STAT 243 Problem Set 3

Treves Li

2024-09-27

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()`:

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

```
['__all__',
 '__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__init__',
 '__loader__',
 '__name__',
 '__package__',
 '__path__',
 '__spec__',
 '__version__',
 '__version_info__',
 '__version_tuple__',
 '_version',
 'api',
 'base',
 'compat',
 'datasets',
 'debug_warnings',
 'discrete',
 'distributions',
 'duration',
 'emplike',
 'formula',
 'gam',
 'genmod',
 'graphics',
 'imputation',
 'iolib',
 'monkey_patch_cat_dtype',
 'multivariate',
 'nonparametric',
 'regression',
 'robust',
 'sandbox',
 'stats',
 'test',
 'tools',
 'tsa']
```

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:

```
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:

```
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:

```
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:

```
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:

```
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:

```
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.

```
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`.

```
# 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 = list(set(candidates))
moderators = list(set(moderators))

# Combine to make a list of speakers
speakers = candidates + moderators

print("\nCandidate names:")
print(candidates)
print("\nModerator names:")
print(moderators)
```

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

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

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.

```
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. Furthermore, I remove dictionary items where the counts are less than 5, since these are likely not real chunks, but some artefact of the transcript metadata or the preamble.

`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.

```
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):
    """
    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.
    """
    # Use regex to find all chunks
    chunk_pattern = r"([A-Z]+): (?<=[A-Z]: )(.*?)(?=[A-Z]*:|$)"
    chunk_matches = re.findall(chunk_pattern, self.transcript)
    matches_list = [list(match) for match 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):
    """
    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
    }

    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
```

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()
  debate_chunked_year[year] = debate_chunked

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

```python
# 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