

Homework 2: Markov Models of Natural Language

Name: William Martinez

Collaborators: None

Date: 4/21/24

This homework focuses on topics related to string manipulation, dictionaries, and simulations.

I encourage collaborating with your peers, but the final text, code, and comments in this homework assignment should still be written by you. Please check the collaboration policy.

Submission instructions:

- Submit `HW2.py` and `HW2.ipynb` compressed in a one zip file on Gradescope under "HW2 - Autograder". Do **NOT** change the file name.
- Convert this notebook into a pdf file and submit it on GradeScope under "HW2 - PDF". Make sure your text outputs in the latter problems are visible.

Language Models

Many of you may have encountered the output of machine learning models which, when "seeded" with a small amount of text, produce a larger corpus of text which is expected to be similar or relevant to the seed text. For example, there's been a lot of buzz about the new [GPT-3 model](#), related to its [carbon footprint](#), [bigoted tendencies](#), and, yes, impressive (and often [humorous](#)) [ability to replicate human-like text in response to prompts](#).

We are not going to program a complicated deep learning model, but we will construct a much simpler language model that performs a similar task. Using tools like iteration and dictionaries, we will create a family of **Markov language models** for generating text. For the purposes of this assignment, an n -th order Markov model is a function that constructs a string of text one letter at a time, using only knowledge of the most recent n letters. You can think of it as a writer with a "memory" of n letters.

```
In [ ]: # This cell imports your functions defined in HW2.py
import random
```

```
from HW2 import count_characters, count_ngrams, markov_text
```

Data

Our training text for this exercise comes from Jane Austen's novel *Emma*, which Professor Chodrow retrieved from the archives at ([Project Gutenberg](#)). Intuitively, we are going to write a program that "writes like Jane Austen," albeit in a very limited sense.

```
In [ ]: with open('emma-full.txt', 'r') as f:
        s = f.read()
```

Problem 1: Define `count_characters` in `HW2.py`

Write a function called `count_characters` that counts the number of times each character appears in a user-supplied string `s`. Your function should loop over each element of the string, and sequentially update a `dict` whose keys are characters and whose values are the number of occurrences seen so far. Your function should then return this dictionary.

You may know of other ways to achieve the same result. However, you are encouraged to use the loop approach, since this will generalize to the next exercise.

Note: While the construct `for character in s:` will work for this exercise, it will not generalize to the next one. Consider using `for i in range(len(s)):` instead.

Example usage:

```
count_characters("Torto ise!")
{'T': 1, 't': 1, 'o': 2, 'r': 1, 'i': 1, 's': 1, 'e': 1, ' ': 1, '!': 1}
```

Hint: Yes, you did a problem very similar to this one on HW1.

```
In [ ]: count_characters("Torto ise!")
```

```
Out[ ]: {'T': 1, 'o': 2, 'r': 1, 't': 1, ' ': 1, 'i': 1, 's': 1, 'e': 1, '!': 1}
```

How many times does 't' appear in Emma? How about '!'?

How many different types of characters are in this dictionary?

```
In [ ]: # How many times does 't' appear in emma-full.txt.
        count_characters(s).get('t')
```

```
# How many different types of characters are in this dictionary?
len(count_characters(s))
```

Out[]: 82

Problem 2: Define `count_ngrams` in `HW2.py`

An `n`-gram is a sequence of `n` letters. For example, `bol` and `old` are the two 3-grams that occur in the string `bold`.

Write a function called `count_ngrams` that counts the number of times each `n`-gram occurs in a string, with `n` specified by the user and with default value `n = 1`. Your function should return the dictionary. You should be able to do this by making only a small modification to `count_characters`.

Example usage:

```
count_ngrams("tortoise", n = 2)

{'to': 2, 'or': 1, 'rt': 1, 'oi': 1, 'is': 1, 'se': 1} #
output
```

```
In [ ]: ## The code below was commented because the printed dictionary is too long 1
## PDF disbursement. Uncomment if you wish to see all dictionary entries.
# print(count_ngrams(s, 2))
```

```
## Print first 5 entries in the dictionary.
print(list(count_ngrams(s, 2).items())[:5])
```

```
[('CH', 58), ('HA', 56), ('AP', 55), ('PT', 55), ('TE', 55)]
```

How many different types of 2-grams are in this dictionary?

```
In [ ]: # write your answer here
len(count_ngrams(s, 2))
```

Out[]: 1236

Problem 3: Define `markov_text` in `HW2.py`

Now we are going to use our `n`-grams to generate some fake text according to a

Markov model. Here's how the Markov model of order `n` works:

A. Compute (`n + 1`)-gram occurrence frequencies

You have already done this in Problem 2!

B. Starting `n`-gram

The starting `n`-gram is the last `n` characters in the argument `seed`.

C. Generate Text

Now we generate text one character at a time. To do so:

1. Look at the most recent `n` characters in our generated text. Say that `n = 3` and the 3 most recent character are `the`.
2. We then look at our list of `n+1`-grams, and focus on grams whose first `n` characters match. Examples matching `the` include `them`, `the`, `thei`, and so on.
3. We pick a random one of these `n+1`-grams, weighted according to its number of occurrences.
4. The final character of this new `n+1` gram is our next letter.

For example, if there are 3 occurrences of `them`, 4 occurrences of `the`, and 1 occurrences of `thei` in the `n`-gram dictionary, then our next character is `m` with probability $3/8$, `[space]` with probability $1/2$, and `i` with probability $1/8$.

Remember: the *3rd*-order model requires you to compute *4*-grams.

What you should do

Write a function `markov_text` that generates synthetic text according to an `n`-th order Markov model. It should have the following arguments:

- `s`, the input string of real text.
- `n`, the order of the model.
- `length`, the size of the text to generate. Use a default value of 100.
- `seed`, the initial string that gets the Markov model started. I used `"Emma Woodhouse"` (the full name of the protagonist of the novel) as my `seed`, but any subset of `s` of length `n` or larger will work.

It should return a string with the length of `len(seed) + length`.

Demonstrate the output of your function for a couple different choices of the order `n`.

Expected Output

Here are a few examples of the output of this function. Because of randomness, your results won't look exactly like this, but they should be qualitatively similar.

```
markov_text(s, n = 2, length = 200, seed = "Emma Woodhouse")
```

```
Emma Woodhouse ne goo thimser. John mile sawas amintrought  
will on I kink you kno but every sh inat he fing as sat buty  
aft from the it. She cousency ined, yount; ate nambery quirdl  
diall yethery, yould hat earatte
```

```
markov_text(s, n = 4, length = 200, seed = "Emma Woodhouse")
```

```
Emma Woodhouse!"—Emma, as love, Kitty, only this  
person no infering ever, while, and tried very were no do be  
very friendly and into aid, Man's me to loudness of  
Harriet's. Harriet belonger opinion an
```

```
markov_text(s, n = 10, length = 200, seed = "Emma Woodhouse")
```

```
Emma Woodhouse's party could be acceptable to them, that if  
she ever were disposed to think of nothing but good. It will  
be an excellent charade remains, fit for any acquainted with  
the child was given up to them.
```

Notes and Hints

Hint: A good function for performing the random choice is the `choices()` function in the `random` module. You can use it like this:

```
import random
```

```
options = ["One", "Two", "Three"]  
weights = [1, 2, 3] # "Two" is twice as likely as "One", "Three"  
# three times as likely.
```

```
random.choices(options, weights)
```

```
['One'] # output
```

The first and second arguments must be lists of equal length. Note also that the return value is a list -- if you want the value *in* the list, you need to get it out via indexing.

Note: For grading purposes, the `options` should be the possible `n+1`-grams in the order of first appearance in the text. If you are working through the strings from beginning to end, you will not have issues with this, as dictionary keys are ordered. Please do NOT use `random.seed()` in your function -- the autograder code will do it. You are welcome to try it out in your notebook for reproducible results if you are interested.

Hint: The first thing your function should do is call `count_ngrams` above to generate the required dictionary. Then, handle the logic described above in the main loop.

```
In [ ]: print(markov_text(s, n = 10, length = 200, seed = "Emma Woodhouse"))
```

Emma Woodhouse's health, not merely no grammatical errors, but as a companion all this, and began talking of its effect upon her approach afraid,
So fatal to my suit before.

"I cannot imagine the

Problem 4

Using a `for`-loop, print the output of your function for `n` ranging from `1` to `10` (including 10).

Then, write down a few observations. How does the generated text depend on `n`? How does the time required to generate the text depend on `n`? Do your best to explain each observation.

What do you think could happen if you were to repeat this assignment but in unit of words and not in unit of characters? For example, 2-grams would indicate two words, and not two characters.

What heuristics would you consider adding to your model to improve its prediction performance?

1. How does the generated text depend on `n`?

Answer: As `n` increases, the text generation becomes more legible, the sentences have more correctly spelled words and better grammar.

2. How does the time required to generate the text depend on `n`?

Answer: As `n` increases, the text generation runtime increases.

3. What heuristics would you consider adding to your model to improve its prediction performance?

Answer: The size of `n` is a determinate of prediction quality, but if `n` is too large, the number of matches will decrease and possibly overfit the Markov method, making the generative text become more like a copy of the original text. To make the prediction performance better I would add a additional similar text sources to increase the number of ngram matches when `n` is large ($n > 15$).

```
In [ ]: random.seed(203)
num_gen_text = 10
## Print the n value used to generate text and the generated text.
for i in range(1, num_gen_text + 1):
    print("Generated test using n-gram n value of ", i, ": ")
    print(markov_text(s, n = i, length = 200, seed = "Emma Woodhouse"))
```

Generated test using n-gram n value of 1 :

Emma Woodhouse rathered, d po quld ised a med fo me d Theverough ind s beaidj ewhed metoote, ig ant ughu? thasherdarremmainsssetherer g bey r bed hede Mrm s litind ain armu al stre be s ney woontared d

Generated test using n-gram n value of 2 :

Emma Woodhouse; "I duch will ither?—Ind hazlect the we henjoymomere so foran d to bodhown Mish ot youled she but fors. He d's, doinigh miturell ble inte ot he musing at as an hing ston.

"I and not owe

Generated test using n-gram n value of 3 :

Emma Woodhousebrank Churrying a laughtley, were to laugh; the anot by so me they circumstateterns, no differrupted she worthy yet. Mr. Comtes not thould by pers rightle, "It was ston. Mrs. She garden,

Generated test using n-gram n value of 4 :

Emma Woodhouse, she for these could busy as was ther to get as much obligenc e, and sent to marry fright instance as than Isabella's calmness of ever, ex tremely well your from the other, or they were h

Generated test using n-gram n value of 5 :

Emma Woodhouse of a finery. Whenever has enough. But there were to marriage, my duty or want moment, supposing her family. Don't ask no more it as the pu lse before herself—"

"No, no, that I thing as

Generated test using n-gram n value of 6 :

Emma Woodhouse not advise?"

"But I," he answer.

Her tears running to stop him; and turn of the others."

Supper mad. He was pleasure grows older at your voice which had prevented to refrain on busin

Generated test using n-gram n value of 7 :

Emma Woodhouse's account of judging one or two handsome, elegant way, most b rother, in forestall Jane Fairfax. Mrs. Martin would be fit for poor Isabell a's coloured deeply as she ought to be lively ci

Generated test using n-gram n value of 8 :

Emma Woodhouse had been very earnestly, on the subject was to help Mr. Westo n's wishing to understood as my wish," said she, "we are determined to ruin the rest.—To Emma's wonder which I did their own

Generated test using n-gram n value of 9 :

Emma Woodhouse began rather a pleasure to me the most solicitation, Harriet, than with another.—But it would be much worse colds at the worst judge of hi s heart and time would be no harm in his profes

Generated test using n-gram n value of 10 :

Emma Woodhouse, one must laugh at my warmth—but, upon my honour, that it has nothing to do with William's wants, but it was very much obliged to be altog ether the uncertainty of being miserably behind

Problem 5

Try running your program with a different text!

You can

- find any movie script from <https://imsdb.com/> or a book by Shakespeare, Hemingway, Beowulf, O. Henry, A.A. Milne, etc from <https://www.gutenberg.org/>
- ctrl + a to select all text on the page
- copy paste into a new `.txt` file. let's call it `book.txt`.
- put `book.txt` in the same folder as `emma-full.txt`.
- run the following code to read the file into variable `s`.

```
with open('book.txt', 'r') as f:  
    s = f.read()
```

- run `markov_text` on `s` with appropriate parameters.

Show your output here. Which parameters did you pick and why? Do you see any difference from when you ran the program with Emma? How so?

1. Which parameters did you pick and why?

Answer: Set `n` to the length of `seed`. The length of `seed` is the maximum length that can be set for `n` and when `n` is large, the the text prediction quality increases.

2. Do you see any difference from when you ran the program with Emma?

Answer: The movie script from "Avengers: End Game" was used to create the (n+1)gram frequency dictionary for generating text. The movie script has more formatting, whitespace, and new lines than Jane Austen's novel *Emma*. Therefore, the generated text from "Avengers: End Game" has more generated formatting than the generated text using the novel *Emma*.

```
In [ ]: # book.txt contains the script for "Avengers: End Game". Read and assign to  
with open('book.txt', 'r') as f:  
    s = f.read()  
  
seed = "Iron Man"  
n_max = len(seed)  
print("Generated text using n-gram n value of ", n_max, ": ")  
print(markov_text(s, n = n_max, length = 200, seed = seed))
```

Generated test using n-gram n value of 8 :
Iron Man.

SNAP!

WHOOOSH! EVERYTHING COMES SMASHING THROUGH THE HORDE,
CARRYING THE PIECES TOGETHER.

SCOTT LANG (CONT'D)

I'd offer to cook you dinner thinking of giving me

```
# HW2
# Name: William Martinez
# Collaborators: None
# Date:4/21/24
```

```
import random
```

```
### Problem 1: Character Frequency in a string
```

```
def count_characters(s):
    """
    Return the frequency of each character in a string.
    Args:
        s: A string.
    Return:
        char_count: A dictionary of all characters in a string and their
        frequencies.
    """
    ## Initialization of Variables:
    char_count = dict()
    ## Character Frequency in a String:
    # If key not in dict, append it with a default value of 1, and if the key
    # already exist, increment the dict value by 1 for each occurrence.
    for i in range(len(s)):
        char = s[i]
        char_count[char] = char_count.get(char, 0) + 1
    return(char_count)
```

```
### Problem 2: N-Gram Frequency in a string
```

```
def count_ngrams(s, n = 1):
    """
    Return the frequency of each unique n-gram in a string.
    Args:
        s: A string.
        n: An integer. Corresponds to the length of an n-gram.
    Return:
        ngram_count: A dictionary of all characters in a string and their
        frequencies.
    """
    ## Initialization of Variables:
    ngram_count = dict()
    ## Character Frequency in a String:
    # Subtract the range of the for-loop by the value of n and add 1.
    # If ngram is not in dict, append it with a default value of 1. If the ngram
    # is in dict, increment the dict value by 1 for each occurrence.
    for i in range(len(s) - n + 1):
        ngram_str = s[i: i + n]
        ngram_count[ngram_str] = ngram_count.get(ngram_str, 0) + 1
    return(ngram_count)
```

```
### Problem 3: Generative Text Using the Markov Text Method
```

```
def markov_text(s, n, length = 100, seed = "Emma Woodhouse"):
```

```
"""
```

Return synthetically generated text using the Markov generative text method. The relative frequency of (n+1)-grams that match the last n-gram of the seed are used as weights in a random choice for the next character to be appended to the seed.

Args:

s: A string. This string will be used to create a frequency dictionary of (n+1)grams.

n: An integer. The ngram length.

length: An integer. The character length of the generative text.

seed: A string. A subset of the string, s. The start of the generative text.

Return:

gen_text: A string. Starts with seed and each character after is generated using the Markov method.

```
"""
```

```
## Error Catch:
```

```
# When n is greater than the seed length.
```

```
if (len(seed) < n):
```

```
    raise ValueError("Length of seed is less than size of n")
```

```
all_ngrams = count_ngrams(s,n+1) # A list of all ngrams that are .
```

```
gen_text = seed # start the generated text with seed.
```

```
## Generated Text Loop:
```

```
# Loop until gen_text is equal or greater than length.
```

```
while len(gen_text) < length:
```

```
    ## Initialization of Variables in Loop.
```

```
    # These variables will re-initialize with each loop
```

```
    last_ngram = gen_text[-n:] # Last ngram from gen_text
```

```
    options = list()
```

```
    weights = list()
```

```
    total = int()
```

```
    ## Match the begining n characters of the (n+1)gram with the characters
```

```
    ## of last ngram of the last generated text:
```

```
    for key, value in all_ngrams.items():
```

```
        if key[:-1] == last_ngram:
```

```
            options.append(key)
```

```
            weights.append(value)
```

```
            total += value
```

```
        else:
```

```
            continue
```

```
    # Sum of matched (n+1)grams frequencies.
```

```
    total = sum(weights)
```

```
    # Calculate relative frequencies (weights).
```

```
    for i in range(len(weights)):
```

```
        weights[i] = weights[i] / total
```

```
    # Randomly select a matched (n+1)gram based on its weight. keep the last
```

```
    # character.
```

```
    gen_ngram = random.choices(options, weights)[0][-1]
```

```
    # append the last character to the generated text
```

```
    gen_text += gen_ngram # append to generated text
```

```
return(gen_text)
```