Elizabeth Coquillette - CS6120 Natural Language Processing NUID 002453121 Final Project Code (Part 1 of 2 - Processing Code)

In [2]:
```
# INSTALL THESE PACKAGES IF NOT YET DONE

#!pip install lexicalrichness
#!pip install textstat
#!pip install vaderSentiment
# nltk.download('universal_tagset')'


# LOAD DEPENDENCIES

from __future__ import annotations
import re, json, math, statistics, time, traceback
from dataclasses import dataclass, asdict
from pathlib import Path
from typing import List, Dict, Tuple
from lexicalrichness import LexicalRichness
import textstat
from datetime import datetime
from collections import Counter
from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer
from functools import lru_cache
from scipy.stats import entropy as shannon_entropy

import pandas as pd
import numpy as np
import nltk
from nltk.tokenize import sent_tokenize, word_tokenize
from nltk.stem import WordNetLemmatizer
from nltk.tag.mapping import map_tag
```

In [3]:
```
# PART ONE - CHAPTER SEGMENTATION

# Regex identification of what counts as a chapter label
#    Accepts only CHAPTER or Chapter followed by digits or roman numerals
#    Also accepts the special case of "Chapter the Last" for a particular bo
CHAPTER_LINE = re.compile(
    r"""(?mi)                       # multiline + case-insensitive
        ^\s*chapter                 # starts with 'chapter' (any case)
        \s+
        (?:[IVXLCDM]+|\d+|the\s+last)   # Roman numerals OR digits OR 'the l
        \b
        [^\n]*$                     # rest of heading line (optional)
    """,
    re.VERBOSE,
)

# Identify chapter class
@dataclass
class Chapter:
    index: int
```

```
    title: str
    text: str

def segment_into_chapters(text):
    """
    Split the text into chapters using CHAPTER_LINE.
    Returns a list of Chapter objects.
    If no headings are found, returns one 'FULL TEXT' chapter.
    """
    # Find matched headings
    spans = list(CHAPTER_LINE.finditer(text))

    chapters = []

    # Take each chapter and save it individually
    for i, m in enumerate(spans):
        start = m.start()

        # End at the start of next match, or end of text if this is the last
        end = spans[i + 1].start() if i + 1 < len(spans) else len(text)
        title = text[m.start(): m.end()].strip().replace("\n", " ")
        body  = text[m.end(): end].strip()
        if body:
            chapters.append(Chapter(len(chapters) + 1, title, body))
    return chapters or [Chapter(1, "FULL TEXT", text.strip())]

    # No matches → single chapter fallback
    if not spans:
        return [Chapter(1, "FULL TEXT", text.strip())]

    # Slice text between matched headings
    chapters: List[Chapter] = []

    # Take each chapter and save it individually
    for i, m in enumerate(spans):
        start = m.start()
        # End at the start of next match, or end of text if this is the last
        end = spans[i + 1].start() if i + 1 < len(spans) else len(text)
        title = text[m.start(): m.end()].strip().replace("\n", " ")
        body  = text[m.end(): end].strip()
        if body:
            chapters.append(Chapter(len(chapters) + 1, title, body))

    return chapters or [Chapter(1, "FULL TEXT", text.strip())]
```

In [4]:

```
# PART TWO – CHAPTER TEXT PROCESSING

# Mapping table for universal POS tags
_UNI2WN = {"NOUN": "n", "VERB": "v", "ADJ": "a", "ADV": "r"}

# Create lemmatizer & Vader
lemmatizer = WordNetLemmatizer()
_VADER = SentimentIntensityAnalyzer()


# All the features we are going to save
```

```python
@dataclass
class ChapterFeatures:
    # Core identifiers
    book_id: str
    chapter_index: int
    chapter_title: str

    # Basic structure stats
    n_sentences: int
    n_tokens: int                # all tokens (incl. punctuation/numerals)
    n_tokens_alpha: int          # alphabetic tokens we POS-tag/lemmatize

    # Basic word & sentence length measurements
    mean_sentence_len_tokens: float
    var_sentence_len: float
    mean_word_len: float
    var_word_len: float

    # POS distributions (counts)
    pos_counts_universal: Dict[str, int]
    pos_counts_penn: Dict[str, int]

    # Lexical richness
    ttr: float                   # type-token ratio over alpha tokens (lower
    mtld: float                  # MTLD (lexicalrichness) over alpha tokens

    # Readability
    flesch_reading_ease: float
    flesch_kincaid_grade: float

    # Lemma variety
    n_unique_lemmas: int

    # QC sample
    sample_sentences: List[str]

    # Entropy
    word_entropy_bits: float

    # Vader sentiment analysis
    vader_compound: float
    vader_pos: float
    vader_neu: float
    vader_neg: float

def _lemmatize_tokens(tokens_alpha, univ_tags):
    """Lemmatize alphabetic tokens using Universal POS to guide WordNet."""
    lemmas = []
    for tok, upos in zip(tokens_alpha, univ_tags):
        # Convert Universal POS tag to the WordNet format
        wn_pos = _UNI2WN.get(upos, "n")  # default to noun

        # Lemmatize token with the right tag
        lemmas.append(lemmatizer.lemmatize(tok, pos=wn_pos))
    return lemmas
```

```python
def _compute_lexical_richness(tokens_alpha):
    """ Returns (TTR, MTLD). TTR is unique/total over lowercased alphabetic
    # If no tokens, can't compute
    if not tokens_alpha:
        return (math.nan, math.nan)

    # Convert tokens to lowercase
    lower = [t.lower() for t in tokens_alpha]

    # Number of unique types
    types = len(set(lower))

    # Type-token ratio (unique words/total words)
    ttr = types / len(lower)

    # Initialize lexical richness with the text string
    lr = LexicalRichness(" ".join(lower))

    # Compute MTLD with the standard threshold (0.72)
    mtld = lr.mtld(threshold=0.72)

    return (ttr, mtld)

def _compute_readability(text):
    """Returns Flesch Reading Ease & Flesch-Kincaid Grade"""
    fre = float(textstat.flesch_reading_ease(text))
    fkg = float(textstat.flesch_kincaid_grade(text))
    return (fre, fkg)

def _word_entropy_bits(tokens_lower):
    """ Compute hannon entropy H(X) over lowercased word tokens """
    # Frequency of each unique token
    counts = np.fromiter(
        (c for c in Counter(tokens_lower).values()),
        dtype=float
    )
    total = counts.sum()
    if total == 0:
        return math.nan

    # Probabilities
    probs = counts / total

    # SciPy entropy with base=2 gives bits
    return float(shannon_entropy(probs, base=2))

def process_segment(chapter_or_chapters, book_id, keep_sentence_texts=True,
    """
    Do full processing e a single chapter or the entire book.

    Parameters -- chapter_or_chapters depends on whether is_full_book=True (
        (single chapter).
    Optional to store sample sentences using keep_sentence_texts = True.

    Returns chapterFeatures summary row
    """
```

```
    # If we are using full book, adjust settings accordingly
    if is_full_book:
        # Concatenate the *clean* chapter texts
        full_text = "\n\n".join(ch.text for ch in chapter_or_chapters if ch.
        text = full_text
        chapter_index = 0
        chapter_title = "__FULL_BOOK__"
        keep_samples = False  # usually omit samples for full-book
    # Otherwise adjust settings for chapters
    else:
        chapter = chapter_or_chapters
        text = chapter.text
        chapter_index = chapter.index
        chapter_title = chapter.title
        keep_samples = keep_sentence_texts

    # Sentence segmentation
    sents = sent_tokenize(text)

    # Word tokenization per sentence and flatten
    sent_tokens = [word_tokenize(s) for s in sents]
    tokens = [t for sent in sent_tokens for t in sent if t.strip() != ""]
    tokens_alpha = [t for t in tokens if t.isalpha()]  # focus on alphabetic

    # POS tagging (Penn), then map to Universal
    penn_pairs = nltk.pos_tag(tokens_alpha)                    # [('fox','
    penn_tags = [tag for _, tag in penn_pairs]                 # ['NN','VB
    univ_tags = [map_tag("en-ptb", "universal", t) for t in penn_tags]  # ['

    # Count penn tagset
    pos_counts_penn: Dict[str, int] = {}
    for tag in penn_tags:
        pos_counts_penn[tag] = pos_counts_penn.get(tag, 0) + 1

    # Count universal tagset
    pos_counts_universal: Dict[str, int] = {}
    for tag in univ_tags:
        pos_counts_universal[tag] = pos_counts_universal.get(tag, 0) + 1

    # Lemmas + lemma variety
    lemmas = _lemmatize_tokens(tokens_alpha, univ_tags)
    n_unique_lemmas = len(set(lemmas))

    # Lexical richness (TTR + MTLD)
    ttr, mtld = _compute_lexical_richness(tokens_alpha)

    # Readability (Flesch) on raw text
    fre, fkg = _compute_readability(text)

    # Basic stats built from the same tokenization
    sent_lengths = [len(st) for st in sent_tokens]

    # Whole-segment tokens
    word_lengths = [len(t) for t in tokens_alpha]  # character lengths of al
```

```
    # Basic counts
    n_sentences = len(sent_lengths)
    n_tokens = sum(sent_lengths)
    n_tokens_alpha = len(tokens_alpha)

    # Mean sentence length in tokens
    mean_sentence_len_tokens = float(np.mean(sent_lengths)) if n_sentences e
    sample = sents[:3] if keep_samples else []

    # Sentence-length variance (population)
    var_sentence_len = float(np.var(sent_lengths, ddof=0)) if n_sentences >=

    # Mean & variance of word length
    mean_word_len = float(np.mean(word_lengths)) if word_lengths else np.nar
    var_word_len  = float(np.var(word_lengths, ddof=0)) if len(word_lengths)

    # Entropy over lowercased alphabetic tokens
    lower_alpha = [t.lower() for t in tokens_alpha]
    word_ent = float(_word_entropy_bits(lower_alpha)) if lower_alpha else np

    # Sentiment (VADER) on raw text
    if '_VADER' in globals() and _VADER is not None and len(text.split()) >=
        vs = _VADER.polarity_scores(text)
        vader_compound = float(vs.get("compound", math.nan))
        vader_pos = float(vs.get("pos", math.nan))
        vader_neu = float(vs.get("neu", math.nan))
        vader_neg = float(vs.get("neg", math.nan))
    else:
        vader_compound = vader_pos = vader_neu = vader_neg = math.nan

    # Return the unified feature row
    return ChapterFeatures(
        book_id=book_id,
        chapter_index=chapter_index,
        chapter_title=chapter_title,
        n_sentences=n_sentences,
        n_tokens=n_tokens,
        n_tokens_alpha=n_tokens_alpha,
        mean_sentence_len_tokens=mean_sentence_len_tokens,
        pos_counts_universal=dict(sorted(pos_counts_universal.items())),
        pos_counts_penn=dict(sorted(pos_counts_penn.items())),
        ttr=float(round(ttr, 4)) if not math.isnan(ttr) else math.nan,
        mtld=float(round(mtld, 4)) if (mtld is not None and not math.isnan(m
        flesch_reading_ease=fre,
        flesch_kincaid_grade=fkg,
        n_unique_lemmas=n_unique_lemmas,
        sample_sentences=sample,
        var_sentence_len=var_sentence_len,
        mean_word_len=mean_word_len,
        var_word_len=var_word_len,
        vader_compound=vader_compound,
        vader_pos=vader_pos,
        vader_neu=vader_neu,
        vader_neg=vader_neg,
        word_entropy_bits=word_ent,
    )
```

In [5]:

```python
# PART THREE - ADDITIONAL HELPER FUNCTIONS

def expand_pos_counts_and_props(df):
    """
    Expand POS dictionary columns into wide numeric columns AND add per-chap
    This is useful data restructuring for later use in analysis.
    """
    # Sub-part 1 - Expand dicts to wide numeric columns

    # Universal POS counts
    if "pos_counts_universal" in df.columns:
        # Convert dict to columns; missing keys → NaN → fill with 0 → int
        pos_univ = (
            df["pos_counts_universal"]
            .apply(lambda d: pd.Series(d))
            .fillna(0)
            .astype(int)
            .add_prefix("pos_univ_")  # clear prefix to avoid name clashes
        )
        # Replace the dict column with expanded columns
        df = pd.concat([df.drop(columns=["pos_counts_universal"]), pos_univ]

    # Penn Treebank POS counts
    if "pos_counts_penn" in df.columns:
        pos_penn = (
            df["pos_counts_penn"]
            .apply(lambda d: pd.Series(d))
            .fillna(0)
            .astype(int)
            .add_prefix("pos_penn_")
        )
        df = pd.concat([df.drop(columns=["pos_counts_penn"]), pos_penn], axi

    # Sub-part 2 - Add proportion (relative frequency) cols

    # Denominator = number of alphabetic tokens tagged; replace 0 with NA to
    denom = df["n_tokens_alpha"].replace(0, pd.NA)

    # For Universal tags: create pos_univ_prop_<TAG> for each count column
    univ_count_cols = [
        c for c in df.columns
        if c.startswith("pos_univ_") and not c.startswith("pos_univ_prop_")
    ]
    for c in univ_count_cols:
        tag = c[len("pos_univ_"):]  # extract the tag name
        df[f"pos_univ_prop_{tag}"] = (df[c] / denom).astype(float)

    # For Penn tags: create pos_penn_prop_<TAG> for each count column
    penn_count_cols = [
        c for c in df.columns
        if c.startswith("pos_penn_") and not c.startswith("pos_penn_prop_")
    ]
    for c in penn_count_cols:
        tag = c[len("pos_penn_"):]
        df[f"pos_penn_prop_{tag}"] = (df[c] / denom).astype(float)
```

```
    return df

# Timestamp helper function for tracking performance
def _now():
    return datetime.now().strftime("%H:%M:%S")

# Helper function to save dataframe to csv
def save_output_csv(df, filename):
    filename = Path(filename)
    filename.parent.mkdir(parents=True, exist_ok=True)
    df.to_csv(filename, index=False)
```

In [6]:

```
# PART FOUR — MAIN PROCESSING FUNCTIONS

def process_book_to_df(path, book_id, full_book = True, keep_sentence_texts
    """
    Read the file at `path` and fully process chapters and optionally FULL_B
    Returns wide dataframe ready for downstream expansion (POS props) and sa
    """

    # Read & clean
    p = Path(path)
    print(f"    [{_now()}] read_text -> {p}")
    text = p.read_text(encoding="utf-8", errors="ignore")

    # Segment into chapters
    print(f"    [{_now()}] segment_chapters")
    chapters = segment_into_chapters(text)
    print(f"    [{_now()}] chapters_found={len(chapters)}")

    rows = []

    # If processing is being run on full book, do that
    if full_book:
        print(f"    [{_now()}] full_book start")
        row0 = process_segment(chapters, book_id, is_full_book=True, keep_se
        rows.append(row0)
        print(f"    [{_now()}] full_book done")
    else:
        print("Full book option not enabled")

    # Run processing on chapters
    print(f"    [{_now()}] chapter_loop start (n={len(chapters)})")
    import time
    for i, ch in enumerate(chapters, 1):
        # Timing settings to track progress
        t0 = time.time()
        if i == 1 or i % 5 == 0:
            print(f"    [{_now()}]   chapter {i}/{len(chapters)} — start")

        # Actually process the segment
        row = process_segment(ch, book_id, is_full_book=False, keep_sentence
        rows.append(row)

        # Timing settings to track progress
```

```
        dt = time.time() - t0
        if i == 1 or i % 5 == 0 or dt > 2.0:
            print(f"    [{_now()}]   chapter {i} done in {dt:.2f}s")

    # Store in dataframe
    df = pd.DataFrame(rows)

    # Accounts for some older code with different variable names
    if "var_sentence_len_tokens" in df.columns and "var_sentence_len" not in
        df["var_sentence_len"] = df["var_sentence_len_tokens"]
    if "mean_sentence_len_tokens" in df.columns and "mean_sentence_len" not
        df["mean_sentence_len"] = df["mean_sentence_len_tokens"]

    # Expand POS dicts + add proportions (skips if dict columns are absent)
    if "pos_counts_universal" in df.columns or "pos_counts_penn" in df.colum
        print("expanding pos_counts")
        df = expand_pos_counts_and_props(df)

    # Column ordering
    front = [
        "book_id",                      # ID or name of the book
        "chapter_index",                # Chapter number (0 = full book summar
        "chapter_title",                # Title/heading of the chapter
        "n_sentences",                  # Number of sentences in chapter
        "n_tokens",                     # Total tokens (including punctuation,
        "n_tokens_alpha",               # Tokens containing only alphabetic ch
        "ttr",                          # Type-Token Ratio (lexical diversity)
        "mtld",                         # Measure of Textual Lexical Diversity
        "flesch_reading_ease",          # Flesch Reading Ease score
        "flesch_kincaid_grade",         # Flesch-Kincaid Grade Level
        "n_unique_lemmas",              # Unique lemmatized word forms
        "mean_sentence_len",            # Mean sentence length
        "var_sentence_len",             # Variance sentence length
        "mean_word_len",                # Mean word length
        "var_word_len",                 # Variance word length
        "word_entropy_bits",            # Word entropy
        "vader_compound",               # Vader sentiment analysis
        "vader_pos",
        "vader_neu",
        "vader_neg",
        "pos_counts_universal",
        "pos_counts_penn"
    ]

    # Reorder columns
    existing_front = [c for c in front if c in df.columns]
    other_cols = [c for c in df.columns if c not in existing_front]

    # Keep sample sentences last if present
    if "sample_sentences" in other_cols:
        other_cols = [c for c in other_cols if c != "sample_sentences"] + ["

    # Reindex and sort
    df = df.reindex(columns=existing_front + other_cols)
    df = df.sort_values(["chapter_index"]).reset_index(drop=True)
    return df
```

```
def batch_process_books(books, full_book=True):
    """
    Process multiple novels and return:
      - df_all: concatenated chapter-level tables plus a FULL_BOOK row
      - df_summary: subset with only FULL_BOOK rows
    """
    frames = []

    # Loop over each book specification
    for spec in books:
        p = Path(spec["path"])
        book_id = spec["book_id"]
        author = spec.get("author", None)

        print(f"[{_now()}] → START {book_id} ({p})")

        try:
            # Process the book with all features enabled except keeping sent
            df = process_book_to_df(p, book_id=book_id, full_book=full_book,

            # Add author if given
            if author is not None:
                df.insert(0, "author", author)

            # Add this book to the frames
            frames.append(df)
            print(f"✓ Processed {book_id} ({p}) with {len(df)} rows")

        # Print exception if one of the books doesn't work and carry on with
        except Exception as e:
            print(f"✗ Failed on {book_id} ({p}): {e}")

    if not frames:
        raise ValueError("No books processed successfully")

    # Combine all into one
    df_all = pd.concat(frames, ignore_index=True, sort=False)

    # Extract just the FULL_BOOK rows for summary
    df_summary = df_all[df_all["chapter_index"] == 0].copy()

    return df_all, df_summary
```

In [20]:

```
# PART FIVE — RUNNING THE FUNCTIONS

# List of books to process
# Include author = Dickens field to allow for adding new authors later
books = [
    {"path": "texts/TheOldCuriosityShop.txt", "book_id": "TheOldCuriositySho
    {"path": "texts/PickwickPapers.txt", "book_id": "PickwickPapers", "autho
    {"path": "texts/OurMutualFriend.txt", "book_id": "OurMutualFriend", "aut
    {"path": "texts/OliverTwist.txt", "book_id": "OliverTwist", "author": "D
    {"path": "texts/NicholasNickelby.txt", "book_id": "NicholasNickelby", "a
    {"path": "texts/MartinChuzzlewit.txt", "book_id": "MartinChuzzlewit", "a
```

```
    {"path": "texts/LittleDorrit.txt", "book_id": "LittleDorrit", "author":
    {"path": "texts/HardTimes.txt", "book_id": "HardTimes", "author": "Dicke
    {"path": "texts/GreatExpectations.txt", "book_id": "GreatExpectations",
    {"path": "texts/DombeyAndSon.txt", "book_id": "DombeyAndSon", "author":
    {"path": "texts/DavidCopperfield.txt", "book_id": "DavidCopperfield", "a
    {"path": "texts/BleakHouse.txt", "book_id": "BleakHouse", "author": "Dic
    {"path": "texts/BarnabyRudge.txt", "book_id": "BarnabyRudge", "author":
    {"path": "texts/ATaleOfTwoCities.txt", "book_id": "ATaleOfTwoCities", "a
]

# Option for shortened version of books for testing purposes
#books = [{"path": "texts/TheOldCuriosityShop.txt", "book_id": "TheOldCurios
#print("SHORTENED VERSION ENABLED")

# Process
df_all, df_summary = batch_process_books(books, full_book=False)

# Save combined outputs
save_output_csv(df_all, filename="data/processed/ALL_books_chapters_plus_ful
save_output_csv(df_summary, filename="data/processed/ALL_books_fullbook_summ
```

```
SHORTENED VERSION ENABLED
[20:07:51] → START TheOldCuriosityShop (texts/TheOldCuriosityShop.txt)
    [20:07:51] read_text -> texts/TheOldCuriosityShop.txt
    [20:07:51] segment_chapters
    [20:07:51] chapters_found=73
Full book option not enabled
    [20:07:51] chapter_loop start (n=73)
    [20:07:51]   chapter 1/73 — start
    [20:07:51]   chapter 1 done in 0.35s
    [20:07:52]   chapter 5/73 — start
    [20:07:52]   chapter 5 done in 0.09s
    [20:07:52]   chapter 10/73 — start
    [20:07:52]   chapter 10 done in 0.08s
    [20:07:53]   chapter 15/73 — start
    [20:07:53]   chapter 15 done in 0.16s
    [20:07:54]   chapter 20/73 — start
    [20:07:54]   chapter 20 done in 0.06s
    [20:07:54]   chapter 25/73 — start
    [20:07:54]   chapter 25 done in 0.13s
    [20:07:55]   chapter 30/73 — start
    [20:07:55]   chapter 30 done in 0.07s
    [20:07:55]   chapter 35/73 — start
    [20:07:56]   chapter 35 done in 0.19s
    [20:07:56]   chapter 40/73 — start
    [20:07:56]   chapter 40 done in 0.13s
    [20:07:57]   chapter 45/73 — start
    [20:07:57]   chapter 45 done in 0.11s
    [20:07:57]   chapter 50/73 — start
    [20:07:57]   chapter 50 done in 0.16s
    [20:07:58]   chapter 55/73 — start
    [20:07:58]   chapter 55 done in 0.08s
    [20:07:59]   chapter 60/73 — start
    [20:07:59]   chapter 60 done in 0.19s
    [20:07:59]   chapter 65/73 — start
    [20:07:59]   chapter 65 done in 0.07s
    [20:08:00]   chapter 70/73 — start
    [20:08:00]   chapter 70 done in 0.08s
expanding pos_counts
✓ Processed TheOldCuriosityShop (texts/TheOldCuriosityShop.txt) with 73 rows
```

In [ ]: