STAT 243 Problem Set 3

Treves Li

2024-09-25

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__',
 '__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:

```
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`.

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 being in parentheses or square brackets. But I ignore words that are two letters or fewer, as it seems to flag party affiliation (like "(D)" or "(R)") or state names (like "(MA)" or "(AZ)").

```
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.
    """
    import re

    # Regex to find all single words that appear within parentheses
    # and square brackets
    # Exclude words with two or fewer characters
    non_spoken_pattern = r"[\(\[]\w{3,}[\)\]]"
    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 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}': \n\
```

```
      Pre-strip:\t {counts_pre_strip} \n\
      Post-strip:\t {counts_post_strip}\n")

debates = Debate(debates_body)
```

In the code below, I run some sanity checks:

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

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

Counts of 'laughter':
      Pre-strip:     [0, 6, 0, 4, 5, 0]
      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".