STAT 243 Problem Set 3

Treves Li

2024-09-30

Collaboration Statement

I did not collaborate with anyone.

Question 1

> Let's investigate the structure of the `statsmodels` package to get some experience with the structure of a large Python package and with how `import` and the `__init__.py` file(s) are used. You'll need to go into the `statsmodels` source code (see Unit 5). Also note that the following cases may involve functions, classes, and class methods. Be sure to be clear to say which of those you are talking about and if it's a class, describe any inheritance structure.

Question 1a

> For this subpart only, consider doing `import statsmodels`. What is in the `statsmodels` namespace that is created? Where (what module file) is the version number for `statsmodels` stored in? What is the absolute path to the package on the machine you are working on?

To access the `statsmodels` namespace, I use `dir()`:

```python
import statsmodels
dir(statsmodels)
```

```
['__all__',
 '__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__path__',
 '__spec__',
 '__version__',
 '__version_info__',
 '__version_tuple__',
 '_version',
 'compat',
 'debug_warnings',
 'monkey_patch_cat_dtype',
 'test',
 'tools']
```

It's interesting that the submodules normally associated with `statsmodels` (e.g., `statsmodels.api` or `statsmodels.tsa`) are not imported at the same time. Based on reading the `__init__.py`, it seems like the import is only loading the minimal high-level `statsmodels` module with some metadata-like dunders, perhaps in an effort to be space efficient. I can imagine that the entire package might use a lot of memory.

The version number for `statsmodels` is stored in the `_version.py` file. I first tried looking for it in `statsmodels`'s `__init__.py`, but opening that file revealed that it imported the "version" information from another module/file based on this line:

`from statsmodels._version import __version__, __version_tuple__`.

We can verify by running the code chunk below:

```
print(statsmodels._version.__file__)
print(statsmodels.__version__)
```

```
/home/treves/.local/lib/python3.10/site-packages/statsmodels/_version.py
0.14.3
```

To find the absolute path to the `statsmodels` package's source code, I utilise the `__file__` attribute:

```python
print(statsmodels.__file__)
```

```
/home/treves/.local/lib/python3.10/site-packages/statsmodels/__init__.py
```

Question 1b

The remaining subparts all relate to using the standard `import statsmodels.api as sm` invocation. First, describe briefly what happens when this is run (what files are accessed). Then, describe what kind of object `MICE` is, how it is imported and where it is found. Do the same for `GLM`.

When I run the invocation, I can see what file is being run:

```python
import statsmodels.api as sm
print(sm.__file__)
```

```
/home/treves/.local/lib/python3.10/site-packages/statsmodels/api.py
```

By looking closely at `api.py`, I can determine that the code is accessing each of the submodules to set up the namespace that it needs to operate. Specifically, it is importing the necessary functions, classes, submodules, and variables. All of this information on what is imported is made available to the user if they were to run `dir(sm)`.

MICE

To find out what kind of object `MICE` is, how it is imported, and where it's found I run the following code:

```python
print(type(sm.MICE))
```

```
<class 'type'>
```

The result shows that `MICE` is a class object.

By reading the `api.py` file, I can determine that it is directly imported using the following statement:

```python
from .imputation.mice import MICE
```

To find where the `MICE` module is, I use the information from the `import` statement while also performing `grep` on the `statsmodels` directory. I also run this code:

```
print(sm.MICE.__module__)
```

```
statsmodels.imputation.mice
```

From all these information, I can find that `MICE`'s inheritance structure and location in:

`/home/treves/.local/lib/python3.10/site-packages/statsmodels/imputation/mice.py`

GLM

I can do the same for the `GLM` module:

```
print(type(sm.GLM))
```

```
<class 'type'>
```

Again, `GLM` is a class object, which is directly imported through:

```
from .genmod.api import GLM
```

If I open `/genmod/api.py`, I find that it is actually imported through the `generalized_linear_model` submodule:

```
from .generalized_linear_model import GLM
```

which allows me to find its inheritance structure and true location here:

`~/.local/lib/python3.10/site-packages/statsmodels/genmod/generalized_linear_model.py`

I can verify the location by running this code chunk:

```
print(sm.GLM.__module__)
```

```
statsmodels.genmod.generalized_linear_model
```

Question 1c

> Consider `sm.gam`. What is in the namespace? Describe how the importing works and in what modules the objects in the namespace are defined.

To determine the namespace, I can use `dir()` on `sm.gam`:

```
dir(sm.gam)
```

```
['BSplines',
 'CyclicCubicSplines',
 'GLMGam',
 'MultivariateGAMCVPath',
 '__all__',
 '__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__']
```

To determine how it's imported, I can trace what file is actually being run.

```
print(sm.gam.__file__)
```

```
/home/treves/.local/lib/python3.10/site-packages/statsmodels/gam/api.py
```

When I open `/gam/api.py`, I find that the importing works by directly accessing the objects in their respective submodules, in such a way that the objects (which are likely classes) now become directly available in the `gam` submodule's local namespace. The import statements are provided below as an example.

```
from .generalized_additive_model import GLMGam
from .gam_cross_validation.gam_cross_validation import MultivariateGAMCVPath
from .smooth_basis import BSplines, CyclicCubicSplines
```

The submodules in which the objects are defined can be determined through the following code:

```
for i in sm.gam.__all__:
  object = getattr(sm.gam, i)
  print(f"'{i}' is in {object.__module__}")
```

```
'BSplines' is in statsmodels.gam.smooth_basis
'CyclicCubicSplines' is in statsmodels.gam.smooth_basis
'GLMGam' is in statsmodels.gam.generalized_additive_model
'MultivariateGAMCVPath' is in statsmodels.gam.gam_cross_validation.gam_cross_validation
```

Question 1d

Consider `sm.distributions.monotone_fn_inverter`. What is it, how it is imported and what file it is defined in?

To determine what type of object it is, I run:

```
print(type(sm.distributions.monotone_fn_inverter))
```

```
<class 'function'>
```

The output shows that it's a **function** inside the `sm.distributions` submodule. Since the object is a function, and not something like a class, then there is no explicit inheritance structure.

To determine how it is imported, I can `grep` the function name in the `statsmodels` directory, and find that it is called in the following file:

`~/.local/lib/python3.10/site-packages/statsmodels/distributions/__init__.py:`

Specifically, the code below from `__init__.py` shows that it is imported directly from the `empirical_distribution` submodule (which is itself from the `distributions` module) and into the `distributions` namespace.

```
from .empirical_distribution import (
  ECDF, ECDFDiscrete, monotone_fn_inverter, StepFunction
  )
```

We confirm that the function is in the `distributions` namespace as a sanity check:

```
import statsmodels.distributions
'monotone_fn_inverter' in dir(sm.distributions)
```

```
True
```

Based on the information above, I can determine that the function is likely defined in some file called `empirical_distribution.py`. I can confirm this by running:

```python
print(sm.distributions.empirical_distribution.__file__[-51:])
```

```
statsmodels/distributions/empirical_distribution.py
```

When I open the `empirical_distribution.py` file above, I indeed find that the function is defined there, specifically in Line 218:

```python
filename = sm.distributions.empirical_distribution.__file__

with open(filename, "r") as f:
  for line_num, line in enumerate(f, start=1):
    if "monotone_fn_inverter" in line:
      print(f"Line {line_num}: {line.strip()}")
```

```
Line 218: def monotone_fn_inverter(fn, x, vectorized=True, **keywords):
```

Question 2

The website Commission on Presidential Debates has the text from recent debates between the candidates for President of the United States. (As a bit of background for those of you not familiar with the US political system, there are usually three debates between the Republican and Democratic candidates at which they are asked questions so that US voters can determine which candidate they would like to vote for.) Your task is to process the information and produce data on the debates. Note that while I present the problem below as subparts (a)-(d), your solution does not need to be divided into subparts in the same way, but you do need to make clear in your solution where and how you are doing what. For the purposes of this problem, please work on the the debates I've selected (see code below) for the years 2000, 2004, 2008, 2012, 2016, and 2020. (I've tried to select debates that cover domestic policy in whole or in part to control one source of variation, namely the topic of the debate.) I'll call each individual response by a candidate to a question a "chunk". A chunk might just be a few words or might be multiple paragraphs.

The goal of this problem is two-fold: first to give you practice with regular expressions and string processing and the second to have you thinking about writing well-structured, readable code (similar to question 4 of PS1). You can choose to use either a functional programming approach or an object-oriented approach. I strongly recommend that you use the approach that you are **less** familiar with so as to gain more experience. Please think about writing short, modular functions or methods. Explore the use of `map`, list comprehension or other techniques to avoid having a lot of nested for loops. Think carefully about how to structure your objects to store the spoken chunks so that the structure works well with your functions/methods. Note that for this problem, for the sake of time, you do not need extensive docstrings, but it should still be clear what each function does. In parts (a)-(c), add simple sanity checks that you are getting reasonable results.

Given that in earlier problem sets, you already worked on downloading and processing HTML, I'm giving you the code (in the file `ps/ps3prob3.py` in the class repository) to download the HTML and do some initial processing, so you can dive right into processing the actual debate text.

I have more experience with using functional programming, so I attempted this question using object-oriented programming.

First, I execute the Python file to download and process the HTML.

```python
ps3prob3 = r"~/fall-2024/ps/ps3prob3.py"

# Expand to full path
import os.path

ps3prob3 = os.path.expanduser(ps3prob3)

with open(ps3prob3) as f:
  script = f.read()

exec(script)

# The code chunk is amended so as not to generate output
# This is to make Quarto formatting neater for this assignment
```

Each debate transcript is thus saved as an element in the list `debates_body`.

I also use the opportunity to extract the list of `candidates` from the `ps3prob3.py` file, as well as the list of `moderators`. I also created a list of speakers for each debate year `speakers_by_year`, which will be utilised in later parts of this assignment.

```python
# Get the names of debaters for years of interest
candidates = [person for entry in candidates for person in entry.values()]

# Retrieve only unique names, for both candidates and moderators
candidates_unique = list(set(candidates))
moderators_unique = list(set(moderators))

# Combine to make a list of speakers
speakers_unique = candidates_unique + moderators_unique

# Create a dict to hold speakers by year
speakers_by_year = {year: None
                    for year in [2000, 2004, 2008, 2012, 2016, 2020]
                    }

for i, year in enumerate(speakers_by_year.keys()):

  # Initialize the list for the year if it's None
  if speakers_by_year[year] is None:
    speakers_by_year[year] = []
  speakers_by_year[year].append(moderators[i])
  speakers_by_year[year].append(candidates[i * 2])
  speakers_by_year[year].append(candidates[i * 2 + 1])

print("\nCandidate names:")
print(candidates_unique)
print("\nModerator names:")
print(moderators_unique)
#print(speakers_by_year)
```

```
Candidate names:
['CLINTON', 'GORE', 'ROMNEY', 'BIDEN', 'TRUMP', 'OBAMA', 'KERRY', 'MCCAIN', 'BUSH']

Moderator names:
['WALLACE', 'SCHIEFFER', 'MODERATOR', 'HOLT', 'LEHRER']
```

Question 2a

> Convert the text so that for each debate, the spoken text is split up into individual chunks of text spoken by each speaker (including the moderator). If there are two chunks in a row spoken by a candidate, combine them into a single chunk. Make sure that any formatting and non-spoken text (e.g., the tags for 'Laughter' and 'Applause') is stripped out. Report the number of chunks per speaker.

I first create a class called `Debate`, and define its methods.

In `find_non_spoken`, I find and strip out non-spoken text, which is normally indicated by single words in parentheses or square brackets.

I also realised there was an issue with the 2020 debate transcript. Somehow, the dialogue was processed as "[SPEAKER]:[dialogue]", i.e., with no space between the colon and dialogue, whereas all the other transcripts were in the format "[SPEAKER]: [dialogue]". So I had to do some additional processing on that particular transcript.

In `strip_non_spoken`, I strip the transcript of the non-spoken texts. I wrote some additional functions (`count_non_spoken` and `compare_pre_post_strip`) for sanity checks.

```python
import re

class Debate:
  def __init__(self, debates_body):
    self.debates_body = debates_body

  def find_non_spoken(self):
    """
    Searches and extracts non-spoken text (e.g., "Laughter" or "Applause") in
    parentheses or square brackets from debates.

    Returns:
        list: A list of non-spoken words found in the debates.
    """

    # Regex to find all single words that appear within parentheses
    # and square brackets
    non_spoken_pattern = r"[\(\[]\w+[\)\]]"
    non_spoken_matches = [
        re.findall(non_spoken_pattern, debate) for debate in self.debates_body
    ]

    # Flatten the list of lists using list comprehension
    non_spoken_matches = [
        match for debate in non_spoken_matches for match in debate
    ]

    return non_spoken_matches

  def strip_non_spoken(self):
    """
    Strips non-spoken text from debates using regex.

    Returns:
        list: A list of debates with non-spoken text removed.
    """

    # Create new regex pattern based on non_spoken_matches
    non_spoken_list_pattern = "|".join(map(re.escape, self.find_non_spoken()))

    # Strip out all non-spoken text using regex
    # Strip out leading and trailing whitespaces
    debates_stripped = [
        re.sub(non_spoken_list_pattern, "", debate).strip()
        for debate in self.debates_body
    ]

    return debates_stripped

  def fix_colons(self, debate):
    """
    Everytime "SPEAKER:" is encountered, replace with "SPEAKER: ",
    i.e., with a space after colon.

    Returns:
        str: Debate transcript with spaces after colons.
    """
    colon_pattern = r"([A-Z]+:)"
    colon_replacement = r"\1 "
    updated_transcript = re.sub(colon_pattern, colon_replacement, debate)
    return updated_transcript
  def count_non_spoken(self, non_spoken_word):
    """
    Count occurrences of a specified "non-spoken word" in a debate.

    Returns:
        list: Counts of occurrences of the word in each debate.
    """
    counts = [
        debate.lower().count(non_spoken_word) for debate in self.debates_body
    ]
    return counts

  def compare_pre_post_strip(self, non_spoken_word):
    """
    Compare the counts of a specified non-spoken word before and after
    stripping "non-spoken" text.

    Returns:
        None: Prints counts before and after stripping.
    """
    counts_pre_strip = self.count_non_spoken(non_spoken_word)
    stripped_debates = Debate(self.strip_non_spoken())
    counts_post_strip = stripped_debates.count_non_spoken(non_spoken_word)
    return print(f"Counts of '{non_spoken_word}' for each year: \n\
      Pre-strip:\t {counts_pre_strip} \n\
      Post-strip:\t {counts_post_strip}\n")
```

I test the class and its methods below, while also running some sanity checks:

```
# Instantiate `Debate` class
debates = Debate(debates_body)

# Strip debates of non-spoken words
debates_stripped = debates.strip_non_spoken()

# Do some special processing on the 2020 debate transcript
# Since the formatting is different from the other debates
debates_stripped[5] = debates.fix_colons(debates_stripped[5])

# Check that non-spoken words are properly stripped
debates.compare_pre_post_strip("applause")
debates.compare_pre_post_strip("laughter")
debates.compare_pre_post_strip("crosstalk")
```

```
Counts of 'applause' for each year:
      Pre-strip:     [2, 2, 1, 2, 12, 0]
      Post-strip:    [0, 0, 0, 1, 0, 0]

Counts of 'laughter' for each year:
      Pre-strip:     [0, 6, 0, 4, 5, 0]
      Post-strip:    [0, 0, 0, 0, 0, 0]

Counts of 'crosstalk' for each year:
      Pre-strip:     [0, 0, 1, 26, 9, 41]
      Post-strip:    [0, 0, 0, 0, 0, 0]
```

Note that the single occurrence of “applause” from the post-stripped transcript was from the moderator literally saying the word “applause”.

To retrieve the chunks, I create a new class called `Chunk` that takes a string/transcript. In `get_debate_year`, I extract the year corresponding to each debate. I wrote `strip_preamble` to get rid of the preamble formatting in each transcript, which would otherwise interfere when counting the chunks of each speaker.

In `get_chunks`, I subdivide each transcript into chunks based on the idea that each chunk will be marked by a speaker (candidate or moderator; stylised in all caps) followed by a colon, using regex. I needed to adjust the regex iteratively, since each transcript had its own transcription idiosyncracies.

In `count_chunks`, I count the number of occurrences that each speaker has their own chunk in a debate, ensuring that only speakers that appear in the `speakers` list are included in the resultant dictionary.

`check_chunk_repeats` is a sanity check function that ensures that the occurrence of two chunks spoken in a row by the same candidate or speaker is zero. If not, then I had to go and revise my code for chunking the transcript.

```python
import spacy
import pandas as pd
import matplotlib.pyplot as plt

# Load spacy nlp model
nlp = spacy.load('en_core_web_sm')
```

```python
class Chunk:
  def __init__(self, transcript):
    self.transcript = transcript

  def get_debate_year(self):
    """
    Gets the debate year.

    Returns:
        int: The year of the debate
    """
    year_pattern = r"^.+?(\d{4})"
    year_matches = re.findall(year_pattern, self.transcript)
    matches_list = [int(match) for match in year_matches]

    return matches_list[0]

  def strip_preamble(self):
    """
    Strips the transcript preamble up to the first time a name in `moderator`
    is encountered.

    Returns:
        str: Transcript with preamble removed.
    """
    preamble_pattern = (
      r"^(.*?)(?:"
      + "|".join(re.escape(moderator) for moderator in moderators)
      + r"):\s*(.*)"
    )
    preamble_match = re.search(preamble_pattern, self.transcript)
    self.transcript = self.transcript[preamble_match.end(1) :].strip()
    return self.transcript

  def get_chunks(self, year):
    """
    Isolates and consolidates chunks of dialogue from the transcript
    using regex.

    Returns:
        list: A list of lists, where inner list contains the speaker's
              identifier and their consolidated speech chunk.
    """
    # Only create chunks for speakers from a given debate year
    valid_speakers = speakers_by_year[year]

    # Use regex to find all chunks of dialogue
    speakers_pattern = '|'.join(valid_speakers)
    # chunk_pattern = r"([A-Z]+): (?<=[A-Z]: )(.*?)(?=[A-Z]*:|$)"
    chunk_pattern = rf"({speakers_pattern}): (.*?)(?=\s*({speakers_pattern}):|$)"
    chunk_matches = re.findall(chunk_pattern, self.transcript)
    matches_list = [[speaker, dialogue]
                    for speaker, dialogue, _ in chunk_matches]

    # Consolidate chunks if spoken in a row by the same speaker
    i = 0
    while i < len(matches_list) - 1:
      # Check if the speaker names (the first element) are the same
      if matches_list[i][0] == matches_list[i + 1][0]:
        matches_list[i][1] += " " + matches_list[i + 1][1]
        del matches_list[i + 1]
      else:
        i += 1

    return matches_list

  def count_chunks(self, chunks, speakers):
    """
    Counts the number of chunks per speaker for each debate.
    Only speakers in the `speakers` list are considered.
    Also drop keys where value is less than 5.

    Returns:
        dict: A dict where the keys are speaker names, and the values
              are the counts of chunks per speaker.
    """
    speaker_count = {}

    for chunk in chunks:
      speaker = chunk[0]  # The first element is the speaker

      # Ensure speaker is only counted if they are in `speakers` list
      if speaker in speaker_count and speaker in speakers:
        # Increment counter for a particular speaker
        speaker_count[speaker] += 1
      elif speaker in speakers:
        # Initialise value for first time speaker is encoutered
        speaker_count[speaker] = 1

    """# Remove items in the dict with values less than 5,
    # since these likely aren't real chunks
    speaker_count = {
      key: value for key, value in speaker_count.items() if value >= 5
    }"""

    # print(f"Speaker counts: {speaker_count}")

    return speaker_count

  def check_chunk_repeats(self, debate_chunked):
    """
    As a sanity check, counts the occurrences of the speaker in
    consecutive chunks being the same.

    Returns:
        int: Count where speakers in consecutive chunks is the same.
    """
    # See if chunk and chunk+1 speakers are the same
    repeat_counts = 0
    for i in range(len(debate_chunked) - 1):
      if debate_chunked[i][0] == debate_chunked[i + 1][0]:
        # Print out the offending lines
        print(f"{debate_chunked[i][0]}: {debate_chunked[i][1]}")
        print(f"{debate_chunked[i+1][0]}: {debate_chunked[i+1][1]}")
        repeat_counts += 1
    return repeat_counts

  def extract_words(self, debate_chunked_year):
    """
    Extracts words spoken by each speaker for each debate year.
    using spaCy to tokenize chunks.

    Returns:
        dict: A dict with debate years as keys each speaker's
              extracted words as values.
    """
    # Hold speakers and their words for each debate year
    debate_year_speakers = {}

    for year, chunks in debate_chunked_year.items():
      speakers = []

      # Use a set to track seen speakers
      seen_speakers = set()

      for chunk in chunks:
        speaker_name = chunk[0]  # Get the speaker name
        dialogue = chunk[1]  # Get the associated dialogue

        # Run spacy nlp on the dialogue
        doc = nlp(dialogue)
        words = [
            token.text for token in doc
            if not token.is_punct
            and not token.is_space
        ]

        # If speaker has not been seen before
        if speaker_name not in seen_speakers:
            speakers.append(
                [speaker_name, words]
            )  # Append name and words as inner list
            seen_speakers.add(speaker_name)  # Mark this speaker as seen

        # If speaker has been seen already, simply append their words
        else:
          for speaker in speakers:
            if speaker[0] == speaker_name:
              speaker[1].extend(words)
              break

      # Save the speakers list for the current debate year
      debate_year_speakers[year] = speakers

    return debate_year_speakers

  def get_avg_word_length(self, extracted_words):
    """
```

```
    Calculates the average word length by each speaker for each debate year.

    Returns:
        dict: A dictionary  debate years as keys and
              each speaker name and their average word length.
    """
    # Hold word length speakers and their words for each debate year
    avg_words = {}

    for year, extr_words in extracted_words.items():

      # Create a new object (list) to store avg word length
      avg_word_length = []

      for speaker_words in extr_words:
        speaker = speaker_words[0]
        words = speaker_words[1]

        num_words = len(words)
        num_chars = sum(len(word) for word in words)
        avg_length = round(num_chars / num_words, 2)

        avg_word_length.append([speaker, avg_length])

      avg_words[year] = avg_word_length

    return avg_words

  def compare_word_counts(self, debates_stripped, extracted_words):
    """
    Compares the word counts of un-chunked transcripts with
    extracted word counts for each debate year.

    Prints:
        The raw word count, extracted word count, and
        differential for each debate year.
    """
    # Create a list to store word count of un-chunked transcipts
    raw_word_count = []

    for debate in debates_stripped:
      # Instantiate `Chunk` class
      raw_debate = Chunk(debate).strip_preamble()

      # Use regex to count number of words in the un-chunked transcript
      raw_words = re.findall(r'\b\w+\b', raw_debate)

      # Count words as part of dialogue
      count = sum(1 for word in raw_words if not re.match(r'^[A-Z]+:$', word))
      raw_word_count.append(count)


    # Create list to store extracted word count
    extracted_word_count = []

    # Loop through each debate year
    for years, words in extracted_words.items():

      # Sum total amount of words from each debate year
      count = sum(len(words[speakers][1]) for speakers in range(len(words)))

      # Store tally in the `extracted_word_count` list
      extracted_word_count.append(count)


    # For each debate year, compare the raw and extracted word counts
    for year, i in zip(extracted_words.keys(), range(len(raw_word_count))):
      difference = (raw_word_count[i] - extracted_word_count[i]) \
                   / raw_word_count[i] * 100
      print(f"{year} Debate")
      print(f"Raw word count: \t {raw_word_count[i]}")
      print(f"Extr. word count: \t {extracted_word_count[i]}")
      print(f"Difference: \t\t {difference:.2f}%\n")
```

Again, I test the class and run some sanity checks:

```
# Initialise dict to store debate chunks by year
debate_chunked_year = {}

# Assign each debate to a separate instance of Chunk
for transcript in debates_stripped:
  # Instantiate Chunk class for each transcript
  debate = Chunk(transcript)
  # Extract debate year
  year = debate.get_debate_year()

  # Remove preamble formatting
  debate.strip_preamble()

  # Chunk the debates, and store the chunks in a dict by year
  debate_chunked = debate.get_chunks(year)
  debate_chunked_year[year] = debate_chunked

  # Count the number of chunks per speaker for each year
  chunk_counts = debate.count_chunks(debate_chunked, speakers_unique)
  print(f"Chunk counts for {year}: {chunk_counts}")

# Sanity checks
print("\nCheck that consecutive chunks from the same speaker are combined.")
print("Count of \"chunk repeats\" per year should be 0.")
for year, chunks in debate_chunked_year.items():
  num_repeats = debate.check_chunk_repeats(chunks)
  print(f"Number of chunk repeats for {year}: {num_repeats}")
```

```
Chunk counts for 2000: {'MODERATOR': 60, 'GORE': 49, 'BUSH': 56}
Chunk counts for 2004: {'SCHIEFFER': 57, 'KERRY': 31, 'BUSH': 29}
Chunk counts for 2008: {'SCHIEFFER': 55, 'MCCAIN': 60, 'OBAMA': 45}
Chunk counts for 2012: {'LEHRER': 76, 'OBAMA': 42, 'ROMNEY': 54}
Chunk counts for 2016: {'HOLT': 97, 'CLINTON': 87, 'TRUMP': 123}
Chunk counts for 2020: {'WALLACE': 245, 'BIDEN': 266, 'TRUMP': 337}

Check that consecutive chunks from the same speaker are combined.
Count of "chunk repeats" per year should be 0.
Number of chunk repeats for 2000: 0
Number of chunk repeats for 2004: 0
Number of chunk repeats for 2008: 0
Number of chunk repeats for 2012: 0
Number of chunk repeats for 2016: 0
Number of chunk repeats for 2020: 0
```

Question 2b

Extract the individual words. To do the extraction I suggest you use existing Python natural language processing packages that can "tokenize" text, but you're also welcome to use regular expressions, particularly if you want to practice them more. Doing this with 100% accuracy will be hard (particularly for spoken text). Try to handle various issues, but don't try to be perfect.

I use the spaCy package to extract the words. I included it as a function in the `Chunk` class above, called `extract_words`. This function takes each debate and creates a list for each unique speaker along with an inner list that collates their individual words. In the code chunk below, the `Chunk` class is again instantiated on the chunked dictionary.

```
# Instantiate the `Chunk` class
chunks = Chunk(debate_chunked_year)

# For each debate, extract words for each speaker
extracted_words = chunks.extract_words(debate_chunked_year)
```

An example of the word extraction output (truncated) is shown below:

```
# Loop through each debate year
for year, speakers in extracted_words.items():
  print(f"\nDebate year: {year}")

  # Loop through each speaker in each debate year
  for speaker in speakers:
    print(f"{speaker[0]}: {speaker[1][:7]}")
```

```
Debate year: 2000
MODERATOR: ['Good', 'evening', 'from', 'the', 'Clark', 'Athletic', 'Center']
GORE: ['Well', 'Jim', 'first', 'of', 'all', 'I', 'would']
BUSH: ['Well', 'we', 'do', 'come', 'from', 'different', 'places']

Debate year: 2004
SCHIEFFER: ['Good', 'evening', 'from', 'Arizona', 'State', 'University', 'in']
KERRY: ['Well', 'first', 'of', 'all', 'Bob', 'thank', 'you']
BUSH: ['Bob', 'thank', 'you', 'very', 'much', 'I', 'want']
```

```
Debate year: 2008
SCHIEFFER: ['Good', 'evening', 'And', 'welcome', 'to', 'the', 'third']
MCCAIN: ['Well', 'let', 'let', 'me', 'say', 'Bob', 'thank']
OBAMA: ['Well', 'first', 'of', 'all', 'I', 'want', 'to']

Debate year: 2012
LEHRER: ['Good', 'evening', 'from', 'the', 'Magness', 'Arena', 'at']
OBAMA: ['Well', 'thank', 'you', 'very', 'much', 'Jim', 'for']
ROMNEY: ['Thank', 'you', 'Jim', 'It', '’s', 'an', 'honor']

Debate year: 2016
HOLT: ['Good', 'evening', 'from', 'Hofstra', 'University', 'in', 'Hempstead']
CLINTON: ['How', 'are', 'you', 'Donald', 'Well', 'thank', 'you']
TRUMP: ['Thank', 'you', 'Lester', 'Our', 'jobs', 'are', 'fleeing']

Debate year: 2020
WALLACE: ['Good', 'evening', 'from', 'the', 'Health', 'Education', 'Campus']
BIDEN: ['How', 'you', 'doing', 'man', 'I', '’m', 'well']
TRUMP: ['How', 'are', 'you', 'doing', 'Thank', 'you', 'very']
```

I finally run a sanity check to make sure that roughly the right number of words is extracted. I do this by comparing the extracted word count to the word count from the each debate’s un-chunked transcript, which is defined as a function `compare_word_counts` in the `Chunk` class from Question 2a.

```
Chunk(extracted_words).compare_word_counts(debates_stripped, extracted_words)
```

```
2000 Debate
Raw word count:      17069
Extr. word count:    16939
Difference:          0.76%

2004 Debate
Raw word count:      16106
Extr. word count:    15969
Difference:          0.85%

2008 Debate
Raw word count:      15978
Extr. word count:    15780
Difference:          1.24%
```

```
2012 Debate
Raw word count:      17606
Extr. word count:    17423
Difference:          1.04%

2016 Debate
Raw word count:      17935
Extr. word count:    17594
Difference:          1.90%

2020 Debate
Raw word count:      20605
Extr. word count:    19740
Difference:          4.20%
```

The sanity check results show that the `extract_words` function is working as it should, given the small differentials between the raw word counts and the extracted word counts.

Question 2c

For each candidate, for each debate, count the number of words and characters and compute the average word length for each candidate. Compare these between candidates and over time in some basic fashion.

To complete this question, I wrote a function called `get_avg_word_length` which I included in the `Chunk` class in Question 2a. The function takes the extracted words as input, and calculates the average words for each candidate, for each debate:

```
# Instantiate `Chunk` class on `extracted_words`
avg_words = Chunk(extracted_words).get_avg_word_length(extracted_words)

for year, debate in avg_words.items():
  print(f"\nAverage word lengths for {year} debate:")

  for speaker in debate:
    print(f"{speaker[0]}: {speaker[1]}")
```

```

Average word lengths for 2000 debate:
MODERATOR: 4.51
GORE: 4.24
BUSH: 4.15

Average word lengths for 2004 debate:
SCHIEFFER: 4.23
KERRY: 4.12
BUSH: 4.25

Average word lengths for 2008 debate:
SCHIEFFER: 4.3
MCCAIN: 4.25
OBAMA: 4.3

Average word lengths for 2012 debate:
LEHRER: 4.27
OBAMA: 4.26
ROMNEY: 4.13
```

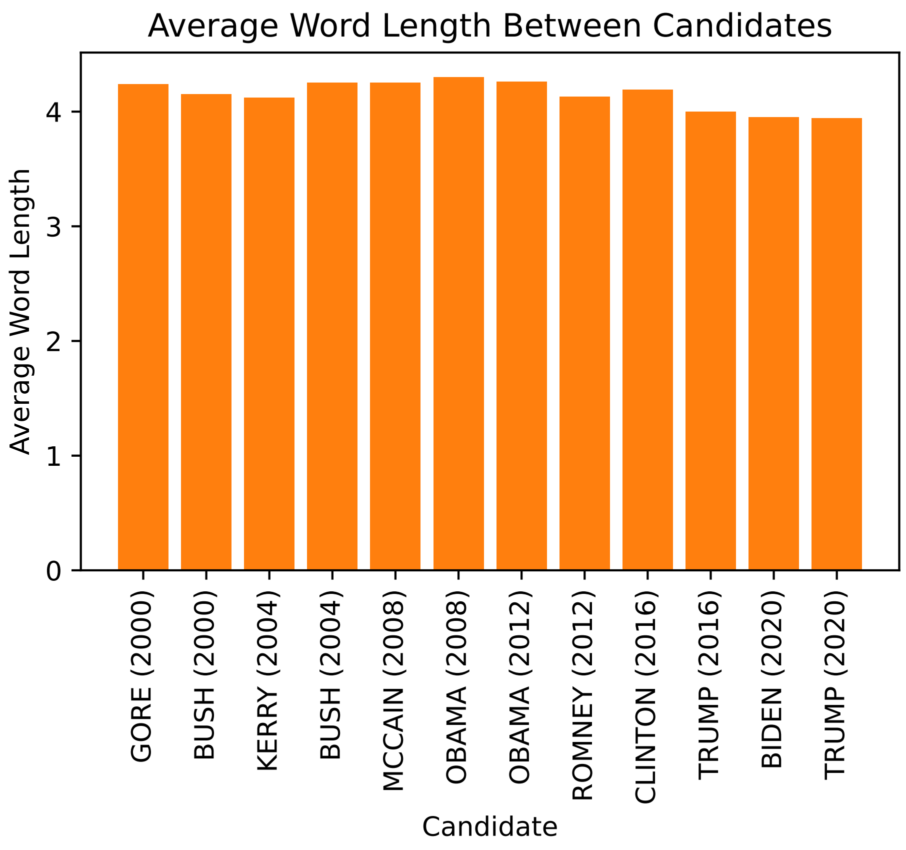
```
Average word lengths for 2016 debate:
HOLT: 4.4
CLINTON: 4.19
TRUMP: 4.0

Average word lengths for 2020 debate:
WALLACE: 4.37
BIDEN: 3.95
TRUMP: 3.94
```

A quick way to compare word lengths between candidates is to simply rank the candidates by “verbosity”, as well as by plotting bar charts.

```python
import pandas as pd

# Initialise an empty list to store verbosity rank
verbosity = []

for year, debates in avg_words.items():

  for speaker in debates:

    # We only care about candidates, not moderators:
    if speaker[0] not in moderators_unique:

      # Append a dict for each candidate and their avg word length
      verbosity.append({
        'Candidate':        speaker[0],
        'Year':             year,
        'Avg Word Length':  speaker[1],
        'Candidate-Year':   f"{speaker[0]} ({year})"
      })

# Convert the lsit of dicts to a dataframe
df_verbosity = pd.DataFrame(verbosity)

# Rank the candidates by verbosity
print(df_verbosity
      .sort_values("Avg Word Length", ascending=False)
      .drop("Candidate-Year", axis=1))
```

```
# Plot bar charts
for candidate in verbosity:
  plt.bar(df_verbosity["Candidate-Year"],
          df_verbosity["Avg Word Length"],
          )

# Plot formatting, including changing x-axis to go up in increments of 4
plt.title("Average Word Length Between Candidates")
plt.xticks(rotation=90)
plt.xlabel("Candidate")
plt.ylabel("Average Word Length")
plt.show()
```

```
   Candidate  Year  Avg Word Length
5      OBAMA  2008             4.30
6      OBAMA  2012             4.26
3       BUSH  2004             4.25
4     MCCAIN  2008             4.25
0       GORE  2000             4.24
8    CLINTON  2016             4.19
1       BUSH  2000             4.15
7     ROMNEY  2012             4.13
2      KERRY  2004             4.12
9      TRUMP  2016             4.00
10     BIDEN  2020             3.95
11     TRUMP  2020             3.94
```

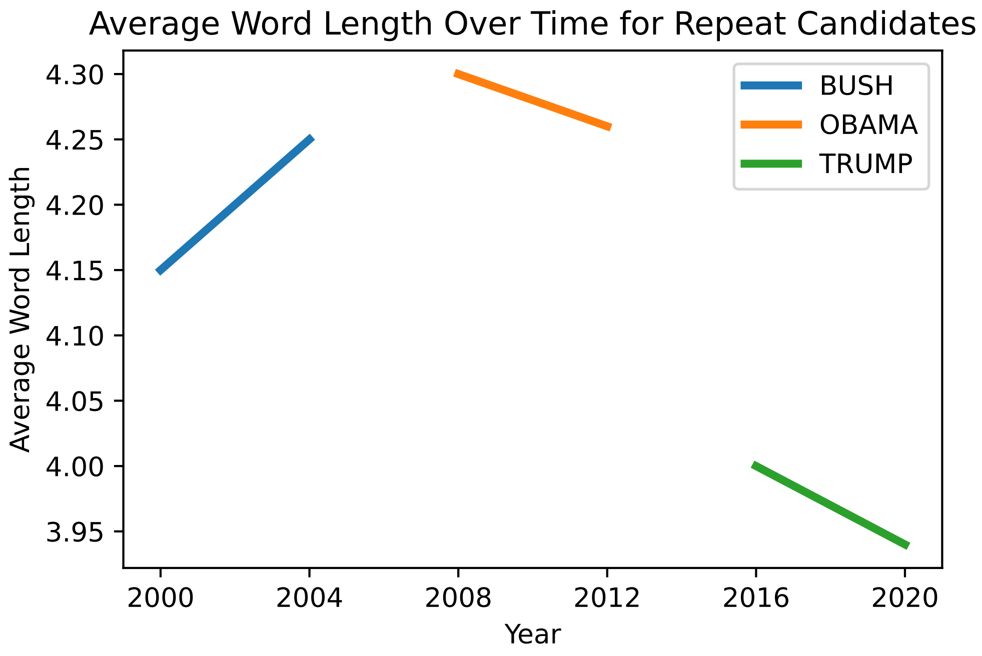


We see a general trend that the average word length decreases as time goes on. Could this have something to do with Trump and Biden being more accelerated in age?

To rank candidates' verbosity over time, we extract data for candidates who have appeared in more than one debate. We can then do a simple plot of their average word lengths over time.

```python
import matplotlib.pyplot as plt

# Count occurrences of each candidate
counts = df_verbosity["Candidate"].value_counts()

# Make list of candidates who have appeared more than once
double_candidates = counts[counts > 1].index.tolist()

# Create second df with double_candidates
df_verbosity_time = df_verbosity[
  df_verbosity["Candidate"].isin(double_candidates)
]

print(df_verbosity_time.sort_values("Year").drop("Candidate-Year", axis=1))

# Plot line charts
for candidate in double_candidates:
  isolate_candidate = df_verbosity[df_verbosity["Candidate"] == candidate]
  plt.plot(isolate_candidate["Year"],
           isolate_candidate["Avg Word Length"],
           label=candidate,
           linewidth=3
          )

# Plot formatting, including changing x-axis to go up in increments of 4
plt.title("Average Word Length Over Time for Repeat Candidates")
plt.xticks(range(
              min(df_verbosity["Year"]),
              max(df_verbosity["Year"]) + 1,
              4
            ))
plt.xlabel("Year")
plt.ylabel("Average Word Length")
plt.legend()
plt.show()
```

```
   Candidate  Year  Avg Word Length
1       BUSH  2000             4.15
3       BUSH  2004             4.25
5      OBAMA  2008             4.30
6      OBAMA  2012             4.26
9      TRUMP  2016             4.00
11     TRUMP  2020             3.94
```



The main thing to note is how low Trump's average word length is compared to Bush and Obama.

Question 2d

Do some checking of your functions on simple test inputs. For the sake of time, you don't need to set up formal testing, and this doesn't need to be extensive.

I have already implemented error checking and sanity checks in the questions above–please refer to each respective question part. For further validation, I can create dummy dialogues and run by functions on it:

```python
# Create a simple test input
dummy_string = 'TRUMP: That's not what you've said and it's not what your \
party is saying.BIDEN: That is simply a lie.TRUMP: Your party doesn't \
say it. Your party wants to go socialist medicine and socialist \
healthcare.BIDEN: The party is me. Right now, I am the Democratic Party.\
TRUMP: And they're going to dominate you, Joe. You know that.BIDEN: \
I am the Democratic Party right now.TRUMP: Not according to Harris.'

# Initiate the chunk class on the test input
dummy = Chunk(dummy_string)
dummy_year = 2020

# Get chunks
print("\nGetting chunks (Question 2a)...")
dummy_chunked = dummy.get_chunks(dummy_year)
for chunk in dummy_chunked:
  print(f"{chunk[0]}: {chunk[1][:22]} ...")

# Count number of chunks per speaker
print("\nCounting number of chunks per speaker (Question 2a) ...")
dummy_chunk_counts = dummy.count_chunks(dummy_chunked, speakers_unique)
print(dummy_chunk_counts)

# Extract words from chunks
print("\nExtracting words (Question 2b) ...")
dummy_dict = {dummy_year: dummy_chunked}
dummy_extracted = dummy.extract_words(dummy_dict)
for year, speakers in dummy_extracted.items():
  for speaker in speakers:
    print(f"{speaker[0]}: {speaker[1][:8]} ...")
```

```
# Calculate average word length
print("\nCalculating average word length (Question 2c) ...")
avg_words = Chunk(dummy_extracted).get_avg_word_length(dummy_extracted)
for year, debate in avg_words.items():
  for speaker in debate:
    print(f"{speaker[0]}: {speaker[1]}")
```

```
Getting chunks (Question 2a)...
TRUMP: That’s not what you’ve ...
BIDEN: That is simply a lie. ...
TRUMP: Your party doesn’t say ...
BIDEN: The party is me. Right ...
TRUMP: And they’re going to d ...
BIDEN: I am the Democratic Pa ...
TRUMP: Not according to Harri ...

Counting number of chunks per speaker (Question 2a) ...
{'TRUMP': 4, 'BIDEN': 3}

Extracting words (Question 2b) ...
TRUMP: ['That', '’s', 'not', 'what', 'you', '’ve', 'said', 'and'] ...
BIDEN: ['That', 'is', 'simply', 'a', 'lie', 'The', 'party', 'is'] ...

Calculating average word length (Question 2c) ...
TRUMP: 4.13
BIDEN: 3.74
```

The output above on a simple input verifies that the functions in my class work as expected.

Question 2e

(Extra credit) For each candidate, count the following words or word stems (feel free to replace some or all of these with words or expressions of your choice) and store in a data structure: I, we, America{,n}, democra{cy,tic}, republic, Democrat{,ic}, Republican, free{,dom}, terror{,ism}, safe{,r,st,ty}, {Jesus, Christ, Christian}. Make a plot or two and comment briefly on the results.

First, I edit the list of words to make them regex-able.

I try to collate each candidate's words from across different debate years, as I'm not comparing over the time dimension.

I then tally the counts that each expression appears in each candidate's words.

I create a dataframe with each column corresponding to the words/expressions above, and I save my word counts for each candidate in this dataframe.

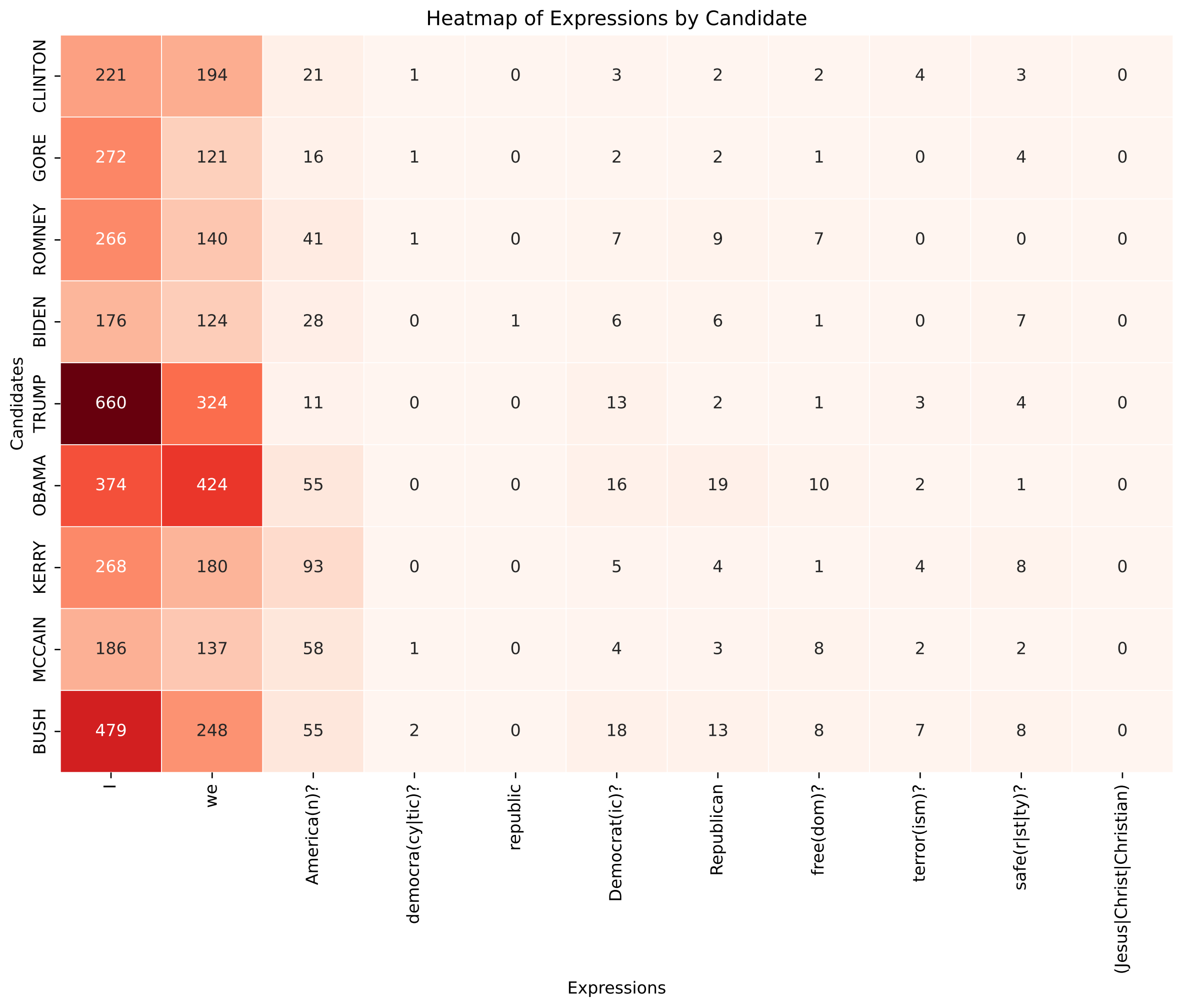
```
exprs = [
  r"I",
  r"we",
  r"America(n)?",
  r"democra(cy|tic)?",
  r"republic",
  r"Democrat(ic)?",
  r"Republican",
  r"free(dom)?",
  r"terror(ism)?",
  r"safe(r|st|ty)?",
  r"(Jesus|Christ|Christian)",
]


# Create a new list to aggregate each candidates words
expr_list = []
[expr_list.append([name, []]) for name in candidates_unique]

for name in expr_list:
  for speakers in extracted_words.values():
    # Loop through each speaker in each debate year
    for speaker in speakers:
      if speaker[0] == name[0]:
        name[1].extend(speaker[1])
        break

# Create a new list to store counts of each expression for each candidate
expr_counts = []
for i in range(len(expr_list)):
  inner = [expr_list[i][0], []]
  expr_counts.append(inner)

# Now count the occurrences that `exprs` appear for each candidate
for i, name in enumerate(expr_counts):
  for expr in exprs:
    count = 0

    # Replace curly brackets with square brackets
    pattern = expr
    matches = re.findall(pattern, " ".join(expr_list[i][1]))

    # Count the occurrences of the word
    count += len(matches)

    expr_counts[i][1].append(count)


# Initialise dataframe
df = pd.DataFrame(columns=["Candidate"] + exprs)

# Append the counts of `exprs` to the dataframe
for candidates in expr_counts:
  # Create new row with candidate name and their respective counts
  new_row = [candidates[0]] + candidates[1]

  # Append the new row to the DataFrame
  df.loc[len(df)] = new_row

print(df)
```

```
  Candidate    I   we  America(n)?  democra(cy|tic)?  republic  Democrat(ic)?  \
0   CLINTON  221  194           21                 1         0              3
1      GORE  272  121           16                 1         0              2
2    ROMNEY  266  140           41                 1         0              7
```

```
3     BIDEN  176  124            28                   0         1             6
4     TRUMP  660  324            11                   0         0            13
5     OBAMA  374  424            55                   0         0            16
6     KERRY  268  180            93                   0         0             5
7    MCCAIN  186  137            58                   1         0             4
8      BUSH  479  248            55                   2         0            18

   Republican  free(dom)?  terror(ism)?  safe(r|st|ty)?  \
0           2           2             4               3
1           2           1             0               4
2           9           7             0               0
3           6           1             0               7
4           2           1             3               4
5          19          10             2               1
6           4           1             4               8
7           3           8             2               2
8          13           8             7               8

   (Jesus|Christ|Christian)
0                         0
1                         0
2                         0
3                         0
4                         0
5                         0
6                         0
7                         0
8                         0
```

To visualise the data, I can use a heatmap:

```python
import seaborn as sns

df_plot = df.set_index('Candidate')

plt.figure(figsize=(12, 8))
sns.heatmap(df_plot, annot=True, cmap='Reds', fmt='d', linewidth=.5, cbar=False)
plt.title('Heatmap of Expressions by Candidate')
plt.xlabel('Expressions')
plt.ylabel('Candidates')
plt.show()
```



The plot highlights the use of the words “I” and “we” between the different candidates. It’s interesting that the highest use of “I” is by Trump (or perhaps it isn’t that interesting given what we know). Bush and Obama follow, highlighting perhaps the relative importance that each candidate places on themselves.

It’s worth noting that the more “egotistical” candidates also score highly on using “we” (namely Obama, Trump, and Bush). It could just be that these three candidates are fond of using pronouns more than the others.

The other insight is that Kerry seems to accentuate “America(n)”, while Trump (ironically) and Gore score lowly on this expression.

Question 3

Now sketch out a design for a functional programming (FP) approach (if your solution to Problem 2 used OOP) or an OOP approach (if your solution to Problem 2 used functional programming). If you're designing an OOP approach, decide what the classes would be and the fields and methods of those classes. If you're designing a FP approach, decide what the functions would be and what inputs/output they would use. **To be clear, you do not have to write any of the code for the methods/classes/functions; the idea is just to design the code.** As your response in the OOP case, for each class, please provide a bulleted list of methods and bulleted list of fields and for each item briefly comment what the purpose is. Or in the FP case, for each function, provide a bulleted list of inputs and output and briefly comment on the purpose of each function.

Since I used an OOP approach, I will answer this question with an FP approach in mind.

- **Function**: `find_non_spoken(debates_body)`
  - **Input**: A list of strings (i.e. debate transcripts) that has been pre-processed
  - **Output**: A list of non-spoken words (e.g. "Applause", "Laughter")
  - **Purpose**: Finds non-spoken words from the transcript
- **Function**: `strip_non_spoken(debates_body, non_spoken_list)`
  - **Input**: The processed debate transcripts as strings, and the list of non-spoken texts from the previous function
  - **Output**: An updated list of debate transcripts with non-spoken words removed
  - **Purpose**: Strips out occurrences of non-spoken words from the transcript
- **Function**: `fix_colons(debate)`
  - **Input**: A string of an individual debate transcript
  - **Output**: A string of debate with properly formatted colons
  - **Purpose**: Ensures proper spacing after colons in the speaker identifiers
- **Function**: `count_non_spoken(debates_body, non_spoken_word)`
  - **Input**: Strings of debate transcripts and a string of non-spoken words to count
  - **Output**: Integer counts of occurrences of the specified non-spoken word in each debate
  - **Purpose**: Sanity check to ensure that non-spoken words were properly removed from the trnascript
- **Function**: `compare_pre_post_strip(debates_body, non_spoken_word)`

  - **Input**: Strings of debate transcripts and a string of non-spoken words to count
  - **Output**: Tuple with pre-strip counts and post-strip counts
  - **Purpose**: Sanity check to compare the counts of a specified non-spoken word before and after stripping non-spoken text.

- **Function**: `get_debate_year(transcript)`
  - **Input**: String of individual debate transcripts
  - **Output**: Year of debate as an integer
  - **Purpose**: Extracts the debate year from the transcript using regex, to build a dictionary

- **Function**: `strip_preamble(transcript, moderators)`
  - **Input**: String of individual debate transcripts, and list of moderators
  - **Output**: String of individual debate transcript but with preamble formatting removed
  - **Purpose**: Strips the introductory content of the transcript up to the first moderator's mention

- **Function**: `get_chunks(transcript, valid_speakers)`
  - **Input**: String of individual debate transcripts, and list of speakers from a specific debate year
  - **Output**: List of tuples (each containing a speaker identifier and their dialogue)
  - **Purpose**: List of tuples (each containing a speaker identifier and their dialogue)

- **Function**: `count_chunks(chunks, speakers)`
  - **Input**: List of tuples containing the speaker and their dialogue, and a list of valid speaker names
  - **Output**: A dictionary with speaker names as keys, and their chunk counts as values
  - **Purpose**: Sanity check to count the number of dialogue chunks per speaker

- **Function**: `check_chunk_repeats(debate_chunked)`
  - **Input**: List of tuples containing the speaker and their dialogue
  - **Output**: Integer count of repeated speakers in consecutive chunks
  - **Purpose**: Sanity check to verify there are no occurrences of the same speaker being in consecutive dialogue chunks

- **Function**: `extract_words(debate_chunked_year)`
  - **Input**: : Dictionary where keys are debate years and values are lists of dialogue chunks
  - **Output**: Dictionary with debate years as keys and lists of speakers and their spoken words as values
  - **Purpose**: Extracts words spoken by each speaker for each debate year using `spaCy` nlp

- **Function**: `get_avg_word_length(extracted_words)`
  - **Input**: : Dictionary where keys are debate years and values are lists of dialogue chunks
  - **Output**: Dictionary with debate years as keys and average word lengths for each speaker
  - **Purpose**: Calculates the average word length for each speaker in each debate year
- **Function**: `get_avg_word_length(extracted_words)`
  - **Input**: : List of cleaned debate transcripts, and a dictionary with debate years as keys and speakers' extracted words as values
  - **Output**: Prints raw (un-chunked) word counts, extracted word counts using nlp, and their percentage difference for each debate year
  - **Purpose**: Sanity check that the difference in word counts between un-chunked transcripts and extracted words is reasonable, to verify if the `get_avg_word_length` function operated as intended or not