizer.tokenize(tweet1)
```

Again we note that some different choices were made about grouping some words with special characters.

Lab Exercise

Choose one of the following, i.e. work with either the regular pattern or the tweet pattern in the tokenizer.

1. Run the regexp tokenizer with the regular pattern on the sentence “Mr. Black and Mrs. Brown attended the lecture by Dr. Gray, but Gov. White wasn’t there.”
    a. Design and add a line to the pattern of this tokenizer so that titles like “Mr.” are tokenized as having the dot inside the token. Test and add some other titles to your list of titles.
    b. Design and add the pattern of this tokenizer so that words with a single apostrophe, such as “wasn’t” are taken as a single token.

OR

2. Run the regexp tokenizer with the tweet pattern on the three example tweets.
    a. Design and add a line to the pattern of this tokenizer so that titles like “Sen.” and “Rep.” are tokenized as having the dot inside the token. Test and add some other titles to your list of titles.

b. Design and add to the pattern of this tokenizer so that words with a single apostrophe, such as “can’t” are taken as a single token.
c. Design and add to the pattern of this tokenizer so that the abbreviation “w/” is taken as a single token.

Choose at least one of your tokenizer solutions and post your revised pattern to the **Assignment in Blackboard for Week 4,** with a short example text that demonstrates its effect. Mention any examples that you think of that need additional regular expressions to be tokenized.