STOR 235 — Lab 2

In this lab, you will use what you've learned about vectors to determine the authors of the disputed Federalist Papers. More background on this task is available in the notes.

Instructions

There are four parts to this lab:

1. Lists, strings, and sets in Python;
2. Text processing and cleaning in Python;
3. Authorship analysis using word counts;
4. Authorship analysis using bigrams.

Each part will be composed of a number of tasks for you to complete. The tasks are labeled in bold as **Problem 1**, **Problem 2**, and so on. Please make sure that you do not forget to complete any of the problems. Each problem will be worth 4 points.

There is a single code block in under this introductory material that imports various things you will use later. Please run it.

Important Information About Submitting Your Assignment

On Gradescope, this assignment is broken into 4 questions, representing each of the four parts. Please upload your assignment as a PDF file, and make sure to select to all relevant pages for each part.

```
import requests
import numpy as np
from nltk.tokenize import word_tokenize
from nltk.util import bigrams
import nltk
nltk.download('punkt_tab')
```

```
[nltk_data] Downloading package punkt_tab to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt_tab.zip.
True
```

Part 1: Lists, Strings, Dictionaries, and Sets

The following code blocks introduce lists, strings, dictionaries, and sets in Python, as well as some basic functions for working with them. Please run the demonstration code blocks and do the associated problems.

Lists are ordered collections (e.g., of numbers or strings). A string is an ordered collection of individual characters, and acts in much the same way as a list. (It must be enclosed in single or double quotes.) A dictionary is a collection of ordered pairs, called keys and values. A set, like in mathematics, is an unordered collection of objects of the same kind (e.g., of numbers or strings).

Important Note. Some Python data structures, including lists and strings, are indexed in an interesting way: the first element has index 0, the second element has index 1, and so on. The "slice" notation [x:y] below means to take all elements between x and y, inclusive of x and exclusive of y.

This will all be much clearer after you see the examples. We begin with strings.

```
my_string = "This is a string."
print(my_string)

my_other_string = "This is another string."
big_string = my_string + " " + my_other_string
print(big_string)

print(big_string[0:5])

#the following (somewhat odd) syntax can be used to insert other data types,
#such as numbers, into the middle of strings
#This is called "string interpolation"
year = 2025
print(f"The year is {year}.")
```

```
This is a string.
This is a string. This is another string.
```

```
This
The year is 2025.
```

Problem 1. Write a function that takes a number as input and prints "My favorite number is [input]." Here [input] represents the input to the function. Demonstrate that the function works by running it with your favorite number and verifying that the output is correct.

```
def favorite_number(number):
  print(f"My favorite number is {number}.")
favorite_number(11)
```

```
My favorite number is 11.
```

We next consider lists.

```
a = [1,2,3]
print(a)
b = ["This", "is", "a", "list"]
print(b)

c = [4,5,6]
print(c)
print(a+c)

c.append(7)
print(c)
print(c[0])
print(c[1:3])
```

```
[1, 2, 3]
['This', 'is', 'a', 'list']
[4, 5, 6]
[1, 2, 3, 4, 5, 6]
[4, 5, 6, 7]
4
[5, 6]
```

Problem 2. Write a function that takes as input a positive integer N and returns a list of the odd numbers between 1 and N (inclusive). You can use the expression k % 2 in Python to check for remainder upon division by 2 (try it!). Run your function with the input 13 and print the output.

```
def odd_number_list(N):
    odd_numbers = []
    for k in range(1, N+1):
        if k % 2 != 0:
            odd_numbers.append(k)
    return odd_numbers

print(odd_number_list(13))
```

```
[1, 3, 5, 7, 9, 11, 13]
```

Next on our tour: dictionaries.

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

print(thisdict['brand'])

thisdict['country'] = "USA"

print(thisdict['country'])
print(thisdict)
print(sorted(thisdict)) #creates a list in alphabetical order
```

```
Ford
USA
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'country': 'USA'}
```

```
['brand', 'country', 'model', 'year']
```

Finally, we consider sets, which are unordered collections.

```
myset = {"UNC", "Duke", "NC State"}
print(myset)

myset.add("Wake Forest")
print(myset)

myset.remove("Duke")
print(myset)
```

```
{'NC State', 'UNC', 'Duke'}
{'Wake Forest', 'NC State', 'UNC', 'Duke'}
{'Wake Forest', 'NC State', 'UNC'}
```

Part 2: Text Processing

The command `word_tokenize`, which we imported at the beginning of this notebook, splits a string into a list individual words. The following code block gives an example.

```
s0 = "How much wood would a woodchuck chuck if a woodchuck could chuck wood? As much wood as a woodchuck would if a woodchuck cc
print(s0)
words = word_tokenize(s0)
print(words)
```

```
How much wood would a woodchuck chuck if a woodchuck could chuck wood? As much wood as a woodchuck would if a woodchuck coul
['How', 'much', 'wood', 'would', 'a', 'woodchuck', 'chuck', 'if', 'a', 'woodchuck', 'could', 'chuck', 'wood', '?', 'As', 'mu
```

For our analysis, we want to get rid of punctuation and just look at words. We also want to ignore capitalization.

Problem 3. The following code takes a string and checks each character to see whether it is a letter or a space. It then adds the character to a new string `cleaned_string` if it is one of these two types, or does nothing (discarding it) if not. Finally, it returns the new string. The end result is to remove all punctuation and numerals from the input, and put the remaining characters in lower case. However, one line is missing. Fix the function so that it has the desired behavior, so that the following code block returns the appropriate version of `s0`.

```
def clean_string(str):
    # Keep only alphabetical characters in the string
    cleaned_string = ""
    for char in str:
        if char.isalpha() or char==" ":  # Check if the character is a letter or a space
            cleaned_string += char
    return cleaned_string.lower()
```

```
clean_string(s0)
```

```
'how much wood would a woodchuck chuck if a woodchuck could chuck wood as much wood as a woodchuck would if a woodchuck cou
ld chuck wood'
```

Problem 4. The next function creates a dictionary from a string, after cleaning it up using the previous function. The dictionary has as keys the words that appear in the string, and as values the number of times they appear. There is a line missing. Fix it so that it has the desired behavior.

```
def word_count_dictionary(str1):
#returns a dictonary containing frequencies of any word in string
#e.g. str1 = 'quick brown fox is quick.'
# returns {quick:2, brown:1, fox:1, is:1}
  x = {}
  str1 = clean_string(str1)
  words = word_tokenize(str1)
  for b in words:
        if b in x:
            x[b]+=1
        else:
            x[b] = 1
  return(x)
```

```
word_count_dictionary(s0)
```

```
{'how': 1,
 'much': 2,
 'wood': 4,
 'would': 2,
 'a': 4,
 'woodchuck': 4,
 'chuck': 3,
 'if': 2,
 'could': 2,
 'as': 2}
```

The next function takes a variable number of dictionaries using the special * construction, which means the function can take any number of arguments; the arguments can then be used as a list. It creates a set of all the words that appear, then returns them in a list in alphabetical order.

```
def build_clean_set(*dicts):
    s = set() #makes empty set
    for d in dicts:
        for word in d.keys():
          s.add(word)
    return sorted(s)


food = {"cucumber" : 1, "apple" : 2, "banana" : 3}
more_food = {"pizza" : 4, "ice cream" : 6}
even_more_food = {"salad" : 99}

food_set = build_clean_set(food, more_food, even_more_food)
print(food_set)
```

```
['apple', 'banana', 'cucumber', 'ice cream', 'pizza', 'salad']
```

The following two functions are what build the word vectors we want. The first function takes in a list of words and a dictionary, and returns a list of the keys for each word in the original list.

```
def word_vector(word_set, word_dict):
    vector = []
    for word in word_set:
        # Get the value from word_dict or use 0 if the word is not found
        value = word_dict.get(word, 0)
        vector.append(value)
    return vector


word_vector(food_set,food)
```

```
[2, 3, 1, 0, 0, 0]
```

The next function takes in a variable number of strings and uses the previous function to build a list of word vectors.

```
def make_vectors(*strings):
  word_dicts = []
  for s in strings:
    word_dicts.append(word_count_dictionary(s))
  word_set = build_clean_set(*word_dicts)
  vectors = []
  for d in word_dicts:
    vectors.append(word_vector(word_set,d))
  print(word_set)
  return vectors


s1 = "I love watching Alfred Hitchcock movies."
s2 = "I hate watching Alfred Hitchcock movies."
s3 = "Squirrels love to attack my bird feeder."


make_vectors(s1,s2,s3)
```

```
['alfred', 'attack', 'bird', 'feeder', 'hate', 'hitchcock', 'i', 'love', 'movies', 'my', 'squirrels', 'to', 'watching']
[[1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1],
 [1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1],
```

```
        [0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0]]
```

Problem 5. Create a function that takes in two lists of numbers and calculates the angle between them in degrees (considered as vectors). The function has been started for you in the code block below.

The following commands from numpy may be useful: `linalg.norm`, `dot` (for inner product), `arccos`, and `degrees`. Remember to prefix them by `np`.

```
def angle(u, v):
    # Convert to numpy arrays
    u = np.array(u)
    v = np.array(v)

    u_norm = np.linalg.norm(u)
    v_norm = np.linalg.norm(v)
    inner_product = np.dot(u,v)
    angle_in_degrees = np.degrees(np.arccos(inner_product/(u_norm*v_norm)))
#finish the function by writing here

    return angle_in_degrees
```

Problem 6. Using `make_vectors`, compute the angles between all three pairs of vectors formed by the set `s1`, `s2`, and `s3`. Note that we expect `s1` and `s2` to make a smaller angle than `s1` and `s3`, because they have fewer words in common.

```
v = make_vectors(s1,s2,s3)

print(angle(v[0],v[1]))
print(angle(v[0],v[2]))
print(angle(v[1],v[2]))
```

```
['alfred', 'attack', 'bird', 'feeder', 'hate', 'hitchcock', 'i', 'love', 'movies', 'my', 'squirrels', 'to', 'watching']
33.557309761920706
81.12360491902666
90.0
```

Problem 7. The following code block contains a string `s4`. Write code that computes the angle between `s4` and each of `s1`, `s2`, and `s3`, and prints the result.

```
s4 = "The Netflix film Do Revenge is a 2022 remake of Strangers on a Train by Alfred Hitchcock."

v = make_vectors(s1,s2,s3,s4)
print(angle(v[0],v[3]))
print(angle(v[1],v[3]))
print(angle(v[2],v[3]))
```

```
['a', 'alfred', 'attack', 'bird', 'by', 'do', 'feeder', 'film', 'hate', 'hitchcock', 'i', 'is', 'love', 'movies', 'my', 'net
78.90419671686361
78.90419671686361
90.0
```

Part 3: Authorship Analysis with Word Counts

To make this lab go faster, I've pre-processed the relevant federalist papers to remove irrelevant material (like dates and the pseudonymous signature), and place the resulting files on my website. The files are:

- `hamilton.txt`, containing all papers undisputedly authored by James Madison, concatenated into a single big text file.
- `madison.txt`, containing all papers undisputedly authored by James Madison, also concatenated into a single big text file.
- Tweleve disputed papers named `disputedX.txt`, where x ranges from 0 to 11.

Problem 8. The three code boxes below should download the text files and print small snippets from them. But the code for Madison's papers is missing. Fix it. Make sure the file goes into a string called `madison`, and print the first 200 characters of this string.

```
url_hamilton = "https://lopat.to/lab2/hamilton.txt"
response_h = requests.get(url_hamilton)
hamilton = response_h.text  # Set the content as the string variable
```

```
# Print a snippet to verify
print(hamilton[0:200])  # Show the first 200 characters of the string
```

```
AFTER an unequivocal experience of the inefficacy of the subsisting federal government, you are called upon to deliberate on
```

```
url_madison = "https://lopat.to/lab2/madison.txt"
response_m = requests.get(url_madison)
madison = response_m.text
print(madison[0:200])
```

```
AMONG the numerous advantages promised by a well constructed Union, none deserves to be more accurately developed than its t
```

```
base_url_disputed = "https://lopat.to/lab2/disputed"  # Base URL for the disputed files
num_files = 12  # Number of files to download (disputed0.txt, disputed1.txt, etc.)
disputed = []

for i in range(num_files):
    url = f"{base_url_disputed}{i}.txt"  # Construct the URL
    response = requests.get(url)
    disputed.append(response.text)  # Add the content to the list

# Print a snippet of each file to verify
for i, content in enumerate(disputed):
    print(f"disputed{i}.txt preview:")
    print(content[0:100])  # Show the first 100 characters
```

```
disputed0.txt preview:
THE author of the "Notes on the State of Virginia," quoted in the last paper, has subjoined to that
disputed1.txt preview:
IT MAY be contended, perhaps, that instead of OCCASIONAL appeals to the people, which are liable to
disputed2.txt preview:
TO WHAT expedient, then, shall we finally resort, for maintaining in practice the necessary partitio
disputed3.txt preview:
FROM the more general inquiries pursued in the four last papers, I pass on to a more particular exam
disputed4.txt preview:
I SHALL here, perhaps, be reminded of a current observation, "that where annual elections end, tyran
disputed5.txt preview:
THE next view which I shall take of the House of Representatives relates to the appointment of its m
disputed6.txt preview:
THE number of which the House of Representatives is to consist, forms another and a very interesting
disputed7.txt preview:
THE SECOND charge against the House of Representatives is, that it will be too small to possess a du
disputed8.txt preview:
THE THIRD charge against the House of Representatives is, that it will be taken from that class of c
disputed9.txt preview:
THE remaining charge against the House of Representatives, which I am to examine, is grounded on a s
disputed10.txt preview:
HAVING examined the constitution of the House of Representatives, and answered such of the objection
disputed11.txt preview:
A FIFTH desideratum, illustrating the utility of a senate, is the want of a due sense of national ch
```

Problem 9. Use `make_vectors` to build a list of word count vectors named `papers`. (Yes, this is a list of lists!) The first entry should correspond to Hamilon, the second to Madison, and then the disputed papers in numerical order. You can use `*disputed` as an argument of `make_vectors` to "unpack" the list into a series of arguments.

```
papers = make_vectors(hamilton, madison, *disputed)
```

```
['a', 'abandon', 'abandoned', 'abandoning', 'abate', 'abatements', 'abetted', 'abhorrence', 'abilities', 'ability', 'able',
```

Problem 10. Compute the angles between the Hamilton vector and all of the disputed papers, and between the Madison vector and the disputed papers.

```
print(angle(papers[0],papers[2]))
print(angle(papers[1],papers[2]))

for i in range(2,len(papers)):
  print(angle(papers[0],papers[i]))
  print(angle(papers[1],papers[i]))
```

```
13.282385915700406
11.835838969288861
13.282385915700406
11.835838969288861
18.542219226944678
16.856261454584658
```

```
15.990602732166584
13.686306408384885
13.721351176631522
12.48243848923609
13.291787974388406
12.222069307207665
15.30568858429241
14.12865785430726
13.501834982889452
14.264172459043317
17.37785628976469
17.111340705228823
13.011412115874109
11.998594904063307
13.943927627364406
12.384909762238395
13.121471378497569
13.678764931468141
12.805882762894047
11.117080514323058
```

Part 4: Bigrams

A bigram is a pair of words.

Problem 11. The following code is similar to the dictionary creation code that you saw before, but it now creates a dictionary of bigrams. Again, please fix it so that it the code blocks following give the correct result.

```
def bigram_dictionary(str1):
#returns a dictonary containing frequencies of any word in string
#e.g. str1 = 'quick brown fox is quick.'
# returns {quick:2, brown:1, fox:1, is:1}
  x = {}
  str1 = clean_string(str1)
  words = word_tokenize(str1)
  bigram_tokens = bigrams(words)
  for b in bigram_tokens:
        if b in x:
            x[b]+=1
        else:
            x[b] = 1
  return(x)
```

```
def make_bigram_vectors(*strings):
  word_dicts = []
  for s in strings:
    word_dicts.append(bigram_dictionary(s))
  word_set = build_clean_set(*word_dicts)
  vectors = []
  for d in word_dicts:
    vectors.append(word_vector(word_set,d))
  print(word_set)
  return vectors
```

```
bs = make_bigram_vectors(s1,s2,s3)
```

```
[('alfred', 'hitchcock'), ('attack', 'my'), ('bird', 'feeder'), ('hate', 'watching'), ('hitchcock', 'movies'), ('i', 'hate')
```

Problem 12. Re-do the analysis from the previous section using bigram vectors. Compute the angles between the Hamilton bigram vector and all of the disputed papers, and between the Madison bigram vector and the disputed papers.

```
bigrams = make_bigram_vectors(hamilton,madison,*disputed)
```

```
[('a', 'bad'), ('a', 'bar'), ('a', 'bare'), ('a', 'barrier'), ('a', 'belief'), ('a', 'beneficial'), ('a', 'benefit'), ('a',
```

```
for i in range(2,len(papers)):
  print(angle(bigrams[0],bigrams[i]))
  print(angle(bigrams[1],bigrams[i]))
```

```
41.83059952131539
41.536884935886825
52.129867037562555
```

```
51.85558330815452
42.60365475227447
41.751706901241526
42.29720614470198
40.43574456320057
44.255894553096404
43.516761125499
47.14243515359013
45.8452109163408
47.11507419930604
46.069144214159195
45.89936608597893
45.873505864393465
43.04969860421117
42.66352901361986
46.45253967655734
45.689196708191155
44.00633360311356
43.81723604310804
43.52924896579982
42.30476110481267
```

Postscript

It is generally thought, on the basis of more sophisticated methods, that Madison was the author of all of the disputed papers. Your results should generally point in this direction. It is interesting that we can recover this finding from fairly elementary methods.

However, there are some obvious questions we should ask. In particular: How reliable are these results? Are we discovering a true signal, or just statistical noise? After all, these angle differences are quite small (which is perhaps to be expected, since the two reference vectors are from two authors writing about the same topic). The fact that both the single-word and bigram analyses are mostly consistent is reassuring, but not dispositive.

In the next lab, you will see less ad hoc ways to approach classification problems.