Lesson: Finding Characters

One of the goals of the Cliff Note Generator was to generate a list of characters in a novel. We can actually use our current skill set and include the techniques discussed in the nGrams lesson to extract (with a good level of accuracy) the main characters of a novel.

We will also make some improvements with some of the parsing, cleaning, and preparation of the data. It would be best to read this entire lesson before doing any coding. Also note that this lesson is a bit different in that you will be responsible for more of the code writing. What is being specified is a minimum. We highly recommend that you decompose any complex processes into multiple functions.

Preparation

Before doing anything, read through the entire set of directions first. You will get a sense of the restrictions and overall goals.

Step 1

Fill in the functions from the previous lessons (ngrams and stopwords).

```
#
# from Ngrams and Stopwords Lessons
#

# copy & paste your code
# from ngrams lesson
def read_text(filename):
  # opens filename and returns its contents
  return ""

def split_text_into_tokens(text):
  return []

def bi_grams(tokens):
  return []

def top_n(tokens, n):
  return []

# from stopwords lesson
def remove_stop_words(tokens, stoplist):
  return []

def load_stop_words(filename):
  # read filename (stopwords.txt)
  # and return a list of stopwords
```

Step 2 : Test your code.

The following should now work

```
def demo_test():

   text = read_text('huck.txt')
   stop = load_stop_words('stopwords.txt')

   tokens  = split_text_into_tokens(text)
   cleaned = remove_stop_words(tokens, stop)
   grams = bi_grams(cleaned)

   print(top_n(grams, 10))

demo_test()
```

You should see the following:

```
[(('old', 'man'), 49), (('Mary', 'Jane'), 41), (('Tom', 'Sawyer'), 40),
(('Aunt', 'Sally'), 39), (('pretty', 'soon'), 37),
(('never', 'see'), 33), (('ever', 'see'), 29), (('Jim', 'said'), 28),
(('every', 'time'), 25), (('come', 'along'), 24)]
```

Note how this compares to the output when we didn't account for the case of the stopword.

Finding the Characters

With this machinery in place, we are ready to find characters in a novel (I hope you are reading this with great anticipation) using different strategies. Each of the strategies has a function to implement that strategy.

Method #1

One attribute (or feature) of the text we are analyzing is that proper nouns are capitalized. Let's capitalize on this and find all single words in the text whose first character is an uppercase letter and the word is **not** a stop word.

Create and define the function find_characters_v1(text, stoplist, top):

- Tokenize and clean the text using the function split_text_into_tokens
- Filter the tokens so it has no stop words in it (regardless of case). The parameter stoplist is the array returned from load_stop_words

- Create a new list of tokens (keep the order) of words that are capitalized. You can test the first character of the token.
- Return the top words as a list of tuples (the first element is the word, the second is the count)

```
def find_characters_v1(text, stoplist, top):
  return []
```

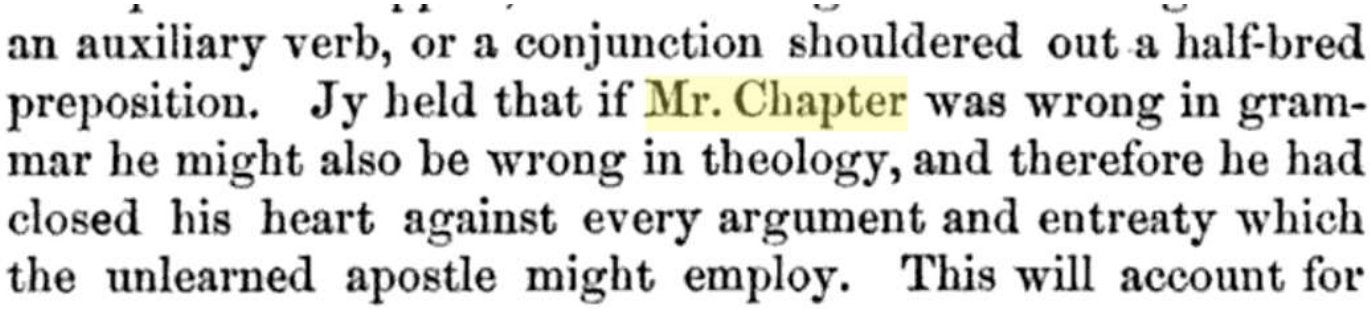
For Huck Finn, you should get the following (the output is formatted for clarity):

```
text = read_text('huck.txt')
stop = load_stop_words('stopwords.txt')
v1 = find_characters_v1(text,stop, 15)
print(v1)
```

You should see:

```
('Jim', 341),
('Well', 318),
('Tom', 217),
('Huck', 70),
('Yes', 68),
('Oh', 65),
('Miss', 63),
('Mary', 60),
('Aunt', 53),
('Now', 53),
('Sally', 46),
('CHAPTER', 43),
('Sawyer', 43),
('Jane', 43),
('Buck', 38)
```

Notice with this very simple method we found 8 characters in the top 15 (those in bold). You also found an Aunt and a Miss too. You might be inclined to start fiddling with the stop-words. The one you could add is 'CHAPTER' and 'Well' -- the interjection, since we know that word does not provide much content in this context. But as we mentioned in the stop words lesson, that's a dangerous game, since other novels might include some of these:


an auxiliary verb, or a conjunction shouldered out a half-bred preposition. Jy held that if Mr. Chapter was wrong in grammar he might also be wrong in theology, and therefore he had closed his heart against every argument and entreaty which the unlearned apostle might employ. This will account for

Method #2

Another feature of characters in a novel is that many of them have two names (Tom Sawyer, Aunt Polly, etc).

Create and define the following function:

```
find_characters_v2(text, stoplist, top)
```

- Tokenize and clean the text using the function split_text_into_tokens
- Convert the list of tokens into a list of bigrams (using your bi_grams method)
- Filter out all bigrams to keep only the ones where both words are capitalized (just the first character)
- Neither word should (either lower or upper) be in stoplist
- Remember stoplist could be an empty list
- Return the top bigrams as a list of tuples: The first element is the bigram tuple, the second is the count

Note that we are **not** removing the stopwords from the text. We are now using the stopwords to make decisions on the text. The stopwords lesson has more details on this as well.

```
def find_characters_v2(text, stoplist, top):
  return []


v2  = find_characters_v2(text, [], 15)
print(v2)
```

With the text of Huckleberry Finn, the following is the output with stopwords being the empty list:

```
 (('Mary', 'Jane'), 41),
 (('Tom', 'Sawyer'), 40),
 (('Aunt', 'Sally'), 39),
 (('Miss', 'Watson'), 20),
```

```
(('Miss', 'Mary'), 19),
(('Mars', 'Tom'), 16),
(('Huck', 'Finn'), 15),
(('Uncle', 'Silas'), 15),
(('Aunt', 'Polly'), 11),
(('Judge', 'Thatcher'), 10),
(('But', 'Tom'), 9),
(('Ben', 'Rogers'), 8),
(('So', 'Tom'), 9),
(('St', 'Louis'), 7),
(('Miss', 'Sophia'), 7)
```

That found 11 characters in the top 15 bigrams frequency table. This method is pretty good and the method didn't even consider stop words. What happens if you consider stop words?

Note: in order to match these outputs, use the collections.Counter class. Otherwise, it's possible that your version of sorting will handle those tuples with equal counts differently (unstable sorting).

Titles, a short diversion

Another feature of characters is that many of them have a title (also called honorifics) precede them (Dr. Mr. Mrs. Miss. Ms. Rev. Prof. Sir. etc). We will look for bi-grams that have these titles. However, we will **not** hard code the titles (we won't specify which titles to look for). We will let the data tell us what the 'titles' are.

Here's the process to use to self discover titles:

- Let's define a title as a capital letter followed by 1 to 3 lower case letters followed by a period. This is not perfect, but it captures a good majority of them.
- Create a list named title_tokens whose text matches the above criteria (hint: use regular expressions) for example: title_tokens = regex1.findall(text)
- Now we need to remove words that might have ended a sentence with those same title characteristics (e.g. Tom. Bill. Pat. Etc.). These names could have been in a sentence like "Please go Tom." Tom is **not** a title, but it would have been found by our definition.
- Use the same definition for titles (above) but instead of ending with a period, the token must end with whitespace. The idea is that hopefully somewhere in the text the same name will appear but without a period. It's very likely that you would encounter 'Tom' somewhere in the text without a period, but it's unlikely that Mr., Mrs., Dr., etc would appear without a period. Let's call this list pseudo_titles. pseudo_titles = regex2.findall(text)
- The set of titles is essentially the first list of tokens, title_tokens with all the tokens in the second set (pseudo_titles) removed. For example, the first list might have 'Dr.', 'Tom.' and 'Mr.' in it and the second set might have 'Tom' and 'Ted' in it. The final title list would be ['Dr', 'Mr'].

Name your function get_titles that encapsulates the above logic; it should return a list of titles:

```
def get_titles(txt):
   return [] # see process above
```

Once you have get_titles working, the following should work:

```
text = read_text('huck.txt')
titles = get_titles(text)
print(sorted(titles))
```

You should get 7 computed titles in Huckleberry Finn:

```
['Col', 'Dr', 'Mr', 'Mrs', 'Otto', 'Rev', 'St']
```

Do not move forward until this is working.

Method #3

Create and define the following function

find_characters_v3(text, stoplist, top)

- Tokenize and clean the text
- Convert the list of tokens into a list of bigrams
- Filter out all bigrams such that the first word in the bigram is a title and the second word is capitalized (hint: use the output of get_titles) **and** the second word (either lower or upper) should not be in stoplist
- Return the top bigrams as a list of tuples: the first element is the bigram tuple, the second is the count

```
def find_characters_v3(text, stoplist, top):
  return []


text = read_text('huck.txt')
stop = load_stop_words('stopwords.txt')
v3 = find_characters_v3(text, stop, 15)
print(v3)
```

For Huck Finn, you should get the following:

```
(('St', 'Louis'), 7),
(('Mr', 'Lothrop's'), 6),
(('Mrs', 'Phelps'), 4),
(('St', 'Petersburg'), 3),
(('Dr', 'Robinson'), 3),
(('Mr', 'Garrick'), 2),
(('Mr', 'Kean'), 2),
(('Mr', 'Wilks'), 2),
(('Mr', 'Mark'), 1),
(('Mrs', 'Judith'), 1),
(('Mr', 'Parker'), 1),
(('Dr', 'Gunn's'), 1),
(('Col', 'Grangerford'), 1),
(('Dr', 'Armand'), 1),
(('St', 'Jacques'), 1)
```

Clearly, that yields a lot of good information. Although looking at the counts, none of them are that

Machine Learning?

You may have heard of (and used) the NLTK Python library that's a popular choice for processing text. These libraries include models that were built by processing large amounts of text. We will use both the NLTK and SpaCy NLP libraries to do something similar in another lesson. However, these libraries have models built from using large data sets to extract entities (called NER for named entity recognition). These entities include organizations, people, places, money.

The models that were built essentially learned what features (like capitalization or title words) were important when analyzing text and came up with a model that attempts to do the same thing we did here. However, we hard coded the rules (use bigrams, remove stop words, look for capital letters, etc). This is sometimes referred to as a rule-based system. The analysis is built on manually crafted rules.

In machine learning (sometimes referred to as an automatic system), some of the algorithms essentially learn what features are important (or can learn how much weight to apply to each feature) to build a model and then uses the model to classify tokens as named entities. The biggest issue is that these models could be built with a very different text source (e.g. journal articles or twitter feed) than what you are processing. Also the models themselves require a large set of resources (memory, cpu) that you may not have available. What you built in this lesson is efficient, fast and fairly accurate.

In the follow-on course, you'll be able to build your own text-based models.

Submission

After implementing all the functions and testing them please download the notebook as "solution.py" and submit to gradescope under "Week13:Project:Finding_Characters" assignment tab and Moodle.

NOTES

- Be sure to use the function names and parameter names as given.
- DONOT use your own function or parameter names.
- Your file MUST be named "solution.py".
- Comment out any lines of code and/or function calls to those functions that produce errors. If your solution has errors, then you have to work on them but if there were any errors in the examples/exercies then comment them before submitting to Gradescope.
- Grading cannot be performed if any of these are violated.