HW2

January 19, 2022

1 Homework 2: Markov Models of Natural Language

1.0.1 Name: [Joanne Qiu]

1.0.2 Collaborators: [N/A]

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.

Submission instructions: - Submit `HW2.py` on Gradescope under "HW2 - Autograder". Do **NOT** change the file name. The grade you see is the grade you get for the accuracy portion of your code (so no surprises). The style and readability of your code will be checked by the reader aka human grader. - 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.

1.1 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.

[1]:
```
# This cell imports your functions defined in HW2.py

from HW2 import count_characters, count_ngrams, markov_text
```

1.2 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.

[2]:
```
with open('emma-full.txt', 'r') as f:
    s = f.read()
```

2 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 sequentually 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 should 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. Use `for i in range(len(s)):` instead.

2.0.1 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.

[3]:
```
# test your count_characters here
count_characters("Torto ise!")
```

[3]:
```
{'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?

[4]:
```
# write your answers here
print("'t' appears " + str(count_characters(s)["t"]) + " times in Emma")
print("'!' appears " + str(count_characters(s)["!"]) + " times in Emma")
print("There are " + str(len(count_characters(s))) + " types of characters in␣
 ↪this dictionary")
```

```
't' appears 58067 times in Emma
'!' appears 1063 times in Emma
There are 82 types of characters in this dictionary
```

3 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`.

3.0.1 Example usage:

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

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

```
[5]: # test your count_ngrams here
     count_ngrams("tortoise", n = 2)
```

```
[5]: {'to': 2, 'or': 1, 'rt': 1, 'oi': 1, 'is': 1, 'se': 1}
```

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

```
[6]: # write your answer here
     print("There are " + str(len(count_ngrams(s, n = 2))) + " types of 2-grams in␣
      ↪this dictionary")
```

```
There are 1236 types of 2-grams in this dictionary
```

4 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:

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

You have already done this in Exercise 2!

4.0.2 B. Pick a starting (n+1)-gram

The starting (n+1)-gram can be selected at random, or the user can specify it.

4.0.3 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 probabiliy 3/8, `[space]` with probability 1/2, and `i` with probability `1/8`.

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

4.1 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+1` or larger will work.

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

4.2 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
```

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

```
Emma Woodhouse!"-Emma, as love,            Kitty, only this person no infering ever, while, anc
```

```
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
```

4.3 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.

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.

[8]:
```
# test your markov_text here
markov_text("tortoise", n = 2, length = 12, seed = "rto")
```

```
The new 2 most character is 'to'
The options are 'r',' i'
The weights of each of the options are 0.5, 0.5
The updated synthetic text is rtor
The new 2 most character is 'or'
The options are 't'
The weights of each of the options are 1.0
The updated synthetic text is rtort


The new 2 most character is 'rt'
The options are 'o'
The weights of each of the options are 1.0
The updated synthetic text is rtorto


The new 2 most character is 'to'
The options are 'r',' i'
The weights of each of the options are 0.5, 0.5
The updated synthetic text is rtortor


The new 2 most character is 'or'
The options are 't'
The weights of each of the options are 1.0
The updated synthetic text is rtortort


The new 2 most character is 'rt'
The options are 'o'
The weights of each of the options are 1.0
The updated synthetic text is rtortorto


The new 2 most character is 'to'
The options are 'r',' i'
The weights of each of the options are 0.5, 0.5
The updated synthetic text is rtortortoi


The new 2 most character is 'oi'
The options are 's'
The weights of each of the options are 1.0
The updated synthetic text is rtortortois


The new 2 most character is 'is'
The options are 'e'
The weights of each of the options are 1.0
The updated synthetic text is rtortortoise
The new 2 most character is 'se'
The options are ''
The text doesn't have 3-gram that contains se as the foremost part

The final generated text is rtortortoise
```

5 Problem 4

Try out your function for varying values of `n`. 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.

(extra credit) 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.

(extra credit) What heuristics would you consider adding to your model to improve its prediction performance?

```
[7]: # write your observations and thoughts here
print("When n increases, the generated text becomes more logical.\
 This is because we takes more characters to evaluate the next character.\
 When taking a longer n-grams, the (n+1)-grams will possibly include phrases␣
 ↪that connect the previous words.\
 Thus, the generated text is more fluent when n is greater.\n")
print("When n increases, the time required to generate the text increases as␣
 ↪well.\
 Since n increases, the program needs to scan each character in the (n+1)-gram␣
 ↪to check if the n previous characters match the n most characters.\
 In this case, the program takes more time to find the options.\
 The length doesn't change, so the program still need to add the character one␣
 ↪by one,\
 thus the total time will definitely increase.\n")
print("I think if I change the unit of characters to unit of words, the types␣
 ↪of (n+1)-grams would significantly decrease,\
 because each word has a limit of match of phrases with other words.\
 As a result, it might take less time to find all the matched (n+1)-grams.\
 Secondly, since there is a limit of choices of phrases, the generated text␣
 ↪should be more fluent than using unit of characters.")
```

```
When n increases, the generated text becomes more logical. This is because we
takes more characters to evaluate the next character. When taking a longer
n-grams, the (n+1)-grams will possibly include phrases that connect the
previous words. Thus, the generated text is more fluent when n is greater.

When n increases, the time required to generate the text increases as well.
Since n increases, the program needs to scan each character in the (n+1)-gram to
```

```
check if the n previous characters match the n most characters. In this case,
the program takes more time to find the options. The length doesn't change, so
the program still need to add the character one by one, thus the total time will
definitely increase.

I think if I change the unit of characters to unit of words, the types of
(n+1)-grams would significantly decrease, because each word has a limit of match
of phrases with other words. As a result, it might take less time to find all
the matched (n+1)-grams. Secondly, since there is a limit of choices of phrases,
the generated text should be more fluent than using unit of characters.
```

[8]:
```python
# run your markov_text here
markov_text(s, n = 2, length = 200, seed = "Emma Woodhouse")
```

[8]:
```
'Emma Woodhouseke gue a re hummant wit."\n\nHe was itaidesty me: beento sor
iffirs. Not ission. I hould han useligin thad berseemem.-Wheircen, lif he
famposed ind hembeell comy ways-"\n\nThe he ining sh hicumbeen Bathin'
```

[9]:
```python
markov_text(s, n = 4, length = 200, seed = "Emma Woodhouse")
```

[9]:
```
'Emma Woodhouse world; for two. Elton with Mrs. Elton was good-bye, shewn the
marry dear you know, she many two be surprize an engaged the had no such a
mixture of the pair long more, Emma very great is worse-back a'
```

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

[10]:
```
'Emma Woodhouse described-it was not wanted to be out, and recur to old stories:
and he was really said only, "Miss Fairfax mistress of music, has not any thing
more was to be done?"\n\n"You would not help replying wi'
```

6 (Extra credit) 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 puppycat.txt. - put puppycat.txt in the same folder as emma-full.txt. - run the following code to read the file into variable `s`.

```python
with open('puppycat.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?

[11]:
```python
# run your new
with open('puppycat.txt', 'r') as f:
  s = f.read()
```

```
[12]: markov_text(s, n = 10, length = 200, seed = "Charles Darnay")
```

```
[12]: 'Charles Darnay had yesterday,\nthe blood spilled yesterday was, as usual, a
      cold one," observed Jacques Three, with his blue cap pointed under a patchwork
      counterpane, like a cold\nwind.\n\nHe gave his arm through the '
```

I pick the same numbers of grams and length as I used with Emma. The output appears to include more dramatic and thrilling words and depictions of actions while Emma's output represents some kinds of solioquy and emotion. I guess this difference may be a result of the authors' rhetoric strategy and writing style and the content of the book. Jane Austen's Emma embraces topics of family, friendship, and romance while Charles Dickens' A Tale of Two Cities is about people's resurrection and transformation.