Homework 2: Markov Models of Natural Language

Name: Yuhui Wang

Collaborators: [Your collaborators]

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

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 [2]:
```
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 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 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 [3]:
```
def count_characters(s):
    d = {}
    for i in range(len(s)):
        if s[i] not in d:
            d[s[i]] = 1
        else:
            d[s[i]] += 1

    return d
```

In [4]:
```
# test your count_characters here
count_characters("Torto ise!")
```

Out[4]:
```
{'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 [5]:
```
# write your answers here
dict_emma = count_characters(s)
print("'t' appears for " + str(dict_emma['t']) + " times")
print("'!' appears for " + str(dict_emma['!']) + " times")

print("There are " + str(len(dict_emma)) + " different types of characters")
```

```
't' appears for 58067 times
'!' appears for 1063 times
There are 82 different types of characters
```

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 [6]:
```
def count_ngrams(string, n = 1):
    counts = {}
    for i in range(len(string) - n + 1):
        ngram = string[i:i+n]
        if ngram in counts:
            counts[ngram] += 1
        else:
            counts[ngram] = 1
    return counts
```

In [7]:
```
# test your count_ngrams here
print(count_ngrams("tortoise", n = 2))
print(count_ngrams("tortoise"))
```

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

```
{'t': 2, 'o': 2, 'r': 1, 'i': 1, 's': 1, 'e': 1}
```

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

In [8]:
```
# write your answer here
dic = count_ngrams(s, n = 2)
print("There are " + str(len(dic)) + " different types of 2-grams")
```

```
There are 1236 different types of 2-grams
```

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 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.

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 quirld
   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 appeareance 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 [9]:
```
def markov_text(s, n, length=100, seed=""):
    ngram_counts = count_ngrams(s, n+1)
    generated_text = seed

    while len(generated_text) < len(seed) + length:
        current_ngram = generated_text[-n:]
        possible_next_chars = [key[n] for key in ngram_counts if key.startsw
        weights = [ngram_counts[current_ngram + char] for char in possible_n
        next_char = random.choices(possible_next_chars, weights)[0]
        generated_text += next_char

    return generated_text
```

In [10]:
```
# test your markov_text here
import random
markov_text(s, n = 2, length = 200, seed = "Emma Woodhouse")
```

Out[10]:
```
'Emma Woodhouse, youbt.\n\n“Oh! He a re ut whould yousinevely crity.”\n\n“I
hichis comperhany id, becte exces.\n\n“But saine practuad the not ithery su
p, spery to eve he ind I died no hopleres sompayse, I hown, soo yourch'
```

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?

In [12]:
```
# write your observations and thoughts here

# From the results, we can see that the generated texts become more coherent
# contextually relevant when n increases.
# That is because the higher order model will consider a longer history char
```

```
# That is because the higher order model will consider a longer history chai
# When n is small, the generated text will appear more random and less meani

# The time required will increase along with the
# increase of n, as a higher order model involves more computations.

# Using word-level n-grams will make the generated texts more
# coherent and meaningful, as it will capture higher-level
# semantics and syntactic structures compared to character-level n-grams.

# Feature engineering, for example, adding contextual
# features and linguistic knowledge, will help the model
# understand the previous text and predict more accurately
```

In [13]:
```
# run your markov_text here
for n in range(1, 11):
    print(f"Markov text with n={n}:")
    print(markov_text(s, n=n, length=200, seed="Emma Woodhouse"))
    print("\n")
```

```
Markov text with n=1:
Emma Woodhousey. ay m dldhas try. uca ie mind rofare ps s ct ped ag w Mrerat
mesuanca bldevot hichurerry, sereathat or ckercie acabuther unne nge weas. a
ithte; ptor aspte y Kndoforint vereraies mpyotsit heving he t


Markov text with n=2:

Emma Woodhouse.—She Emma con diest. Youly purried ges, to in a wholly lon ar
d te te couressurse, he thea-bat ito Mrselto John, sen motheir a whand equie
ttle.—But ateng a caund friderapeat have shour ton. She hille


Markov text with n=3:
Emma Woodhouse a be unce, rapture I me occasisteptabliged when, and as state
s herward taledge as in angere. If Mr. West put outs, (glad by her. Mrs. Wes
ting to him, my desenseeing to been inted I man at not pres—cu


Markov text with n=4:
Emma Woodhouse made.

The every smoothink outcry. After have carry on start, this have been the wh
ened, I do the quests bears, the subjects interest, then Mrs. Cole of paid,
a regular come and of it! I say, by read


Markov text with n=5:
Emma Woodhouse, and call on her in general apples and might hold me if Jane;
and she long on freak so sorry, honest repetitionable; and, and coarse not s
ee young to do. That, but it will not much you should only yo


Markov text with n=6:
Emma Woodhouse, would I ever parent, if I could not find a pretty letter if
it were he made to herself—and a very well enough to praise of his own will
done this apologies; and stood out of Harriet; and to get inti


Markov text with n=7:
Emma Woodhouse. I merely employed in tranquillised and construction for the
subject on succeeded in the arrangement made her at once.—Something beyond w
hat we never shall. And Mr. John Knightley. I will not liking


Markov text with n=8:
Emma Woodhouse had not be consent before, are unjust to Hartfield, perhaps,
or a little just—every imagination. Harriet was puzzling over-trimmed; I hav
e no doubt: accomplished: and every thing he was quite alone.
```

```
e no doubt, accomplished, and every thing he was quite alone.

Markov text with n=9:
Emma Woodhouse! who can say how perfectly remembrance."

"I am,"—she answer, but we shall gradually done full justice—only we do now,
it is clear; the state, I assure you, Mr. Knightley, what a perfectly dry. C
ome

Markov text with n=10:
Emma Woodhouse and Emma, at last, when Mrs. Weston kindly and persuaded that
you would be wanted, and his father's being gone out to put the horse were u
seable; but he was sufferings acted as a cure of every thing
```

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?

In [14]:
```
with open('book.txt', 'r') as f:
    s = f.read()
```

In [15]:
```
# run your new code
markov_text(s, 12, 800, seed="Nerve enough")
```

Out[15]:
```
'Nerve enough\nThis ebook is for the use of anyone anywhere in the United St
ates with eBooks not protected by U.S. copyright law. Redistributing, perfor
ming, displayed their tiny square outlines in regular patterns around the gr
ound. There was no longer any rush of wind or roar of motor; nothing but a f
ew tatters of silk and several shroud lines were securely held by the rudde
r.\nO'Connell's eyes glinted.\n"'Tis not a bad idea at all," he admitted, an
d looked at each other. Determination was imprinted in the lines of both cou
ntenances, and together they squirmed to their feet in that cramped compartm
ent.\nThe motor labored on, and both men thrust feet out straight, and moved
shoulders tentatively, as if to drive away any incipient stiffness that migh
t hinder action in that one swift leap into space.\nBut ju'
```

In [16]:
```
# I choose n = 12 length = 800, and seed = "Nerve enough".
# This is because "Nerve enough" is the book name,
# and n = 12 is the maximum length of the book name.
# Choosing the max n can help the model understand the text best.

# This generated text followed the original text more,
# as I set the parameter higher to involve more computation
# and learn more from previous text.
```