M04 Homework

Michael Vaden, mtv2eva

In [ ]:
```
import pandas as pd
import numpy as np
from glob import glob
import re
import nltk
import plotly_express as px
from lib.textparser import TextParser
```

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

In [ ]:
```
source_files = f'{data_home}/eliot-set'
data_prefix = 'ELIOT_GEORGE'
```

In [ ]:
```
OHCO = ['book_id', 'chap_num', 'para_num', 'sent_num', 'token_num']
```

In [ ]:
```
source_file_list = sorted(glob(f"{source_files}/*.*"))

#source_file_list
book_data = []

for source_file_path in source_file_list:
    book_id = int(source_file_path.split('-')[-1].split('.')[0].replace('pg',''))
    book_title = source_file_path.split('/')[-1].split('-')[0].replace('_', ' ')
    book_data.append((book_id, source_file_path, book_title))

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

Out[ ]:

book_id	source_file_path	raw_title
145	/Users/michaelvaden/GithubRepos/DS5001-Workpla...	ELIOT GEORGE MIDDLEMARCH
507	/Users/michaelvaden/GithubRepos/DS5001-Workpla...	ELIOT GEORGE ADAM BEDE
6688	/Users/michaelvaden/GithubRepos/DS5001-Workpla...	ELIOT GEORGE THE MILL ON THE FLOSS

In [ ]:
```
clip_pats = [
    r"\*\*\*\s*START OF",
    r"\*\*\*\s*END OF"
]

# All are 'chap'and 'm'
roman = '[IVXLCM]+'
caps = "[A-Z';, -]+"
ohco_pat_list = [
    (145, rf"^\s*CHAPTER\s+{roman}\.s*$"),
    (507,   rf"^\s*Chapter\s+{roman}\s*$"),
    (6688,   rf"^\s*Chapter\s+{roman}\.s*$")
]
```

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

In [ ]:
```
LIB
```

Out[ ]:

book_id	source_file_path	raw_title	chap_regex
145	/Users/michaelvaden/GithubRepos/DS5001-Workpla...	ELIOT GEORGE MIDDLEMARCH	^\s*CHAPTER\s+[IVXLCM]+\.s*$
507	/Users/michaelvaden/GithubRepos/DS5001-Workpla...	ELIOT GEORGE ADAM BEDE	^\s*Chapter\s+[IVXLCM]+\s*$
6688	/Users/michaelvaden/GithubRepos/DS5001-Workpla...	ELIOT GEORGE THE MILL ON THE FLOSS	^\s*Chapter\s+[IVXLCM]+\.s*$

In [ ]:
```
def tokenize_collection(LIB):

    clip_pats = [
        r"\*\*\*\s*START OF",
        r"\*\*\*\s*END OF"
```

```
    ]

    books = []
    for book_id in LIB.index:

        # Announce
        print("Tokenizing", book_id, LIB.loc[book_id].raw_title)

        # Define vars
        chap_regex = LIB.loc[book_id].chap_regex
        ohco_pats = [('chap', chap_regex, 'm')]
        src_file_path = LIB.loc[book_id].source_file_path

        # Create object
        text = TextParser(src_file_path, ohco_pats=ohco_pats, clip_pats=clip_pats, use_nltk=True)

        # Define parameters
        text.verbose = True
        text.strip_hyphens = True
        text.strip_whitespace = True

        print(text)

        # Parse
        text.import_source().parse_tokens()

        # Name things
        text.TOKENS['book_id'] = book_id
        text.TOKENS = text.TOKENS.reset_index().set_index(['book_id'] + text.OHCO)

        # Add to list
        books.append(text.TOKENS)

    # Combine into a single dataframe
    CORPUS = pd.concat(books).sort_index()

    # Clean up
    del(books)
    del(text)

    print("Done")

    return CORPUS
```

In [ ]:
```
CORPUS = tokenize_collection(LIB)
```

```
Tokenizing 145 ELIOT GEORGE MIDDLEMARCH
<lib.textparser.TextParser object at 0x7fe9a9155820>
Importing  /Users/michaelvaden/GithubRepos/DS5001-Workplace/data/eliot-set/ELIOT_GEORGE_MIDDLEMARCH-pg145.txt
Clipping text
Parsing OHCO level 0 chap_id by milestone ^\s*CHAPTER\s+[IVXLCM]+\.s*$
line_str chap_str
Index(['chap_str'], dtype='object')
Parsing OHCO level 1 para_num by delimitter \n\n
Parsing OHCO level 2 sent_num by NLTK model
Parsing OHCO level 3 token_num by NLTK model
Tokenizing 507 ELIOT GEORGE ADAM BEDE
<lib.textparser.TextParser object at 0x7fe98802a550>
Importing  /Users/michaelvaden/GithubRepos/DS5001-Workplace/data/eliot-set/ELIOT_GEORGE_ADAM_BEDE-pg507.txt
Clipping text
Parsing OHCO level 0 chap_id by milestone ^\s*Chapter\s+[IVXLCM]+\s*$
line_str chap_str
Index(['chap_str'], dtype='object')
Parsing OHCO level 1 para_num by delimitter \n\n
Parsing OHCO level 2 sent_num by NLTK model
Parsing OHCO level 3 token_num by NLTK model
Tokenizing 6688 ELIOT GEORGE THE MILL ON THE FLOSS
<lib.textparser.TextParser object at 0x7fe9a9150fa0>
Importing  /Users/michaelvaden/GithubRepos/DS5001-Workplace/data/eliot-set/ELIOT_GEORGE_THE_MILL_ON_THE_FLOSS-pg6688.tx
t
Clipping text
Parsing OHCO level 0 chap_id by milestone ^\s*Chapter\s+[IVXLCM]+\.s*$
line_str chap_str
Index(['chap_str'], dtype='object')
Parsing OHCO level 1 para_num by delimitter \n\n
Parsing OHCO level 2 sent_num by NLTK model
Parsing OHCO level 3 token_num by NLTK model
Done
```

A library LIB with the following metadata (and data) about each book: The book_id, matching the first level of the index in the CORPUS. The raw book title will be sufficient, i.e. with title and author combined. The path of the source file. The regex used to parse chapter milestones. The length of the book (number of tokens). The number of chapters in the book.

In [ ]:
```
LIB['book_len'] = CORPUS.groupby('book_id').term_str.count()
```

```
LIB['n_chaps'] = CORPUS.reset_index()[['book_id','chap_id']]\
    .drop_duplicates()\
    .groupby('book_id').chap_id.count()

LIB
```

Out[ ]:

book_id	source_file_path	raw_title	chap_regex	book_len	n_chaps
145	/Users/michaelvaden/GithubRepos/DS5001-Workpla...	ELIOT GEORGE MIDDLEMARCH	^\s*CHAPTER\s+[IVXLCM]+\.s*$	317305	86
507	/Users/michaelvaden/GithubRepos/DS5001-Workpla...	ELIOT GEORGE ADAM BEDE	^\s*Chapter\s+[IVXLCM]+\s*$	215404	55
6688	/Users/michaelvaden/GithubRepos/DS5001-Workpla...	ELIOT GEORGE THE MILL ON THE FLOSS	^\s*Chapter\s+[IVXLCM]+\.s*$	207461	58

An aggregate of all the novels' tokens CORPUS with an appropriate OHCO index, with following features: The token string. The term string. The part-of-speech tag inferred by NLTK.

In [ ]:
```
CORPUS
```

Out[ ]:

book_id	chap_id	para_num	sent_num	token_num	pos_tuple	pos	token_str	term_str
145	1	0	0	0	(Since, IN)	IN	Since	since
				1	(I, PRP)	PRP	I	i
				2	(can, MD)	MD	can	can
				3	(do, VB)	VB	do	do
				4	(no, DT)	DT	no	no
...	...	...	...	...	...	...	...	...
6688	58	69	0	2	(death, NN)	NN	death	death
				3	(they, PRP)	PRP	they	they
				4	(were, VBD)	VBD	were	were
				5	(not, RB)	RB	not	not
				6	(divided.", JJ)	JJ	divided."	divided

740213 rows × 4 columns

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

In [ ]:
```
CORPUS = CORPUS[CORPUS.term_str != '']
CORPUS['pos_group'] = CORPUS.pos.str[:2]


VOCAB = CORPUS.term_str.value_counts().to_frame('n').sort_index()
VOCAB.index.name = 'term_str'
VOCAB['n_chars'] = VOCAB.index.str.len()
VOCAB['p'] = VOCAB.n / VOCAB.n.sum()
VOCAB['i'] = -np.log2(VOCAB.p)

VOCAB['max_pos'] = CORPUS[['term_str','pos']].value_counts().unstack(fill_value=0).idxmax(1)
VOCAB['max_pos_group'] = CORPUS[['term_str','pos_group']].value_counts().unstack(fill_value=0).idxmax(1)

VOCAB['n_pos_group'] = CORPUS[['term_str','pos_group']].value_counts().unstack().count(1)
VOCAB['cat_pos_group'] = CORPUS[['term_str','pos_group']].value_counts().to_frame('n').reset_index()\
    .groupby('term_str').pos_group.apply(lambda x: set(x))
VOCAB['n_pos'] = CORPUS[['term_str','pos']].value_counts().unstack().count(1)
VOCAB['cat_pos'] = CORPUS[['term_str','pos']].value_counts().to_frame('n').reset_index()\
    .groupby('term_str').pos.apply(lambda x: set(x))

from nltk.stem.porter import PorterStemmer
stemmer1 = PorterStemmer()
VOCAB['stem_porter'] = VOCAB.apply(lambda x: stemmer1.stem(x.name), 1)

from nltk.stem.snowball import SnowballStemmer
stemmer2 = SnowballStemmer("english")
VOCAB['stem_snowball'] = VOCAB.apply(lambda x: stemmer2.stem(x.name), 1)

from nltk.stem.lancaster import LancasterStemmer
stemmer3 = LancasterStemmer()
VOCAB['stem_lancaster'] = VOCAB.apply(lambda x: stemmer3.stem(x.name), 1)
```

```
VOCAB
```

Out[ ]:

term_str	n	n_chars	p	i	max_pos	max_pos_group	n_pos_group	cat_pos_group	n_pos	cat_pos	stem_porter	stem_snow
1	1	1	0.000001	19.497458	CD	CD	1	{CD}	1	{CD}	1	
1790	1	4	0.000001	19.497458	CD	CD	1	{CD}	1	{CD}	1790	1
1799	2	4	0.000003	18.497458	CD	CD	1	{CD}	1	{CD}	1799	1
1801more	1	8	0.000001	19.497458	CD	CD	1	{CD}	1	{CD}	1801more	1801r
1807	1	4	0.000001	19.497458	CD	CD	1	{CD}	1	{CD}	1807	1
...	...	...	...	...	...	...	...	...	...	...	...	
œdipus	2	6	0.000003	18.497458	NN	NN	1	{NN}	1	{NN}	œdipu	œd
μέγεθος	1	7	0.000001	19.497458	NNP	NN	1	{NN}	1	{NNP}	μέγεθος	μέγ
τι	1	2	0.000001	19.497458	NNP	NN	1	{NN}	1	{NNP}	τι	
ἀπέρωτος	1	8	0.000001	19.497458	JJ	JJ	1	{JJ}	1	{JJ}	ἀπέρωτος	ἀπέρ
ἔρως	1	4	0.000001	19.497458	NNP	NN	1	{NN}	1	{NNP}	ἔρως	ἐ

26337 rows × 13 columns

Questions

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

What regular expression did you use to chunk *Middlemarch* into chapters?

What is the title of the book that has the most tokens?

How many chapter level chunks are there in this novel?

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?

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

1.What regular expression did you use to chunk *Middlemarch* into chapters?

In [ ]:
```
LIB.query("book_id == 145")['chap_regex']
```

Out[ ]:
```
book_id
145    ^\s*CHAPTER\s+[IVXLCM]+\.s*$
Name: chap_regex, dtype: object
```

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

In [ ]:
```
LIB.sort_values('book_len', ascending=False).iloc[0,:]['raw_title']
```

Out[ ]:
```
'ELIOT GEORGE MIDDLEMARCH'
```

3.How many chapter level chunks are there in this novel? (Middlemarch)

In [ ]:
```
LIB.sort_values('book_len', ascending=False).iloc[0,:]['n_chaps']
```

Out[ ]:
```
86
```

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 [ ]:
```
print(f"There are {len(VOCAB.stem_porter.unique())} unique stems for Porter")
print(f"There are {len(VOCAB.stem_lancaster.unique())} unique stems for Lancester")
print(f"There are {len(VOCAB.stem_snowball.unique())} unique stems for Snowball")
```

```
There are 17540 unique stems for Porter
There are 14612 unique stems for Lancester
There are 17203 unique stems for Snowball
```

In [ ]:
```
VOCAB.reset_index().groupby(['stem_porter'])['n'].sum().mean()
```

Out[ ]:
```
42.197833523375145
```

In [ ]:
```
VOCAB.reset_index().groupby(['stem_snowball'])['n'].sum().mean()
```

Out[ ]:
```
43.024472475730974
```

In [ ]:
```
VOCAB.reset_index().groupby(['stem_lancaster'])['n'].sum().mean()
```

Out[ ]:
```
50.653572406241445
```

We can see from the results above that **Lancester** has the fewest unique stems and the highest average associated words per stem, so we consider it to be the most aggressive

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

In [ ]:
```
VOCAB['stem_lancaster'].value_counts().head()
```

Out[ ]:
```
cont    34
man     27
com     25
adv     21
pass    19
Name: stem_lancaster, dtype: int64
```

cont is the stem with the most (unique) associated terms