Info

Name: Jacqui Unciano **Date:** Feb 9, 2024 **Assignment:** HW04

Directions:

In this week’s homework, you will use NLTK to help tokenize and annotate a small corpus of George Eliot's novels to create an F3 level digital analytical edition from them.

Using this week's Lab notebook M04_01_Pipeline.ipynb as a guide, import and combine the novels contained in the course repo directory /data/gutenberg/eliot-set.

You should produce the following related dataframes:

In [1]:

```
import pandas as pd
import numpy as np
import configparser
from glob import glob
import re
import nltk

config = configparser.ConfigParser()
config.read('../../../env.ini')
data_home = config['DEFAULT']['data_home']
output_dir = config['DEFAULT']['output_dir']
```

In [2]:

```
source_files = f'{data_home}/gutenberg/eliot-set'
OHCO = ['book_id', 'chap_num', 'para_num', 'sent_num', 'token_num']
```

In [3]:

```
import sys
local_lib = config['DEFAULT']['local_lib']
sys.path.append(local_lib)
from textimporter import TextImporter
from textparser import TextParser
```

```
src_file = f'{data_home}/gutenberg/eliot-set/ELIOT_GEORGE_THE_MILL_ON_THE_FLOSS-pg6688.txt' LINES =
pd.DataFrame(open(src_file, 'r', encoding='utf-8-sig').readlines(), columns=['line_str']) LINES.index.name = 'line_num'
LINES.line_str = LINES.line_str.str.replace(r'\n+', ' ', regex=True).str.strip() LINES.sample(5)clip_pats = [ r"\*\*\*\s*START OF
(?:THE|THIS) PROJECT", r"\*\*\*\s*END OF (?:THE|THIS) PROJECT" ] line_a = 0 line_b = len(LINES) try: pat_a =
LINES.line_str.str.match(clip_pats[0]) line_a = LINES.loc[pat_a].index[0] + 1 except: print("pat_a not found") try: pat_b =
LINES.line_str.str.match(clip_pats[1]) line_b = LINES.loc[pat_b].index[0] - 1 except: print("pat_b not found") LINES =
LINES.loc[line_a : line_b] LINES.head()LINES.tail()roman = '[IVXLCM]+' chap_pat = rf"^\s*Chapter\s+{roman}\.$" chap_lines
= LINES.line_str.str.match(chap_pat, case=False) LINES.loc[chap_lines, 'chap_num'] = [i+1 for i in
range(LINES.loc[chap_lines].shape[0])] LINES.loc[chap_lines].head()
```

A library LIB with the following metadata (and data) about each book:

1. The book_id, matching the first level of the index in the CORPUS.
2. The raw book title will be sufficient, i.e. with title and author combined.
3. The path of the source file.
4. The regex used to parse chapter milestones.

5. The length of the book (number of tokens).
6. The number of chapters in the book.

In [4]:

```
source_file_list = sorted(glob(f"{source_files}/*.*"))
```

In [5]:

```
book_data = []
for source_file_path in source_file_list:
    book_id = int(source_file_path.split("\\")[-1].split("-")[-1].split(".")[0].rep
    book_title = source_file_path.split("\\")[-1].split("-")[0]
    book_data.append((book_id, source_file_path, book_title))
```

In [6]:

```
LIB = pd.DataFrame(book_data, columns=['book_id','source_file_path','raw_title'])\
    .set_index('book_id').sort_index()
LIB.head()
```

Out[6]:

	source_file_path	raw_title
book_id		
145	/Users/jacqu/OneDrive/Documents/MSDS-at-UVA-20...	ELIOT_GEORGE_MIDDLEMARCH
507	/Users/jacqu/OneDrive/Documents/MSDS-at-UVA-20...	ELIOT_GEORGE_ADAM_BEDE
6688	/Users/jacqu/OneDrive/Documents/MSDS-at-UVA-20...	ELIOT_GEORGE_THE_MILL_ON_THE_FLOSS

In [7]:

```
roman = '[IVXLCM]+'
ohco_pat_list = [
    (145,   rf"^\s*((PRELUDE\.)|(CHAPTER\s+{roman}\.))$"),
    (507,   rf"^\s*(Chapter\s+{roman}|Epilogue)$"),
    (6688,  rf"^\s*Chapter\s+{roman}\.$")]
clip_pats = [
    r"\*\*\*\s*START OF (?:THE|THIS) PROJECT",
    r"\*\*\*\s*END OF (?:THE|THIS) PROJECT"
]
```

In [8]:

```
LIB['chap_regex'] = LIB.index.map(pd.Series({x[0]:x[1] for x in ohco_pat_list}))
LIB.head()
```

Out[8]:

book_id	source_file_path	raw_title	
145	/Users/jacqu/OneDrive/Documents/MSDS-at-UVA-20...	ELIOT_GEORGE_MIDDLEMARCH	
507	/Users/jacqu/OneDrive/Documents/MSDS-at-UVA-20...	ELIOT_GEORGE_ADAM_BEDE	[IV
6688	/Users/jacqu/OneDrive/Documents/MSDS-at-UVA-20...	ELIOT_GEORGE_THE_MILL_ON_THE_FLOSS	

text = TextParser(src_file = LIB.iloc[0].source_file_path, ohco_pats=[('chap', LIB.iloc[0].chap_regex, 'm')], clip_pats = clip_pats, use_nltk=True) text.import_source() text.parse_tokens() text.extract_vocab() TOKEN = text.TOKENS TOKEN.sample(10)

In [9]:

```python
num_tokens = []
num_chaps = []
blist = []

for i in range(len(LIB)):
    text = TextParser(src_file = LIB.iloc[i].source_file_path,
                      ohco_pats=[('chap', LIB.iloc[i].chap_regex, 'm')],
                      clip_pats = clip_pats,
                      use_nltk=True)
    text.import_source()
    text.parse_tokens()
    text.extract_vocab()
    TOKEN = text.TOKENS
    CHAPS = text.gather_tokens(level=0)
    num_tokens.append((LIB.index[i], len(TOKEN)))
    num_chaps.append((LIB.index[i], len(CHAPS)))
    blist.append((LIB.index[i], TOKEN))

LIB['num_tokens'] = LIB.index.map(pd.Series({x[0]:x[1] for x in num_tokens}))
LIB['num_chaps'] = LIB.index.map(pd.Series({x[0]:x[1] for x in num_chaps}))
```

```
C:\Users/jacqu/OneDrive/Documents/MSDS-at-UVA-2023/DS5001/repo/lessons/lib\textparse
r.py:132: UserWarning: This pattern is interpreted as a regular expression, and has
match groups. To actually get the groups, use str.extract.
  div_lines = self.TOKENS[src_col].str.contains(div_pat, regex=True, case=True) # TO
DO: Parametize case
line_str chap_str
Index(['chap_str'], dtype='object')
C:\Users/jacqu/OneDrive/Documents/MSDS-at-UVA-2023/DS5001/repo/lessons/lib\textparse
r.py:132: UserWarning: This pattern is interpreted as a regular expression, and has
match groups. To actually get the groups, use str.extract.
  div_lines = self.TOKENS[src_col].str.contains(div_pat, regex=True, case=True) # TO
DO: Parametize case
line_str chap_str
Index(['chap_str'], dtype='object')
line_str chap_str
Index(['chap_str'], dtype='object')
```

In [10]:
```
LIB.head()
```

Out[10]:

	source_file_path	raw_title	
book_id			
145	/Users/jacqu/OneDrive/Documents/MSDS-at-UVA-20...	ELIOT_GEORGE_MIDDLEMARCH	
507	/Users/jacqu/OneDrive/Documents/MSDS-at-UVA-20...	ELIOT_GEORGE_ADAM_BEDE	[IV
6688	/Users/jacqu/OneDrive/Documents/MSDS-at-UVA-20...	ELIOT_GEORGE_THE_MILL_ON_THE_FLOSS	

A an aggregate of all the novels' tokens CORPUS with an appropriate OHCO index, with following features:

1. The token string.
2. The term string.
3. The part-of-speech tag inferred by NLTK.TK.

In [11]:
```
CORPUS = pd.concat([i[1] for i in blist], keys=[i[0] for i in blist])
CORPUS.index.names = OHCO[:]
```

In [12]:
```
CORPUS.sample(10)
```

Out[12]:

					pos_tuple	pos	token_str	term_str
book_id	**chap_num**	**para_num**	**sent_num**	**token_num**				
6688	**25**	**58**	**2**	**9**	(s, NN)	NN	s	s
507	**52**	**4**	**3**	**7**	(thee—I, NN)	NN	thee—I	theei
145	**69**	**18**	**3**	**16**	(., .)	.	.	NaN
507	**53**	**24**	**2**	**6**	(be, VB)	VB	be	be
	18	**11**	**6**	**7**	(to, TO)	TO	to	to
145	**55**	**38**	**0**	**33**	(and, CC)	CC	and	and
	35	**4**	**9**	**6**	(being, VBG)	VBG	being	being
507	**45**	**2**	**6**	**65**	(,, ,)	,	,	NaN
6688	**14**	**9**	**7**	**9**	(by, IN)	IN	by	by
	58	**54**	**3**	**19**	(,, ,)	,	,	NaN

A vocabulary VOCAB of terms extracted from CORPUS, with the following annotation features derived from either NLTK or by using operations presented in the notebook:

1. Stopwords.
2. Porter stems.
3. Maximum POS; i.e. the most frequently associated POS tag for the term using .idxmax(). Note that ties are handled by the method.
4. POS ambiguity expressed a number of POS tags associated with a term's tokens.ens.

In [13]:

```
VOCAB = CORPUS.groupby(level=0)['term_str'].value_counts().to_frame('n')
VOCAB.index.names = ['book_id', 'term_str']
# VOCAB.head()
```

In [14]:

```
sw = pd.DataFrame(nltk.corpus.stopwords.words('english'), columns=['term_str'])
sw = sw.reset_index().set_index('term_str')
sw.columns = ['dummy']
sw.dummy = 1
```

sw.head()

In [15]:

```
VOCAB['stop'] = VOCAB.index.get_level_values(1).map(sw.dummy)
VOCAB['stop'] = VOCAB['stop'].fillna(0).astype('int')
```

In [16]:

```
VOCAB[VOCAB.stop == 1].sample(10)
```

Out[16]:

book_id	term_str	n	stop
507	they	750	1
	our	255	1
6688	only	354	1
	been	732	1
	needn	11	1
	after	226	1
507	ours	5	1
	through	143	1
6688	such	214	1
507	only	315	1

In [17]:

```
from nltk.stem.porter import PorterStemmer
stemmer = PorterStemmer()
VOCAB['stem_porter'] = VOCAB.index.get_level_values(1).map(lambda x: stemmer.stem(x

from nltk.stem.snowball import SnowballStemmer
stemmer2 = SnowballStemmer("english")
VOCAB['stem_snowball'] = VOCAB.index.get_level_values(1).map(lambda x: stemmer2.ste

from nltk.stem.lancaster import LancasterStemmer
stemmer3 = LancasterStemmer()
VOCAB['stem_lancaster'] = VOCAB.index.get_level_values(1).map(lambda x: stemmer3.st

VOCAB.sample(10)
```

Out[17]:

		n	stop	stem_porter	stem_snowball	stem_lancaster
book_id	**term_str**					
6688	**agreeable**	25	0	agreeabl	agreeabl	agr
	dictated	2	0	dictat	dictat	dict
507	**sky**	19	0	sky	sky	sky
145	**slips**	2	0	slip	slip	slip
6688	**unmapped**	1	0	unmap	unmap	unmap
507	**convulsion**	1	0	convuls	convuls	convuls
145	**casting**	3	0	cast	cast	cast
	bedrooms	1	0	bedroom	bedroom	bedroom
	fling	4	0	fling	fling	fling
	tickled	1	0	tickl	tickl	tickl

In [18]:

```
VOCAB['max_pos'] = CORPUS.groupby(level=0)[['term_str','pos']].value_counts().unsta
```

In [19]:

```
VOCAB['n_pos'] = CORPUS.groupby(level=0)[['term_str','pos']].value_counts().unstack
VOCAB['cat_pos'] = CORPUS.groupby(level=0)[['term_str','pos']].value_counts().to_fr
    .groupby(['book_id','term_str']).pos.apply(lambda x: set(x))
```

In [20]:

```
VOCAB.sample(10)
```

Out[20]:

		n	stop	stem_porter	stem_snowball	stem_lancaster	max_pos
book_id	**term_str**						
507	**footsteps**	4	0	footstep	footstep	footstep	NNS
	unlocked	2	0	unlock	unlock	unlock	VBD
6688	**corpses**	2	0	corps	corps	corps	NNS
507	**insist**	2	0	insist	insist	insist	NN
6688	**godmother**	3	0	godmoth	godmoth	godmoth	NN
145	**mothers**	7	0	mother	mother	moth	NNS
507	**goodfornought**	1	0	goodfornought	goodfornought	goodfornought	JJ
	comforted	9	0	comfort	comfort	comfort	VBN
6688	**wrapped**	4	0	wrap	wrap	wrap	VBD
145	**oblivion**	1	0	oblivion	oblivion	obl	NN

Once you have these, use the dataframes to answer these questions:

Question 1: What regular expression did you use to chunk *Middlemarch* into chapters?s?

In [21]:

```
LIB.chap_regex.loc[145]
```

Out[21]:

```
'^\\s*((PRELUDE\\.)|(CHAPTER\\s+[IVXLCM]+\\.))$'
```

Question 2: What is the title of the book that has the most tokens?

In [22]:

```
LIB.loc[LIB['num_tokens'].idxmax()].raw_title
```

Out[22]:

```
'ELIOT_GEORGE_MIDDLEMARCH'
```

Question 3: How many chapter level chunks are there in this novel?

In [23]:

```
LIB.loc[LIB['num_tokens'].idxmax()].num_chaps
```

Out[23]:

```
87
```

Question 4: Among the three stemming algorithms -- Porter, Lancaster, and Snowball -- which is the most aggressive, in terms of the number of words associated with each stem?

In [24]:

```
p = VOCAB.stem_porter.value_counts().to_frame()
p.sort_values(by='count', ascending=False).head()
```

Out[24]:

	count
stem_porter	
admir	26
observ	26
respect	24
impress	24
continu	23

In [25]:

```python
s = VOCAB.stem_snowball.value_counts().to_frame()
s.sort_values(by='count', ascending=False).head()
```

Out[25]:

	count
stem_snowball	
admir	29
observ	26
wonder	25
impress	24
respect	24

In [26]:

```python
l = VOCAB.stem_lancaster.value_counts().to_frame()
l.sort_values(by='count', ascending=False).head()
```

Out[26]:

	count
stem_lancaster	
cont	74
man	47
adv	44
pass	43
not	39

Answer 4: I would say that Lancaster was the most aggressive since there are a lot of terms that have been chopped into their supposed stem. It really goes at it with stemming the words.

Question 5: Using the most aggressive stemmer from the previous question, what is the stem with the most associated terms?

```
In [27]: l.sort_values(by='count', ascending=False).head(1)
```

```
Out[27]:
                    count
 stem_lancaster
           cont        74
```

Answer 5: Using the Lancaster stemmer, the stem with the most associated terms is 'cont'.