Created by: Jeffrey Niu and Fadime Bekmambetova, in collaboration with Christopher Lucasius and Sourojeet Chakraborty

ESC180 LAB 4

Natural Language Processing: N-grams

Due: November 12, 9:00 am

INSTRUCTIONS

1. `git pull` to access the provided module, `utilities.py`. You will find it under `<utorid>/lab4/`. **Do not modify `utilities.py`**.
2. Your submission **MUST** be stored in the git repository under **`<utorid>/lab4/` and named `lab4.py`.**
3. Only import one file, utilities.py.
4. When submitting, do not leave the block under "if __name__ == '__main__'" blank. That is, **THE FILE YOU SUBMIT should run and be importable without any errors.**
5. To test (4) completely, you may create a new .py file, import your lab4, and run your functions from there to ensure your module runs smoothly.

BACKGROUND

The Turing Test is a hypothetical test that would determine whether artificial intelligence is indistinguishable from human intelligence. In lieu of the Turing Test, we propose the Henry test, where we test whether Professor Henry can distinguish between AI and humans. In this lab, you will implement one of the tests: whether Professor Henry can detect whether a text is machine generated or written by a human.

In your test, you will use an n-gram model to generate an original text inspired by existing texts. The main idea behind the n-gram model is to predict the next word in a text based on the previous n words. To build your model, you will need to understand the following terminology:

- Corpus: A collection of texts. Generally, researchers curate corpora using texts with a common theme. For instance, there are corpuses of Reddit comments and Shakespearean plays.
- Lexicon: A vocabulary of words and punctuation in a corpus. In the corpus, capitalization in words does not matter. The punctuation is described in the `parse_story` function.
- Token: A single element of the lexicon.
- N-gram: A series of N tokens that appear consecutively in the corpus, where N is the number of these consecutive tokens in the N-gram (examples follow).

Consider the short corpus below:

> Once upon a time there was a child. The child is sad today. However, the child will go out to play, and the child can not be sad anymore.

Examples of 2-grams include: (once, upon), (upon, a), (a, time), … ,(sad, anymore), (anymore, .)
Examples of 3-grams include: (once, upon, a), (., the, child), (to, play, ,)[1], (sad, anymore, .)

Your text generation will consider the probability of a specific token following an N-gram and pick a word/punctuation based on those probabilities. To generate the probabilities, you will parse the corpus and count the different occurrences of a single word after an N-gram. Returning to the short corpus using 2-grams, the 2-gram (once, upon) is followed by "a". The 2-gram (the, child) is followed once by "is", once by "will", and once by "can". Thus, the next word after "once upon" will be "a" with 100% probability. The next word after "the child" will be one of "is", "will", and "can", each with one-thirds probability.

Implement the functions below to complete the Henry test.

PROVIDED TEXTS FOR GENERATING CORPORA

The following texts are provided:

- 45506.txt: *Flatland: A Romance of Many Dimensions* by Edwin Abbott Abbott.
  Downloaded from: https://www.gutenberg.org
- 308.txt: *Three Men in a Boat* by Jerome K. Jerome.
  Downloaded from: https://www.gutenberg.org
- 18155.txt: *The Story of the Three Little Pigs* by unknown author.
  Downloaded from: https://www.gutenberg.org

The files have been modified to only include the main text of the literary work – not chapter headers, website information, or other data that could corrupt N-gram model generation. Try generating text based on these works.

Some other small files for testing individual functions are also provided (see details below).

PROVIDED CODE

Some useful functions and constants are provided in utilities.py. **Do not** modify the contents of this file.

TASKS

Complete the following functions as well as docstrings.

`parse_story(file_name)`

[1] Here, the third comma represents the comma punctuation

Returns an ordered list of words with bad characters processed (removed) from the text in the file given by `file_name`.

Valid punctuation is provided as a list of characters, `VALID_PUNCTUATION`, in utilities.py. Consider those characters as individual tokens when parsing the text. Bad characters are provided in the list `BAD_CHARS` found in utilities.py. These are not considered words. Discard them when parsing the text.

Bad characters and valid punctuation may or may not be separated by spaces. For example, a file with text "They said "Hello."" should be parsed into ['They', 'said', 'Hello', '.']. In this case, the period was not separated by a space and yet should be counted as a separate token (not part of "Hello."). Text "Why?!" would be parsed into ['Why', '?', '!'].

White space should not be included in the list of tokens neither as a separate token nor as part of a token.

Input format

`file_name`: str giving the name of the file

Sample inputs and outputs

```
>>> parse_story('test_text_parsing.txt')
['the', 'code', 'should', 'handle', 'correctly', 'the',
'following', ':', 'white', 'space', '.', 'sequences', 'of',
'punctuation', 'marks', '?', '!', '!', 'periods', 'with',
'or', 'without', 'spaces', ':', 'a', '.', '.', 'a', '.', 'a',
"don't", 'worry', 'about', 'numbers', 'like', '1', '.', '5',
'remove', 'capitalization']
```

get_prob_from_count(counts)

Return a list of probabilities derived from counts. Counts is a list of counts of occurrences of a token after the previous n-gram. You should not round the probabilities.

Input format

counts: a list of positive integer counts

Sample inputs

```
>>> get_prob_from_count([10, 20, 40, 30])
[0.1, 0.2, 0.4, 0.3]
```

build_ngram_counts(words, n)

Return a dictionary of N-grams (where N=n) and the counts of the words that follow the N-gram. The key of the dictionary will be the N-gram in a tuple. The corresponding value will be a list containing two lists. The first list contains the words and the second list contains the corresponding counts.

Input format

words: a list of words obtained from `parse_story`

n: the size of the N-gram

Sample inputs

```
>>> words = [‘the’, ‘child’, ‘will’, ‘go’, ‘out’, ‘to’, ‘play’, ‘,’, ‘and’, ‘the’, ‘child’, ‘can’, ‘not’, ‘be’, ‘sad’, ‘anymore’, ‘.’]
>>> build_ngram_counts(words, 2)
{
        (‘the’, ‘child’): [[‘will’, ‘can’], [1, 1]],
        (‘child’, ‘will’): [[‘go’], [1]],
        (‘will’, ‘go’): [[‘out’], [1]],
        (‘go’, out’): [[‘to’], [1]],
        (‘out’, ‘to’): [[‘play’], [1]],
        (‘to’, ‘play’): [[‘,’], [1]],
        (‘play’, ‘,’): [[‘and’], [1]],
        (‘,’, ‘and’): [[‘the’], [1]],
        (‘and’, ‘the’): [[‘child’], [1]],
        (‘child’, ‘can’): [[‘not’], [1]],
        (‘can’, ‘not’): [[‘be’], [1]],
        (‘not’, ‘be’): [[‘sad’], [1]],
        (‘be’, ‘sad’): [[‘anymore’], [1]],
        (‘sad’, ‘anymore’): [[‘.’], [1]]
}
```

Note:

- The key-value pairs do not have to appear in the exact order above because Python dictionaries are unordered.
- The words and counts do not have to appear in the exact order above (though it may be helpful to store the counts in sorted order), but the words and counts indices must match. e.g. the value [[‘at’, ‘be’], [4, 2]] means that “at” appears four times, and “be” appears twice.

`prune_ngram_counts(counts, prune_len)`

Return a dictionary of N-grams and counts of words with lower frequency (i.e. occurring less often) words removed. You will prune the words based on their counts, keeping the `prune_len` highest frequency words. In case of a tie (for example, if `prune_len` was 5

and the 5th and 6th most frequent words had the same frequency), then keep all words that are in the tie (e.g. keep both the 5th and 6th words).

Input format

counts: a dictionary of N-grams and word counts formatted according to the output of build_ngram_counts

prune_len: the number of highest frequency words to keep (potentially more if ties occur)

Sample inputs

```
>>> ngram_counts = {
      (‘i’, ‘love’): [[‘js’, ‘py3’, ‘c’, ‘no’], [20, 20, 10, 2]],
      (‘u’, ‘r’): [[‘cool’, ‘nice’, ‘lit’, 'kind’], [8, 7, 5, 5]],
      ('toronto’, ‘is’): [[‘six’, ‘drake’], [2, 3]]
}
>>> prune_ngram_counts(ngram_counts, 3)
{
      (‘i’, ‘love’): [[‘js’, ‘py3’, ‘c’], [20, 20, 10]],
      (‘u’, ‘r’): [[‘cool’, ‘nice’, ‘lit’, 'kind’], [8, 7, 5, 5]],
      ('toronto’, ‘is’): [[‘six’, ‘drake’], [2, 3]]
}
```

probify_ngram_counts(counts)

Take a dictionary of N-grams and counts and convert the counts to probabilities. The probability of each word is defined as the observed count divided by the total count of all words.

Input format

counts: a dictionary of N-grams and word counts formatted according to the output of prune_ngram_counts

Sample inputs

```
>>> ngram_counts = {
      (‘i’, ‘love’): [[‘js’, ‘py3’, ‘c’], [20, 20, 10]],
      (‘u’, ‘r’): [[‘cool’, ‘nice’, ‘lit’, 'kind’], [8, 7, 5, 5]],
      ('toronto’, ‘is’): [[‘six’, ‘drake’], [2, 3]]
}
>>> probify_ngram_counts(ngram_counts)
{
      (‘i’, ‘love’): [[‘js’, ‘py3’, ‘c’], [0.4, 0.4, 0.2]],
      (‘u’, ‘r’): [[‘cool’, ‘nice’, ‘lit’, 'kind’], [0.32, 0.28,
        0.2, 0.2]],
      ('toronto’, ‘is’): [[‘six’, ‘drake’], [0.4, 0.6]]
}
```

build_ngram_model(words, n)

Create and return a dictionary of the format given above in probify_ngram_counts. This dictionary is your final model that will be used to auto-generate text.

For your final model, keep the 15 most likely words that follow an N-gram. Moreover, for each N-gram, the corresponding next words should appear in descending order of probability.

Input format

words: a list of words/punctuation obtained from `parse_story`

n: the size of N in the N-grams.

Sample inputs

```
>>> words = ['the', 'child', 'will', 'the', 'child', 'can', 'the',
'child', 'will', 'the', 'child', 'may', 'go', 'home', '.']
>>> build_ngram_model(words, 2)
{
      ('the', 'child'): [['will', 'can', 'may'], [0.5, 0.25,
        0.25]],
      ('child', 'will'): [['the'], [1.0]],
      ('will', 'the'): [['child'], [1.0]],
      ('child', 'can'): [['the'], [1.0]],
      ('can', 'the'): [['child'], [1.0]],
      ('child', 'may'): [['go'], [1.0]],
      ('may', 'go'): [['home'], [1.0]],
      ('go', 'home'): [['.'], [1.0]]
}
```

`gen_bot_list(ngram_model, seed, num_tokens=0)`

Returns a randomly generated list of tokens (strings) that starts with the N tokens in `seed`, selecting all subsequent tokens using `gen_next_token`. The list ends when any of the following happens:

- List contains `num_tokens` tokens, including repetitions. In case seed is longer than `num_tokens`, the returned list should contain the first `num_tokens` tokens of the seed.
- Any of the assumptions of `gen_next_token` is violated. I.e. if either an N-gram is not in the model, or if an N-gram has no tokens that follow it.

Input format

`ngram_model`: the format of this input is the same as the format of the output of `build_ngram_model`. However, your code for this function should be able to handle cases where not all N-grams are present in the dictionary.

`seed`: a tuple of strings representing the first N tokens in the list.

`num_tokens`: a positive int representing the largest number of tokens to be put in the list

Assume that you will never have a test case where N in the ngram_model and N in the seed are mismatched. For example, if ngram_model contains 4-grams, `seed` will be a 4-gram as well.

Sample inputs and outputs

```
>>> ngram_model = {('the', 'child'): [['will', 'can','may'], [0.5, 0.25, 0.25]], \
                   ('child', 'will'): [['the'], [1.0]], \
                   ('will', 'the'): [['child'], [1.0]], \
                   ('child', 'can'): [['the'], [1.0]], \
                   ('can', 'the'): [['child'], [1.0]], \
                   ('child', 'may'): [['go'], [1.0]], \
                   ('may', 'go'): [['home'], [1.0]], \
                   ('go', 'home'): [['.'], [1.0]] \
                   }
>>> random.seed(10)
>>> gen_bot_list(ngram_model, ('hello', 'world'))
[]
>>> gen_bot_list(ngram_model, ('hello', 'world'), 5)
['hello', 'world']
>>> gen_bot_list(ngram_model, ('the', 'child'), 5)
['the', 'child', 'can']
```

Note that the removal of the crossed out `('child', 'can')` 2-gram is the reason for the termination.

```
>>> gen_bot_list(ngram_model, ('the', 'child'), 5)
['the', 'child', 'will', 'the', 'child']
```

gen_bot_text(token_list, bad_author)

If `bad_author` is `True`, returns the string containing all tokens in `token_list`, separated by a space. Otherwise, returns this string of text, respecting the following grammar rules:

- There are no spaces before tokens found in `VALID_PUNCTUATION`

- A sentence cannot start with a lower-case letter. The array in utilities.py called `END_OF_SENTENCE_PUNCTUATION` tells you how to determine that the previous sentence has finished. (Recall that strings are case-sensitive).
- Words in `ALWAYS_CAPITALIZE` should start with a capital letter

Sample inputs and outputs

```
>>> token_list = ['this', 'is', 'a', 'string', 'of', 'text',
'.', 'which', 'needs', 'to', 'be', 'created', '.']
>>> gen_bot_text(token_list, False)
'This is a string of text. Which needs to be created.'
>>> token_list = parse_story("308.txt")
>>> text = gen_bot_text(token_list, False)
>>> write_story('test_gen_bot_text_student.txt', text, 'Three
Men in a Boat', 'Jerome K. Jerome', 'Jerome K. Jerome', 1889)
```

Our output to this code is saved in test_gen_bot_text.txt. You can run the following command in the terminal to compare files:

```
diff -c test_gen_bot_text_student.txt test_gen_bot_text.txt
```

If there is no output produced, the files are the same. Make sure that the two files match.

`write_story(file_name, text, title, student_name, author, year)`

Writes the text to the file with name `file_name`. The file should begin with a title "page", starting with 10 newline chars and ending with 17 newline chars, and having the following content:

`title`: `year`, UNLEASHED

`student_name`, inspired by `author`

Copyright year published (`year`), publisher: EngSci press

For exact formatting and spacing between characters refer to `test_write_story.txt` or `test_gen_bot_text.txt`.

You can comfortably read about 90 characters on a line (excluding new line character from the count). Place the largest number of words you can fit on the 90-character line. Do not break words or any other sequences of characters that were not initially separated by white space. Do not begin or end a line in a space character.

When EngSci press prints the text, it can fit exactly 30 lines on a page. Place a page number as the last line of each page (starting with 1 and the title page is not numbered). The line before the page number should be blank. The last page should also display the page number on the 30th line (no new line character at the end of the file)

Every 12 pages is a new chapter. Write chapter headings followed by two newline characters:

CHAPTER #\n\n

starting with CHAPTER 1\n\n

Input format

- `file_name`: string giving the name of the output file
- `text`: string of text to be written to the output file
- `title`: string giving the title of the text
- `student_name`: string with your name or your creative pen name
- `author`: string with the name of the author of the original text
- `year`: integer giving the year of publication (2019 if you submit the code on time)

You can assume that `text` has no white space except single space characters delimiting groups of characters (words or words with punctuation marks). Also, assume that chapter headings and lines on the title page will never go over the 90-character budget. Moreover, assume that you will never have a sequence of characters without spaces that is longer than 90 characters.

```
>>> text = ' '.join(parse_story('308.txt'))
>>> write_story('test_write_story_student.txt', text, 'Three Men in a Boat', 'Jerome K. Jerome', 'Jerome K. Jerome', 1889)
```

This will produce a file 'test_write_story_student.txt'. Our output is in 'test_write_story.txt'. Running the following command in the terminal should produce no output if the files are the same.

```
diff -c test_write_story.txt test_write_story_student.txt
```

The two files need to match exactly to ensure correctness of your solution.

SUBMISSION INSTRUCTIONS

- No late submissions will be accepted. The submission closest to and before the indicated due date will be marked.
- Your code must run on the Linux ECF machines with the specified addition in point using the command python3 lab4.py
- DO NOT ASSUME your code will run after minor modifications – TEST IT as this is the only way to be certain.
- Remarks will only be granted for code that has already been submitted prior; i.e. no chances for resubmissions will be granted.

- Review Git_instructions.pdf and the git commands worksheet solutions if you are unsure whether your submission was successful.