Homework 2: Tokenization

Due Sept. 18 at 10pm

1 Tokenization and Word Frequencies (35 points)

Put all of your code for this part in **tokenizer.py**.

Along with some of the functions we wrote in class, the starter code has an empty **main** function that you should fill in with code to demonstrate how your functions work.

You should call your **main** function using the following pattern:

```
if __name__ == "__main__":
    main()
```

This pattern will be useful for you as you develop more complex python programs. It allows you to write functions that will can be imported into other programs while still having your **tokenizer.py** be runnable as a stand-alone program.

1.1 Project Gutenberg

We will continue exploring files from Project Gutenberg. Start by reading in Lewis Carroll's **Alice's Adventures in Wonderland**, which is stored in the file **carroll-alice.txt**. Use the **words_by_frequency** and **count_words** functions we wrote in class to explore the text. You should find that the five most frequent words in the text are:

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

For this entire assignment, we will lowercase when getting a list of words.

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.

1.2 Punctuation (10 points)

There is a deficiency in how we implemented the **get_words** function! When we calculate word frequencies, 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 our **count_words** Counter says it appears only 24 times. Because it sometimes appears next to a punctuation mark, those instances are being counted separately:

```
>>> word_freq = words_by_frequency(words)
>>> for (word, freq) in word_freq:
...     if "hatter" in word:
...         print(f"{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** at the top of your file 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', ' ',
 ' ', 'little', ' ', 'Fly', ',', '', ' ', '', '"', 'to', ' ', 'ask', ' ', 'me', ' '
 ' ', 'in', ' ', 'vain', '.', '', '"', '']
```

Note that this is not exactly what we want, but it is a lot closer.

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 defaults to False to determine 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
```

1.3 Getting words (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 if 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']
```

Checking In

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
```

1.4 Most frequent words (5 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 = "gutenberg_data"
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: This **open** function 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!

1.5 Token Analysis Questions (10 pts)

Answer the following questions in your assignment writeup, which should be submitted as a PDF on Gradescope along with your Python files.

1. **Most Frequent Word**: Loop through all the files 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?
2. **Impact of Lowercasing**: If you don't lowercase all the words before you count them, how does this result change, if at all?
3. **Impact of Removing Punctuation**: Regenerate the frequency graphs we looked at in class. How does removing punctuation affect the trends we discussed?

2 Sentence Segmentation (65 points)

In this next part of the lab, you will write a simple sentence segmenter.

The brown_data directory includes three text files taken from the Brown Corpus:

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

The files do not indicate where one sentence ends and the next begins. 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. There are many abbreviations ('U.S.A.', 'Dr.', 'Mon.', 'etc.', etc.) that can appear in the middle of a sentence where the periods do not indicate the end of a sentence. There are also 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, there is also a file containing the line number (counting from 0) containing 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. You should write your code in **segmenter.py** and submit this to Gradescope.

2.1 Getting Started

The given **segmenter.py** has some starter code, and can be run from the command line. When it's called from the command line, it takes one required argument and two optional arguments:

```
$ python3 ./segmenter.py --help
usage: segmenter.py [-h] --textfile FILE [--hypothesis_file FILE]

Sentence Segmenter for NLP Lab

optional arguments:
```

```
-h, --help            show this help message and exit
--textfile FILE, -t FILE            Unlabeled text is in FILE.
--hypothesis_file FILE, -y FILE           Write hypothesized boundaries to FILE
```

Make sure you understand how this code uses the **argparse** module to process command-line arguments. (You may find the argparse documentation useful.)

The most confusing part of this is often seeing that argparse is able to open files directly: for instance, the '–textfile' argument given in the segmenter code includes the type **argparse.FileType('r')**, implying that among the resulting parsed arguments, args, you will be able to access a pointer to the text file in "read" mode using **args.textfile**. **Since this file is already open, you should not call open() on it**.

All print statements should be in your **main()** function, which should only be called if **segmenter.py** is run from the command line.

The **segmenter.py** starter code imports the **tokenize** function from the last section.

Checking In

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

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

2.2 Baseline Segmenter (15 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 writeable pointer to either an open file or stdout. You will need to loop through all of the sentencess in the document. For each sentence, you will want to write the index of the **last** word in the sentence to the filepointer.

Checking In

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

2.3 Evaluating the Segmenter

To evaluate your system, I am providing you a program called **evaluate.py** that compares your hypothesized sentence boundaries with the ground truth boundaries. This program will report to you the true positives, true negatives, false positives and false negatives, along with some other metrics (precision, recall, F-measure), which may be new to you. You can run **evaluate.py** with the -h option to see all of the command-line options that it supports.

A sample run with the output of your baseline segmenter from above stored in **editorial.hyp** would be:

```
python3 evaluate.py -d brown -c editorial -y editorial.hyp
```

Run the evaluation on the baseline system's output for the editorial category, and confirm that you get the following before moving on:

```
TP:    2719     FN:       0
FP:     559     TN:   60055

PRECISION: 82.95%       RECALL: 100.00% F: 90.68%
```

(A quick aside: this is a good case for why we like to think about F1 score to help reason about acceptable tradeoffs of precision and recall. If only 25% of the punctuation marks we retrieve were true sentence boundaries, we would get recall of 100% still, precision of 25%, and an F1 of 40%: much lower than the arithmetic average of 62.5% we would get from precision and recall. The further either precision or recall gets from 1, the more it affects the F1 score.)

2.4 Improving the Segmenter (30 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.

You can see the type of tokens that your system is mis-characterizing by setting the verbosity of **evaluate.py** to something greater than 0. Setting it to 1 will print out all of the false positives and false negatives so you can work to improve your **my_best_segmenter** function.

To test your segmenter, I will run it on a hidden text you haven't seen yet. It may not make sense for you to spend a lot of time trying to fix obscure cases that show up in the three texts I am providing you because these cases may never show up in the hidden text that you haven't seen. But it is important to handle cases that occur multiple times and even some cases that appear only once if you suspect they could appear in the hidden text. You want to write your code to be as general as possible so that it works well on the hidden text without leading to too many false positives.

2.5 Segmentation Questions (10 points)

Answer the following questions in your assignment writeup and submit as a PDF on Gradescope:

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. What are you most proud of in your segmenter? Include specific examples that are handled well.
3. Describe at least 3 places where your segmenter still makes mistakes. 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?

2.6 Curiosity

As usual, you will receive 90 points for implementing everything described above. To increase your score further, you can investigate your findings more closely, research segmentation approaches for other languages, or demonstrate intellectual curiosity in some other way.

Please include a section in your report detailing what (if anything) you are submitting for this section.

3 Submission

Submit your **tokenizer.py** and **segmenter.py** files on Gradescope, along with your assignment writeup.