CSCI 4140: Natural Language Processing

CSCI/DASC 6040: Computational Analysis of Natural Languages

Spring 2023

Homework 1 - Tokenization and segmentation

Due Sunday, January 29, at 11:59 PM

Part 1: Tokenizer basics - 30 points

Part 1(a) - 10 points

Write a function called `get_words` that takes a string `s` as its only argument. The function should return a list of the words in the same order as they appeared in `s`. Note that in this question a "word" is defined as a "space-separated item". For example:

```
get_words('The cat in the hat ate the rat in the vat')

['The', 'cat', 'in', 'the', 'hat', 'ate', 'the', 'rat', 'in',
'the', 'vat']
```

Hint: If you don't know how to approach this problem, read about str.split().

In [ ]:
```python
# Add your code here
import re

def get_words(s, do_lower=False):
    if do_lower:
        s=s.lower()
    return s.split()

print(get_words('The cat in the hat ate the rat in the vat'))
```

```
['The', 'cat', 'in', 'the', 'hat', 'ate', 'the', 'rat', 'in', 'the', 'vat']
```

Part 1(b) - 10 points

Write a function called `count_words` that takes a list of the words of `s` as its only argument and returns a `collections.Counter` that maps a word to the frequency that it occurred in `s`. Use the output of the `get_words` function as the input to this function.

```
s = 'The cat in the hat ate the rat in the vat'
words = get_words(s)
count_words(words)

Counter({'the': 3, 'in': 2, 'The': 1, 'cat': 1, 'hat': 1,
'ate': 1, 'rat': 1, 'vat': 1})
```

Notice that this is somewhat unsatisfying because **the** is counted separately from **The**. To fix this, have your `get_words` function be able to lower-case all of the words before returning them. You won't want to break any previous code you wrote, though (backwards compatibility is important!), so add a new parameter to `get_words` with a default value:

```
def get_words(s, do_lower=False)
```

Now, if `get_words` is called the way we were using it above, nothing will change. But if we call `get_words(s, do_lower=True)` then `get_words` should lowercase the string before getting the words. You can make use of `str.lower` to modify the string. When you're done, the following should work:

```
s = 'The cat in the hat ate the rat in the vat'
words = get_words(s, do_lower=True)
count_words(words)

Counter({'the': 4, 'in': 2, 'cat': 1, 'hat': 1, 'ate': 1,
'rat': 1, 'vat': 1})
```

In [ ]:
```
# Add your code here
import collections

def count_words(listS):
    return collections.Counter(listS)

s = 'The cat in the hat ate the rat in the vat'
words = get_words(s, do_lower=True)
print(count_words(words))
```

```
Counter({'the': 4, 'in': 2, 'cat': 1, 'hat': 1, 'ate': 1, 'rat': 1, 'vat':
1})
```

Part 1(c) - 10 points

Write a function called `words_by_frequency` that takes a list of words as its only required argument. The function should return a list of `(word, count)` tuples sorted by count such that the first item in the list is the most frequent item. Items with the same frequency should be in the same order they appear in the original list of words.

`words_by_frequency` should, additionally, take a second parameter `n` that specifies the maximum number of results to return. If `n` is passed, then only the `n` most frequent words should be returned. If `n` is not passed, then all words should be returned in order of frequency.

```
words_by_frequency(words)

[('the', 4), ('in', 2), ('cat', 1), ('hat', 1), ('ate', 1),
('rat', 1), ('vat', 1)]


words_by_frequency(words, n=3)

[('the', 4), ('in', 2), ('cat', 1)]
```

In [ ]:
```python
# Add your code here
import collections

def words_by_frequency(listOfWords, n=None):
    return words.most_common(n)

words = count_words(words)
print(words_by_frequency(words, n=3))
```

```
[('the', 4), ('in', 2), ('cat', 1)]
```

Part 2: Through the rabbit hole - 50 points

Next, you will explore some files from Project Gutenberg, a library of free eBooks for texts outside of copyright.

Some of the Gutenberg texts are all available in the `data/gutenberg/` directory.

Part 2(a) - 10 points

Let's use the copy of Lewis Carroll's "Alice's Adventures in Wonderland" from **data/gutenberg/carroll-alice.txt**. Use your `words_by_frequency` and `count_words` functions from Part 1 to explore the text. For the rest of this exercise, you will always lowercase when getting a list of words. You should find that the five most frequent words in the text are:

```
the      1603
and       766
to        706
a         614
she       518
```

Note: If your numbers were right in the previous part, but don't match here, it may be because of how you're calling `split`. Take a look at the documentation for `split` to see if there's a different way you can call it.

Check-In

1. If your `count_words` function is working correctly, it should report that the word **alice** occurs 221 times. Confirm that you get this result with your code.
2. The word **alice** actually appears 398 times in the text, though this is not the answer you got for the previous question. Why? Examine the data to see if you can figure it out before continuing.

In [ ]:
```
# Add your code here
import collections

filepath = "data/gutenberg/carroll-alice.txt"
with open(filepath, "r") as file:
    content = file.read()

words = get_words(content, do_lower=True)
# print(count_words(words))
words = count_words(words)
print(words_by_frequency(words, 5))
```

```
[('the', 1603), ('and', 766), ('to', 706), ('a', 614), ('she', 518)]
```

Part 2(b) - 10 points

A spoiler for 2(a): there is a deficiency in how we implemented the `get_words` function. When we are counting words, we probably don't care whether the word was adjacent to a punctuation mark. For example, the word **hatter** appears in the text 57 times, but if we queried the `count_words` dictionary, we would see it only appeared 24 times. However, it also appeared numerous times adjacent to a punctuation mark, so those instances got counted separately:

```
word_freq = words_by_frequency(words)
for (word, freq) in word_freq:
    if 'hatter' in word:
        print('{:10} {:3d}'.format(word, freq))
```

```
hatter      24
hatter.     13
hatter,     10
hatter:      6
hatters      1
hatter's     1
hatter;      1
hatter.'     1
```

Our `get_words` function would be better if it separated punctuation from words. We can accomplish this by using the `re.split` function. Be sure to import `re` to make `re.split()` work. Below is a small example that demonstrates how `str.split` works on a small text and compares it to using `re.split`:

```
text = '"Oh no, no," said the little Fly, "to ask me is in
vain."'
text.split()
```

```
['"Oh', 'no,', 'no,"', 'said', 'the', 'little', 'Fly,',
'"to', 'ask', 'me', 'is', 'in', 'vain."']
```

```
re.split(r'(\W)', text)
```

```
['', '"', 'Oh', ' ', 'no', ',', '', ' ', 'no', ',', '', '"',
'', ' ', 'said', ' ', 'the',

 ' ', 'little', ' ', 'Fly', ',', '', ' ', '', '"', 'to', ' ',
'ask', ' ', 'me', ' ', 'is',

 ' ', 'in', ' ', 'vain', '.', '', '"', '']
```

Note that this is not exactly what we want, but it is a lot closer. In the resulting list, we find empty strings and spaces, but we have also successfully separated the punctuation from the words.

Using the above example as a guide, write and test a function called `tokenize` that takes a string as an input and returns a list of words and punctuation, but not extraneous spaces and empty strings. Like `get_words`, `tokenize` should take an optional argument `do_lower` that determines whether the string should be case normalized before separating the words. You don't need to modify the `re.split()` line: just remove the empty strings and spaces.

```
tokenize(text, do_lower=True)

['"', 'oh', 'no', ',', 'no', ',', '"', 'said', 'the',
'little', 'fly', ',', '"', 'to', 'ask', 'me', 'is', 'in',
'vain', '.', '"']

print(' '.join(tokenize(text, do_lower=True)))

" oh no , no , " said the little fly , " to ask me is in vain
. "
```

Checking In

Use your `tokenize` function in conjunction with your `count_words` function to list the top 5 most frequent words in **carroll-alice.txt**. You should get this:

```
'        2871      <-- single quote
,        2418      <-- comma
the      1642
.         988      <-- period
and       872
```

In [ ]:
```python
# Add your code here
def tokenize(string, do_lower = True):
    if do_lower:
        string = string.lower()
    re.split(r'(\W)', string)
    return ([w for w in re.split(r'(\W)', string) if w.strip()])

text = '"Oh no, no" said the little Fly, "to ask me is in vain."'
print(tokenize(text, do_lower=True))
print(' '.join(tokenize(text, do_lower=True)))
print(count_words(tokenize(content, do_lower=True)).most_common(5))

# print(count_words(tokenize(content, do_lower=True)).most_common(5))


# print(tokenize(content, do_lower=True))
# print(count_words(tokenize(content, do_lower=True)))

# print(count_words(tokenize(content, do_lower=True),).most_common(5))
```

```
['"', 'oh', 'no', ',', 'no', '"', 'said', 'the', 'little', 'fly', ',', '"',
'to', 'ask', 'me', 'is', 'in', 'vain', '.', '"']
" oh no , no " said the little fly , " to ask me is in vain . "
[('"', 2871), (',', 2418), ('the', 1642), ('.', 988), ('and', 872)]
```

Part 2(c) - 10 points

Write a function called `filter_nonwords` that takes a list of strings as input and returns a new list of strings that excludes anything that isn't entirely alphabetic. Use the `str.isalpha()` method to determine is a string is comprised of only alphabetic characters.

```
text = '"Oh no, no," said the little Fly, "to ask me is in
vain."'
tokens = tokenize(text, do_lower=True)
filter_nonwords(tokens)

['oh', 'no', 'no', 'said', 'the', 'little', 'fly', 'to',
'ask', 'me', 'is', 'in', 'vain']
```

Use this function to list the top 5 most frequent words in **carroll-alice.txt**. Confirm that you get the following before moving on:

```
the      1642
and       872
to        729
a         632
it        595
```

In [ ]:
```python
# Add your code here
def filter_nonwords(lstStrings):
    return [x for x in lstStrings if x.isalpha()]



text = ' "Oh no, no," said the little Fly, "to ask me is in vain."'
tokens = tokenize(text, do_lower=True)
print(filter_nonwords(tokens))
print(count_words(filter_nonwords(tokens)))
tokens = tokenize(content, do_lower=True)
print(count_words(filter_nonwords(tokens)).most_common(5))
# filter_nonwords(tokens)

# tokens = tokenize(text, do_lower=True)
# print(filter_nonwords(text))
# print(tokens)
# print(filter_nonwords(tokens.split()))
# print(words_by_frequency(filter_nonwords(tokens.split())))
```

```
['oh', 'no', 'no', 'said', 'the', 'little', 'fly', 'to', 'ask', 'me', 'is',
'in', 'vain']
Counter({'no': 2, 'oh': 1, 'said': 1, 'the': 1, 'little': 1, 'fly': 1, 'to'
: 1, 'ask': 1, 'me': 1, 'is': 1, 'in': 1, 'vain': 1})
[('the', 1642), ('and', 872), ('to', 729), ('a', 632), ('it', 595)]
```

Part 2(d) - 20 points

Iterate through all of the files in the **gutenberg** data directory and print out the top 5 words for each. To get a list of all the files in a directory, use the `os.listdir` function:

```
import os

directory = 'data/gutenberg/'
files = os.listdir(directory)
infile = open(os.path.join(directory, files[0]), 'r',
encoding='latin1')
```

This example also uses the function `os.path.join` that you might want to read about.

Note about encodings: This `open` function above uses the optional encoding argument to tell Python that the source file is encoded as latin1. Be sure to use this encoding flag to read the files in the **Gutenberg** corpus, as the default (Unicode) won't work!

Token Analysis Questions

Answer the following questions.

1. **Most Frequent Word:** Loop through all the files in the **gutenberg** data directory that end in **.txt**. Is **the** always the most common word? If not, what are some other words that show up as the most frequent word (and in which documents)? What do you notice about these words?
2. **Impact of Lowercasing:** If you don't lowercase all the words before you count them, how does this result change, if at all? Discuss what you observe.

Note: If a question (like the one above) asks you to discuss results, that always means both what the results were and what that implies about the world (i.e., your corpus, your method, etc.). A good answer on this sort of question is a paragraph that goes something like "the result was X, specific interesting examples were X' and X", this is/isn't surprising because it would imply P or Q, this implies it might be better to do Y / to evaluate Z to learn more".

In [ ]:
```
# Add your code here
import os

i=0
directory = 'data/gutenberg/'
files = os.listdir(directory)
infile = open(os.path.join(directory, files[i]), 'r', encoding='latin1')

for filename in files:
    if filename.endswith(".txt"):
        print(filename)
        infile = open(os.path.join(directory, files[i]), 'r', encoding='lati
        print(infile.name)
        with open(infile.name, 'r') as f:
            text = f.read()
            tokens = tokenize(text, do_lower=True)
            print(count_words(filter_nonwords(tokens)).most_common(5))
    i=i+1
```

```
austen-emma.txt
data/gutenberg/austen-emma.txt
[('to', 5239), ('the', 5201), ('and', 4896), ('of', 4291), ('i', 3178)]
austen-persuasion.txt
data/gutenberg/austen-persuasion.txt
[('the', 3329), ('to', 2808), ('and', 2801), ('of', 2570), ('a', 1595)]
austen-sense.txt
data/gutenberg/austen-sense.txt
[('to', 4116), ('the', 4105), ('of', 3572), ('and', 3491), ('her', 2551)]
bible-kjv.txt
data/gutenberg/bible-kjv.txt
[('the', 64023), ('and', 51696), ('of', 34670), ('to', 13580), ('that', 129
12)]
blake-poems.txt
data/gutenberg/blake-poems.txt
[('the', 439), ('and', 348), ('of', 146), ('in', 141), ('i', 130)]
bryant-stories.txt
data/gutenberg/bryant-stories.txt
[('the', 3451), ('and', 2098), ('to', 1180), ('a', 1036), ('he', 1017)]
burgess-busterbrown.txt
data/gutenberg/burgess-busterbrown.txt
[('he', 678), ('the', 660), ('and', 516), ('to', 436), ('of', 342)]
carroll-alice.txt
data/gutenberg/carroll-alice.txt
[('the', 1642), ('and', 872), ('to', 729), ('a', 632), ('it', 595)]
chesterton-ball.txt
data/gutenberg/chesterton-ball.txt
[('the', 4965), ('and', 2667), ('of', 2555), ('a', 2262), ('to', 1580)]
chesterton-brown.txt
data/gutenberg/chesterton-brown.txt
[('the', 4670), ('and', 2221), ('a', 2132), ('of', 2093), ('to', 1391)]
chesterton-thursday.txt
data/gutenberg/chesterton-thursday.txt
[('the', 3636), ('a', 1742), ('of', 1725), ('and', 1658), ('he', 1126)]
```

```
edgeworth-parents.txt
data/gutenberg/edgeworth-parents.txt
[('the', 7728), ('to', 5220), ('and', 4983), ('of', 3745), ('i', 3657)]
melville-moby_dick.txt
data/gutenberg/melville-moby_dick.txt
[('the', 14431), ('of', 6609), ('and', 6430), ('a', 4736), ('to', 4625)]
milton-paradise.txt
data/gutenberg/milton-paradise.txt
[('and', 3395), ('the', 2968), ('to', 2228), ('of', 2050), ('in', 1366)]
shakespeare-caesar.txt
data/gutenberg/shakespeare-caesar.txt
[('and', 627), ('the', 579), ('i', 533), ('to', 446), ('you', 391)]
shakespeare-hamlet.txt
data/gutenberg/shakespeare-hamlet.txt
[('the', 993), ('and', 863), ('to', 685), ('of', 610), ('i', 574)]
shakespeare-macbeth.txt
data/gutenberg/shakespeare-macbeth.txt
[('the', 650), ('and', 546), ('to', 384), ('i', 348), ('of', 338)]
whitman-leaves.txt
data/gutenberg/whitman-leaves.txt
[('the', 10113), ('and', 5334), ('of', 4265), ('i', 2933), ('to', 2244)]
```

1)The seems to be a frequent common word, but in other documents there are cases of other words occurring more often such as 'he' or 'to'. One noticeable thing about these words is that they occur at much smaller amounts than the frequency of 'the' in the other documents. 2) The results from testing with non lowercasing did not vary much from the initial test. This was not a shocking result since the overall occurrence of a certain string tended to be much higher than the next words in a greatest to least order.

Part 3: Sentence segmentation - 30 points

Next, you will write a simple sentence segmenter.

The **data/brown** directory includes three English-language text files taken from the Brown Corpus:

- `editorial.txt`
- `fiction.txt`
- `lore.txt`

These files represent large strings of natural language text, with no line breaks nor other special symbols to annotate where sentence splits occur. In the data set you are working with, sentences can only end with one of 5 characters: period, colon, semi-colon, exclamation point and question mark.

However, there is a catch: not every period represents the end of a sentence. Many abbreviations (U.S.A., Dr., Mon., etc., etc.) that can appear in the middle of a sentence, and the period does not indicate the end of the sentence. (If you have a phone that uses autocomplete to type, you may already have had annoying experiences where it automatically capitalized words after these abbreviations!) These texts also have many examples where colon is not the end of the sentence. The other three punctuation marks are all nearly unambiguously the ends of a sentence (yes, even semi-colons).

For each of the above files, I have also provided a file in the same directory containing the **character index** (counting from 0 for the first character) of each of the actual locations of the ends of sentences:

- `editorial-eos.txt`
- `fiction-eos.txt`
- `lore-eos.txt`

Your job is to write a sentence segmenter, and to output the predicted token number of each sentence boundary.

Part 3(a) - 10 points

Below is some starter code.

In [ ]:

```
def my_best_segmenter(token_list):
    # i = 0
    # all_sentences = []
    # predicted_eos = []
    # this_sentence = []
    # for token in token_list:
    #     this_sentence.append(token)
    #     if token in ['.', ':', ';', '!', '?']:
    #         all_sentences.append(this_sentence)
    #         predicted_eos.append(len(this_sentence) - 1)
    #         sum = sum + len(this_sentence) - 1
    #         this_sentence = []
    pass

def baseline_segmenter(token_list):
    all_sentences = []
    this_sentence = []
    for token in token_list:
        this_sentence.append(token)
        if token in ['.', ':', ';', '!', '?']:
            all_sentences.append(this_sentence)
            this_sentence = []
    return all_sentences

def write_sentence_boundaries(sentence_list, out):
    f = open(out, "w")
    # f = open("data/parsefile.txt", "w")
    all_sentences = []
    predicted_eos = []
    this_sentence = []
    sum = -1
    for sentence in sentence_list:
        for token in sentence:
            this_sentence.append(token)
            if token in ['.', ':', ';', '!', '?']:
                all_sentences.append(token)
                predicted_eos.append(len(this_sentence) - 1)
                sum = sum + len(this_sentence)
                f.write("%s\n" % sum)
                this_sentence = []
    f.close()
    return

""" TODO: Write out the code to parse a text file. """
filepath = "data/brown/editorial.txt"
with open(filepath, "r") as file:
    content = file.read()

print(len(tokenize(content, do_lower=True)))
# print(baseline_segmenter(tokenize(content, do_lower=True)))
write_sentence_boundaries(baseline_segmenter(tokenize(content, do_lower=True
```

```
63333
```

Checking In

Confirm that your code can open the file **data/brown/editorial.txt** and that your code from the previous part splits it into 63,333 tokens.

Note: Do not filter out punctuation, since those tokens will be exactly the ones we want to consider as potential sentence boundaries!

Part 3(b) - 10 points

The starter code contains a function called `baseline_segmenter` that takes a list of tokens as its only argument. It returns a list of tokenized sentences; that is, a list of lists of words, with one list per sentence.

```
baseline_segmenter(tokenize('I am Sam. Sam I am.')

[['I', 'am', 'Sam', '.'], ['Sam', 'I', 'am', '.']]
```

Remember that every sentence in our data set ends with one of the five tokens ['.', ':', ';', '!', '?']. Since it's a baseline approach, `baseline_segmenter` predicts that every instance of one of these characters is the end of a sentence.

Fill in the function `write_sentence_boundaries`. This function takes two arguments: a list of lists of strings (like the one returned by `baseline_segmenter`) and a pointer to a stream to write output (an open write-enabled file). You will need to loop through all of the sentences in the document. For each sentence, you will want to write the index of the last word in the sentence to the filepointer. Remember that Python lists are 0-indexed!

Confirm that when you run `baseline_segmenter` on the file **data/brown/editorial.txt**, it predicts 3278 sentence boundaries, and that the first five predicted boundaries are at tokens 22, 54, 74, 99, and 131.

Part 3(c) - extra credit, 10 points

Now it's time to improve the baseline sentence segmenter. We don't have any false negatives (since we're predicting that every instance of the possibly-end-of-sentence punctuation marks is, in fact, the end of a sentence), but we have quite a few false positives.

There's a placeholder for a second segmentation function defined in the starter code. You will fill in that `my_best_segmenter` function to do a (hopefully!) better job identifying sentence boundaries. The specifics of how you do so are up to you.

Questions

1. Describe (using the metrics from the evaluation script) the performance of your final segmenter.
2. Describe at least 3 things that your final segmenter does better than the baseline segmenter and discuss them. What cases are you most proud of catching in your segmenter? Include specific examples that are handled well.
3. Describe at least 3 places where your segmenter still makes mistakes and discuss them. Include specific examples where your segmenter makes the wrong decision. If you had another week to work on this, what would you add? If you had the rest of the semester to work on it, what would you do?

Your answers go here.