Homework 2: Regular Expressions

As part of the "NLP pipeline", we need to be able to search, tokenize, and normalize a text. This **pre-processing** can be done using **regular expressions**. In this homework assignment, you will use regular expressions, as seen in Table 3.3 and Table 3.4 of the NLTK book) in order to gain familiarity with how regular expressions are used in NLP.

First, let's use regular expressions for detecting word patterns in the Walt Whitman's *Leaves of Grass* from NLTK's Project Gutenberg corpus. We can load in the text of *Leaves of Grass* into a long string, `leaves`, and look at the first 200 characters.

```python
from nltk.corpus import gutenberg as gut

# to see file names
gut.fileids()
```

```
['austen-emma.txt',
 'austen-persuasion.txt',
 'austen-sense.txt',
 'bible-kjv.txt',
 'blake-poems.txt',
 'bryant-stories.txt',
 'burgess-busterbrown.txt',
 'carroll-alice.txt',
 'chesterton-ball.txt',
 'chesterton-brown.txt',
 'chesterton-thursday.txt',
 'edgeworth-parents.txt',
 'melville-moby_dick.txt',
 'milton-paradise.txt',
 'shakespeare-caesar.txt',
 'shakespeare-hamlet.txt',
 'shakespeare-macbeth.txt',
 'whitman-leaves.txt']
```

```python
# load in raw text as a long string
leaves = gut.raw('whitman-leaves.txt')

# first 200 characters
leaves[:200]
```

```
'[Leaves of Grass by Walt Whitman 1855]\n\n\nCome, said my soul,\nSuch
verses for my Body let us write, (for we are one,)\nThat should I
after return,\nOr, long, long hence, in other spheres,\nThere to some
g'
```

Now, let's use regular expressions to find word patterns in this long string. We can use `re.findall()`, which requires two arguments: a regular expression and a string to search. Let's say we want to know how often Whitman uses the archaic past tense affix "-'d". For example, *As I **ponder'd** in silence.* You should compile your regular expression first using `re.compile`[1].

[1]You could also use the module-level function and define your regular expression as an argument to that function, as in

```
import re

# this regular expression means any alphanumeric character one or more times
# followed by an apostrophe (which must be escaped) and the letter "d"

# NB: this regular expression must contain the word character one or more times,
# otherwise it will only return 'd
past_re = re.compile(r'\w+\'d')
past_tense = past_re.findall(leaves)

# to see how many past tense constructions there are
len(past_tense)
```

```
1577
```

```
# to look at the first 10 items
past_tense[:10]
```

```
["pleas'd",
 "form'd",
 "Ponder'd",
 "ponder'd",
 "answer'd",
 "deferr'd",
 "Cabin'd",
 "cabin'd",
 "buoy'd",
 "purpos'd"]
```

Q1 Now, create a regular expression to find how often Whitman uses the past perfect (for example, *I had form'd*). How many past perfect constructions are there in *Leaves of Grass*? List them.

We can also use `re.search()` to search through multiple lines in a text. The function `re.search()` takes the same two arguments as `re.findall` takes: the regular expression, and the string to be searched[2]. You can split `leaves` into sentences by splitting on the newline character [3] before searching. In this example, we are searching for words that have 15 or more letters in them. This example also uses the `group()` function[4], which, in this case, returns what is matched. This example uses a **list comprehension**[5] and uses the % operator in order to format variables concatenated with a string.

```
leaves_lines = leaves.split('\n')

# this regular expression means any word character 15 or more times
long_re = re.compile(r'\w{15,}')

long_words_list = ["%s - %s" % (match.group(), a)
for a in leaves_lines for match in [long_re.search(a)] if match]
long_words_list[:10]
```

```
['compassionating - Stands amused, complacent, compassionating, idle, unitary,',
 'inseparableness - The oath of the inseparableness of two together, of the woman that',
 'constructiveness - I see the constructiveness of my race,',
```

`past_tense = re.findall(r'\w+\'d', leaves)`. However, the Python documentation says: "It is important to note that most regular expression operations are available as module-level functions and methods on compiled regular expressions. The functions are shortcuts that don?t require you to compile a regex object first, but miss some fine-tuning parameters."

[2]The difference between `re.findall()` and `re.search` is that `re.search` finds the first match. For more information, consult the relevant section of the Python documentation on regular expression objects

[3]`sent_tokenize()` is similar

[4]For more information on `group()`, read the relevant Python documentation

[5]For more information, see the list comprehension section of Na-Rae Han's Python3 notes or see section 5.1.3 of the Python documentation.

```
 'indiscriminately -     mix indiscriminately,',
 'incomprehensible - The earth is rude, silent, incomprehensible at first, Nature is rude',
 'incomprehensible -     and incomprehensible at first,',
 "excrementitious - Myself discharging my excrementitious body to be burn'd, or render'd",
 'untransmissible - They are calm, subtle, untransmissible by print,',
 'discriminations - Makes no discriminations, has no conceivable failures,',
 'transformations - Changes and transformations every hour, every moment,']
```

Q2 Write regular expressions to find words containing the following word patterns:

1. words with the comparative/superlative suffix-*er, -est*
2. words ending with the suffix *-ment*
3. words beginning with the prefixes meaning 'not': *in-, im-, un-, non-*
4. a group of two words, the first of which is an adverb (ending in *-ly*), the second of which is shorter than 8 characters long

Will the first three regular expressions capture words that do not have affixes? Try your regular expressions on a text and explain what you found (i.e. comment on matches that you found that you did/did not expect to find)

Q3 Now, try your own regular expression to search *Leaves of Grass* or a different Project Gutenberg text. In your write-up, you should provide a specific step-by-step explanation of what you did, including:

1. what word pattern you are looking for
2. the regular expression that you used
3. what you found in the corpus (again, commenting on matches that you found that you did/did not expect to find)

In addition to using regular expressions to find word parts (i.e. affixes), we can use regular expressions for **stemming**, a type of **text normalization**. First, we can normalize the text so that it only includes lowercase letters[6], and then we can try the `porter` and `lancaster` stemmer[7].

```
leaves_lc = leaves.lower()

from nltk.tokenize import word_tokenize

import nltk
porter = nltk.PorterStemmer()
lanc = nltk.LancasterStemmer()

tokens = word_tokenize(leaves_lc)
tokens[:20]
```

```
['[',
 'leaves',
 'of',
 'grass',
 'by',
 'walt',
 'whitman',
 '1855',
```

[6]You may not always want to normalize your text by making it all lowercase, as you would lose the distinction between US (country) and us (pronoun). See *case folding* in Jurafsky and Martin Chapter 2 Section 2.4.2 Word Tokenization and Normalization

[7]These stemmers are useful for English, and in morphologically rich languages, like Russian, but would not work well with nonconcatenative morphology. See Jurafsky and Martin Chapter 2 Section 2.4.4 Collapsing words: Lemmatization and Stemming for more information on morphology.

```
 ']',
 'come',
 ',',
 'said',
 'my',
 'soul',
 ',',
 'such',
 'verses',
 'for',
 'my',
 'body']
```

```
[porter.stem(t) for t in tokens[:20]]
```

```
['[',
 'leav',
 'of',
 'grass',
 'by',
 'walt',
 'whitman',
 '1855',
 ']',
 'come',
 ',',
 'said',
 'my',
 'soul',
 ',',
 'such',
 'vers',
 'for',
 'my',
 'bodi']
```

```
[lanc.stem(t) for t in tokens[:20]]
```

```
['[',
 'leav',
 'of',
 'grass',
 'by',
 'walt',
 'whitm',
 '1855',
 ']',
 'com',
 ',',
 'said',
 'my',
 'soul',
 ',',
 'such',
 'vers',
 'for',
 'my',
 'body']
```

These stemmers are not always morphologically appropriate. The `Lancaster` stemmer erroneously removes the 'e' from the end of words like *come, are, write, one* and the `Porter` stemmer also removes the 'e' of *verse*.

We can also use regular expressions to normalize the text by stripping out non-text elements. The function `re.sub()` replaces what the regular expression matches with a given string. It takes three arguments: the regular expression, the replacement string, and the string to be searched. Let's go back to our raw text `leaves`. Looking at the first 1000 characters, we see the title and author in brackets followed by three newline characters `\n\n\n` and a title "BOOK I: INSCRIPTIONS" in brackets. We may want to get rid of these non-text elements and also eliminate excess white space[8].

```
leaves[:1000]
```

```
"[Leaves of Grass by Walt Whitman 1855]\n\n\nCome, said my soul,\nSuch
verses for my Body let us write, (for we are one,)\nThat should I
after return,\nOr, long, long hence, in other spheres,\nThere to some
group of mates the chants resuming,\n(Tallying Earth's soil, trees,
winds, tumultuous waves,)\nEver with pleas'd smile I may keep
on,\nEver and ever yet the verses owning--as, first, I here and
now\nSigning for Soul and Body, set to them my name,\n\nWalt
Whitman\n\n\n\n[BOOK I.  INSCRIPTIONS]\n\n}  One's-Self I
Sing\n\nOne's-self I sing, a simple separate person,\nYet utter the
word Democratic, the word En-Masse.\n\nOf physiology from top to toe I
sing,\nNot physiognomy alone nor brain alone is worthy for the Muse, I
say\n    the Form complete is worthier far,\nThe Female equally with
the Male I sing.\n\nOf Life immense in passion, pulse, and
power,\nCheerful, for freest action form'd under the laws divine,\nThe
Modern Man I sing.\n\n\n\n}  As I Ponder'd in Silence\n\nAs I ponder'd
in silence,\nReturning upon my poems, c"
```

```
# a regular expression that captures brackets and what is in between them
bracket_re = re.compile(r'\[.*\]')
leaves_norm = bracket_re.sub(" ", leaves)
# regular expression that condenses one or more whitespace characters into a single space
space_re = re.compile(r'\s+')
leaves_norm = space_re.sub(" ", leaves_norm)

leaves_norm[:1000]
```

```
" Come, said my soul, Such verses for my Body let us write, (for we
are one,) That should I after return, Or, long, long hence, in other
spheres, There to some group of mates the chants resuming, (Tallying
Earth's soil, trees, winds, tumultuous waves,) Ever with pleas'd smile
I may keep on, Ever and ever yet the verses owning--as, first, I here
and now Signing for Soul and Body, set to them my name, Walt Whitman }
One's-Self I Sing One's-self I sing, a simple separate person, Yet
utter the word Democratic, the word En-Masse. Of physiology from top
to toe I sing, Not physiognomy alone nor brain alone is worthy for the
Muse, I say the Form complete is worthier far, The Female equally with
the Male I sing. Of Life immense in passion, pulse, and power,
Cheerful, for freest action form'd under the laws divine, The Modern
Man I sing. } As I Ponder'd in Silence As I ponder'd in silence,
Returning upon my poems, considering, lingering long, A Phantom arose
before me with distrustful aspect, Ter"
```

Q4 Previously, we looked at the construction "I X'd". Normalize this past tense suffix to modern English orthography using the past tense suffix "-ed". Make sure to include partial output in your write up.

[8] If you are familiar with Unix, `leaves_norm = space_re.sub(r'\s+', " ", leaves_norm)` is equivalent to `tr -s "[:space:]" " "`. For more information, see the `tr` manual by typing `man tr` at the Unix shell.

Much like the past constructions we just looked at, many other words in *Leaves of Grass* contain apostrophes (to check, search the regular expression `\s\w+\'\w+\s` [9]. Let's normalize one of these cases. First, we could search for words that begin with "o'e", where the apostrophe replaces a "v", (this is often used in the word "over" being written/pronounced as "o'er" as in *Gliding **o'er** all.*), as in then replace the apostrophe with a "v".

```
oer_word = re.compile(r'o\'e\w+')
sorted(set(oer_word.findall(leaves_norm)))
```

```
["o'er", "o'erarches", "o'erarching", "o'ermastering", "o'erweening"]
```

```
oer_replace = re.compile(r'o\'e')
leaves_norm = oer_replace.sub('ove', leaves_norm)
```

Q5 Now, try to normalize another instance of spelling containing an apostrophe (but avoid ambiguous "'s" as it could be possessive *-s* or short for *is*). In your write up, explain the steps you took and give your output.

We might also want to normalize different terms of address in *Leaves of Grass* with `re.sub()` or `replace()`. For example, Whitman mentions "Shakespere" (and "Shakspere"[10]) and other literary figures. We can look at our output by printing part of the string that starts with the name "Job"

```
find_job = re.compile(r'Job.*\n')
find_job.findall(leaves)
```

```
['Job, Homer, Eschylus, Dante, Shakespere, Tennyson, Emerson;\n']
```

```
shakes_re = re.compile(r'Shake?spere')
leaves_norm = shakes_re.sub('William Shakespeare', leaves_norm)
leaves_norm = leaves_norm.replace('Tennyson', 'Alfred Lord Tennyson')
leaves_norm = leaves_norm.replace('Dante', 'Dante Alighieri')
leaves_norm = leaves_norm.replace('Schiller', 'Friedrich Schiller')
leaves_norm = leaves_norm.replace('Emerson', 'Ralph Waldo Emerson')
leaves_norm = leaves_norm.replace('Eschylus', 'Aeschylus')

# since we eliminated line breaks, we should search for 'Job' and 100 characters afterwards
find_job_2 = re.compile(r'Job.{100}')
find_job_2.findall(leaves_norm)
```

```
['Job, Homer, Aeschylus, Dante Alighieri, William Shakespeare, Alfred Lord Tennyson, Ralph Waldo Emerson;']
```

We could also use regular expressions in order to perform tokenization of a text. For example, the Penn Treebank contains 10% of *The Wall Street Journal* from 1987. As you might imagine, money is often mentioned in this newspaper. Let's imagine, counterfactually, that your Treebank sample wasn't tokenized. You could untokenize using `join()`. However, there is still an issue, as there are some non-word characters. This corpus contains traces (i.e. what is "left behind" after syntactic movement[11]). Finally, we can then tokenize using a regular expression. This regular expression must escape the character "$" in order to match the literal character "$" (rather than be used as a special character to assert position at the end of a line). Furthermore, it also contains a **non-capturing group**, which is represented by `?:` inside parentheses. This groups `million`, `billion`, and `thousand` together so that they are recognized as alternatives, but *without* making them a capturing group. If you try this line without the capturing group (i.e. remove `?:`), you should see why this distinction is important; the regular expression will capture and return empty matches.

```
# undo the tokenization for the first one hundred words
' '.join(nltk.corpus.treebank.words()[:100])
```

[9] This regular expression finds all the words in the order they appear in the text, but you can also find the sorted set of these words using `sorted(set(apos_re.findall(leaves)))`, which should make Q5 easier.

[10] There are six surviving Shakespeare signatures. All of them are spelled differently.

[11] For example, "Which story$_t$ did John tell Mary that he liked $\_\_\__t$?"

```
'Pierre Vinken , 61 years old , will join the board as a nonexecutive
director Nov. 29 . Mr. Vinken is chairman of Elsevier N.V. , the Dutch
publishing group . Rudolph Agnew , 55 years old and former chairman of
Consolidated Gold Fields PLC , was named *-1 a nonexecutive director
of this British industrial conglomerate . A form of asbestos once used
* * to make Kent cigarette filters has caused a high percentage of
cancer deaths among a group of workers exposed * to it more than 30
years ago , researchers reported 0 *T*-1 . The'
```

```python
# to undo tokenization while eliminating representation of traces
untokenized = ' '.join(w for w in nltk.corpus.treebank.words() if not w.startswith('*') and w is not '0')
untokenized[:1000]
```

```
"Pierre Vinken , 61 years old , will join the board as a nonexecutive
director Nov. 29 . Mr. Vinken is chairman of Elsevier N.V. , the Dutch
publishing group . Rudolph Agnew , 55 years old and former chairman of
Consolidated Gold Fields PLC , was named a nonexecutive director of
this British industrial conglomerate . A form of asbestos once used to
make Kent cigarette filters has caused a high percentage of cancer
deaths among a group of workers exposed to it more than 30 years ago ,
researchers reported . The asbestos fiber , crocidolite , is unusually
resilient once it enters the lungs , with even brief exposures to it
causing symptoms that show up decades later , researchers said .
Lorillard Inc. , the unit of New York-based Loews Corp. that makes
Kent cigarettes , stopped using crocidolite in its Micronite cigarette
filters in 1956 . Although preliminary findings were reported more
than a year ago , the latest results appear in today 's New England
Journal of Medicine , a forum like"
```

```python
# tokenize using a regular expression
tokenized = nltk.regexp_tokenize(text=untokenized,
pattern=r'[A-Za-z]+|--|[.,?]|\$ [0-9]+ (?:million|billion|thousand)?')

# check to see if it correctly tokenized money expressions together
[ t for t in tokenized if re.search(r'\$', t) ][:20]
```

```
['$ 27 ',
 '$ 212 million',
 '$ 295 million',
 '$ 370 million',
 '$ 101 million',
 '$ 68 billion',
 '$ 71 million',
 '$ 2 billion',
 '$ 250 million',
 '$ 55 million',
 '$ 45 ',
 '$ 50 ',
 '$ 49 million',
 '$ 190 million',
 '$ 195 million',
 '$ 55 million',
 '$ 140 million',
 '$ 245 million',
 '$ 100 million',
 '$ 120 million']
```

Q6 Now, write your own regular expression that does a better job of capturing money expressions (e.g. capturing decimals, etc.). Test it, and turn in the first 100 money expressions that you find in the sample of the *Wall Street Journal* from the Penn Treebank. Comment on what your regular expression found that the one given did not.