CS 505 Homework 03: N-Gram Modelling

Due Thursday 10/5 at midnight (1 minute after 11:59 pm) in Gradescope (with a grace period of 6 hours)

You may submit the homework up to 24 hours late (with the same grace period) for a penalty of 10%.

All homeworks will be scored with a maximum of 100 points; point values are given for individual problems, and if parts of problems do not have point values given, they will be counted equally toward the total for that problem.

Note: I strongly recommend you work in **Google Colab** (the free version) to complete homeworks in this class; in addition to (probably) being faster than your laptop, all the necessary libraries will already be available to you, and you don't have to hassle with `conda`, `pip`, etc. and resolving problems when the install doesn't work. But it is up to you! You should go through the necessary tutorials listed on the web site concerning Colab and storing files on a Google Drive. And of course, Dr. Google is always ready to help you resolve your problems.

I will post a "walk-through" video ASAP on my Youtube Channel (https://www.youtube.com/channel/UCfSqNB0yh99yuG4p4nzjPOA).

Submission Instructions

You must complete the homework by editing **this notebook** and submitting the following two files in Gradescope by the due date and time:

- A file `HW03.ipynb` (be sure to select `Kernel -> Restart and Run All` before you submit, to make sure everything works); and
- A file `HW03.pdf` created from the previous.

  For best results obtaining a clean PDF file on the Mac, select `File -> Print Review` from the Jupyter window, then choose `File-> Print` in your browser and then `Save as PDF`. Something similar should be possible on a Windows machine -- just make sure it is readable and no cell contents have been cut off. Make it easy to grade!

The date and time of your submission is the last file you submitted, so if your IPYNB file is submitted on time, but your PDF is late, then your submission is late.

In [1]:

```
### FIRST VERSION
```

Collaborators (5 pts)

Describe briefly but precisely

1. Any persons you discussed this homework with and the nature of the discussion;
2. Any online resources you consulted and what information you got from those resources; and
3. Any AI agents (such as chatGPT or CoPilot) or other applications you used to complete the homework, and the nature of the help you received.

A few brief sentences is all that I am looking for here.

```
<Your answer here>

1.
2. I looked at lecture notes from the course to review the materials for perplexity and probability.
3. I searched ChatGPT to get an example of adding with log function and putting it into math.exp when calculating perplexity when the float is very low.
```

Overview

etc.

In [ ]:

```
import math
import numpy as np
from numpy.random import shuffle, seed, choice
import nltk
from tqdm import tqdm
from collections import defaultdict
import random

# First time you will need to download the corpus:
# Run the following and download the book collection

from nltk.corpus import brown
nltk.download('brown')
```

```
[nltk_data] Downloading package brown to /root/nltk_data...
[nltk_data]   Package brown is already up-to-date!
```

Out[64]:

```
True
```

Problem One: Bag of N-Grams (30 pts)

A BOW is a language modelling technique (also called a Term Frequency Vector) which creates a frequency distribution for a set of tokens -- or unigrams! Extending this idea a bit, we can also create a Bag of N-Grams, which is a frequency distribution for a set of N-grams for some N. If we divide the frequency by the number of N-grams, we have a probability distribution, such as we showed for the exciting text about John and Mary in Lecture 5.

For this homework, we are going to create such Bag of N-Gram models for N = 1, 2, 3, & 4, for the sentences in `brown.sents()` . We will evaluate them using a test set, and then in the second part of the homework, we shall use them to generate sentences.

Note 1: We do not want to do the same low-level transformations in this project as we did in HW 02. We will keep the capitalization, punctuation, and words in all their various forms. There are some strange things in `brown.sents()` , such as double semicolons, and bad sentence segmentation, but we will assume the processing of the texts into `brown.sents()` was consistent, and we will see what our model makes of this data.

Note 2: Since `brown.sents()` contains punctuation marks as well as words, we shall use the term **tokens** for the strings stored in the sentence lists.

Part A: Randomize the list of sentences and split into training and testing sets

We will use `brown.sents()` (a list of list of tokens) as the basis of our N-gram models. The list `brown.words()` is simply the concatenation of all these lists of tokens.

We will shuffle the list into a random order, but using a seed value so that the order of the random shuffle is the same each time.

1. Read about `numpy.random.seed` and `numpy.random.shuffle` .
2. Set the seed to `0` and shuffle the list: you can't shuffle `brown.sents()` and because `numpy.random.shuffle` modifies the list **in place**, to avoid reshuffling:

   - Convert **brown.sents()** to a list and assign to a new variable **sentences** and then
   - **Copy sentences** to a new variable **shuffled_sentences** and then
   - Shuffle that list.

In this way, you will have the original list, and a randomized list, but because of `seed(0)` it will be in the same order every time you run your code (and when we grade it).

3. Then split `shuffled_sentences` into sets `training_sents` (first 99.9% of the sentences) and `testing_sents` (last 0.1%).
4. Print out the length of the training and testing sets.
5. Print out the first sentence in each of these sets.

In 4 and 5, label the outputs so we know which is which. (Always make outputs easy to understand!)

NOTE: The terms "training set" and "testing set" are very standard, even though we store these in lists (it is possible that there are duplicate sentences).

In [ ]:
```
# First, shuffle the set of sentences

seed(0)

sentences = list(brown.sents())
shuffled_sentences = list(sentences[:])
shuffle(shuffled_sentences)





# your code here

# split into training and testing sets
length_train = int(len(shuffled_sentences) * .999)
training_sents = shuffled_sentences[:length_train]
testing_sents = shuffled_sentences[length_train:]

# your code here
print(f"Length training set: {len(training_sents)}.")
print(f"Length of testing set: {len(testing_sents)}. \n")

print(f"Start of training set: \n {training_sents[0]}) \n")
print(f"Start of testing set: \n {testing_sents[0]}")
```

```
Length training set: 57282.
Length of testing set: 58.

Start of training set: 
 ['Muscle', 'weakness', 'did', 'not', 'improve', ',', 'and', 'the', '
patient', 'needed', 'first', 'a', 'cane', ',', 'then', 'crutches', '.
'])

Start of testing set: 
 ['It', 'is', 'at', 'least', 'as', 'important', 'as', 'the', 'more',
'dramatic', 'attempts', 'to', 'break', 'down', 'barriers', 'of', 'ine
quality', 'in', 'the', 'South', '.']
```

Part B

Now, you must add the beginning `<s>` and ending `</s>` markers to each sentence in both the training and testing lists. Do not make any other changes to the sentences -- you will see that punctuation has been left in, such as periods at the end of sentences. Again, we will see what our models make of this data set.

Print out the first sentence in each of the training and testing sets to check that all is well.

In [ ]:

```
# put `<s>` at beginning and `</s>` at end of all sentences.

def bracket_sentence(sent):
    for x in sent:
        x.insert(0, "<s>")
        x.append("</s>")
    return sent

training_sents = bracket_sentence(training_sents)
testing_sents = bracket_sentence(testing_sents)


# your code here
print(f"Start of training set: \n {training_sents[0]} \n")
print(f"Start of testing set: \n {testing_sents[0]}")
```

```
Start of training set:
 ['<s>', 'Muscle', 'weakness', 'did', 'not', 'improve', ',', 'and', '
the', 'patient', 'needed', 'first', 'a', 'cane', ',', 'then', 'crutch
es', '.', '</s>']

Start of testing set:
 ['<s>', 'It', 'is', 'at', 'least', 'as', 'important', 'as', 'the', '
more', 'dramatic', 'attempts', 'to', 'break', 'down', 'barriers', 'of
', 'inequality', 'in', 'the', 'South', '.', '</s>']
```

Part C

Complete the following template for a function to extract N-grams from one sentence, and test it for N = 1,2,3,4 for the first sentences in the training set.

In [ ]:

```
# Return a list of the N-grams for all sentences s

# Store all N-grams as tuples, so that a unigram is (w,), a bigram is

def get_Ngrams_for_sentence(N,s):
    answer = []
    for x in range(N):
      for y in range(0, len(s) - N + 1):
        answer.append(tuple(s[:N]))
        s = s[1:]
    return answer                            # your code here



# test on first sentence in the training set
sentence_one = get_Ngrams_for_sentence(1, training_sents[0])
sentence_two = get_Ngrams_for_sentence(2, training_sents[0])
sentence_three = get_Ngrams_for_sentence(3, training_sents[0])
sentence_four = get_Ngrams_for_sentence(4, training_sents[0])
# your code here

print(sentence_one)
print()
print(sentence_two)
print()
print(sentence_three)
print()
print(sentence_four)
```

```
[('<s>',), ('Muscle',), ('weakness',), ('did',), ('not',), ('improve'
,), (',',), ('and',), ('the',), ('patient',), ('needed',), ('first',)
, ('a',), ('cane',), (',',), ('then',), ('crutches',), ('.',), ('</s>
',)]

[('<s>', 'Muscle'), ('Muscle', 'weakness'), ('weakness', 'did'), ('di
d', 'not'), ('not', 'improve'), ('improve', ','), (',', 'and'), ('and
', 'the'), ('the', 'patient'), ('patient', 'needed'), ('needed', 'fir
st'), ('first', 'a'), ('a', 'cane'), ('cane', ','), (',', 'then'), ('
then', 'crutches'), ('crutches', '.'), ('.', '</s>')]

[('<s>', 'Muscle', 'weakness'), ('Muscle', 'weakness', 'did'), ('weak
ness', 'did', 'not'), ('did', 'not', 'improve'), ('not', 'improve', '
,'), ('improve', ',', 'and'), (',', 'and', 'the'), ('and', 'the', 'pa
tient'), ('the', 'patient', 'needed'), ('patient', 'needed', 'first')
, ('needed', 'first', 'a'), ('first', 'a', 'cane'), ('a', 'cane', ','
), ('cane', ',', 'then'), (',', 'then', 'crutches'), ('then', 'crutch
es', '.'), ('crutches', '.', '</s>')]

[('<s>', 'Muscle', 'weakness', 'did'), ('Muscle', 'weakness', 'did',
'not'), ('weakness', 'did', 'not', 'improve'), ('did', 'not', 'improv
e', ','), ('not', 'improve', ',', 'and'), ('improve', ',', 'and', 'th
e'), (',', 'and', 'the', 'patient'), ('and', 'the', 'patient', 'neede
d'), ('the', 'patient', 'needed', 'first'), ('patient', 'needed', 'fi
rst', 'a'), ('needed', 'first', 'a', 'cane'), ('first', 'a', 'cane',
','), ('a', 'cane', ',', 'then'), ('cane', ',', 'then', 'crutches'),
(',', 'then', 'crutches', '.'), ('then', 'crutches', '.', '</s>')]
```

Part D

Now create lists of N-grams for all the sentences in your training set (NOT the testing set). Complete the following template to assign these to the given list.

Print out the number of N-grams, and the first 5 N-grams in each list for N = 1, 2, 3, 4.

Note that this number is the number of occurrences of N-grams, which may not be unique in the list.

In [ ]:

```
Ngrams = [None]*5    # first slot is empty, then Ngram[1] will hold un

unigrams = []
for x in training_sents:
  unigrams.append(get_Ngrams_for_sentence(1,x))
unigrams = [tup for listlist in unigrams for tup in listlist]

bigrams = []
for x in training_sents:
  bigrams.append(get_Ngrams_for_sentence(2,x))
bigrams = [tup for listlist in bigrams for tup in listlist]

trigrams = []
for x in training_sents:
  trigrams.append(get_Ngrams_for_sentence(3,x))
trigrams = [tup for listlist in trigrams for tup in listlist]

quadgrams = []
for x in training_sents:
  quadgrams.append(get_Ngrams_for_sentence(4,x))
quadgrams = [tup for listlist in quadgrams for tup in listlist]

Ngrams[1] = unigrams
Ngrams[2] = bigrams
Ngrams[3] = trigrams
Ngrams[4] = quadgrams

print(f"There are {len(unigrams)} N-grams in Ngrams[1] and the first 5
print(f"There are {len(bigrams)} N-grams in Ngrams[2] and the first 5
print(f"There are {len(trigrams)} N-grams in Ngrams[3] and the first 5
print(f"There are {len(quadgrams)} N-grams in Ngrams[4] and the first
```

```
There are 1274667 N-grams in Ngrams[1] and the first 5 are:
[('<s>',), ('Muscle',), ('weakness',), ('did',), ('not',)]

There are 1217385 N-grams in Ngrams[2] and the first 5 are:
[('<s>', 'Muscle'), ('Muscle', 'weakness'), ('weakness', 'did'), ('di
d', 'not'), ('not', 'improve')]

There are 1160103 N-grams in Ngrams[3] and the first 5 are:
[('<s>', 'Muscle', 'weakness'), ('Muscle', 'weakness', 'did'), ('weak
ness', 'did', 'not'), ('did', 'not', 'improve'), ('not', 'improve', '
,')]

There are 1102821 N-grams in Ngrams[4] and the first 5 are:
[('<s>', 'Muscle', 'weakness', 'did'), ('Muscle', 'weakness', 'did',
'not'), ('weakness', 'did', 'not', 'improve'), ('did', 'not', 'improv
e', ','), ('not', 'improve', ',', 'and')]
```

Part E

We will now create a probability distribution for each of the Ngram collections. Note carefully that you must divide the frequency of each N-gram by the number of occurrences of N-grams, not the number of unique N-grams.

Complete the following template and then

1. Print out the total number of N-grams in each dictionary (they should be a bit smaller than the totals in the last part - why?).
2. Test your code by printing out the probability of the following Ngrams to 8 digits of precision:

```
('to',)
('to','the')
('to','the','house')
('to','the','house','.')
```

In [ ]:

```
# Create a defaultdict with the frequency distribution for the trainin

def get_Ngram_distribution(N,Ngrams):
  def_dict = defaultdict(int)
  ngrams = Ngrams[N]
  length = len(ngrams)
  for x in ngrams:
    def_dict[x] += 1/length
  return def_dict



                                     # your code here


# now create for N = 1,2,3,4

Ngram_distribution = [None]*5
one = get_Ngram_distribution(1, Ngrams)
two = get_Ngram_distribution(2, Ngrams)
three = get_Ngram_distribution(3, Ngrams)
four = get_Ngram_distribution(4, Ngrams)

Ngram_distribution[1] = one
Ngram_distribution[2] = two
Ngram_distribution[3] = three
Ngram_distribution[4] = four

print(f"The Probability of ('to',) is {round(Ngram_distribution[1].get
print(f"The Probability of ('to', 'the') is {round(Ngram_distribution[
print(f"The Probability of ('to','the','house') is {round(Ngram_distri
print(f"The Probability of ('to','the','house','.') is {round(Ngram_di


# your code here
```

```
The Probability of ('to',) is 0.02017703.

The Probability of ('to', 'the') is 0.00281341.

The Probability of ('to','the','house') is 9.48e-06.

The Probability of ('to','the','house','.') is 2.72e-06.
```

Probability and Perplexity

Now we will calculate the probability and the perplexity of sequences of tokens, using the principle of "Stupid Backoff" as explained in the paper:

https://aclanthology.org/D07-1090.pdf (https://aclanthology.org/D07-1090.pdf) and explicated in this StackOverflow post:

[https://stackoverflow.com/questions/16383194/stupid-backoff-implementation-clarification (https://stackoverflow.com/questions/16383194/stupid-backoff-implementation-clarification)](https://stackoverflow.com/questions/16383194/stupid-backoff-implementation-clarification)

Before describing "Stupid Backoff," let us consider the naive way to calculate the probability of a sequence of tokens which starts with `<s>` .

A simple and naive way to calculate probabilities of sequences of tokens

Suppose we have a quadrigram model (N = 4), we have a sequence of tokens

$$['\text{<s>}', w_1, w_2, \cdots, w_n],$$

(w_n may or may not be `<\s>`), and we have calculated all the N-gram probabilities in `Ngram_distribution[N]` for N = 1,2,3,4.

Then, let

$$\begin{aligned}
p_1 &= P(\ ('\text{<s>}', w_1)\ ) \ = \ \text{Ngram\_distribution}[2][('\text{<s>}', w_1)] \\
p_2 &= P(\ ('\text{<s>}', w_1, w_2)\ ) \ = \ \text{Ngram\_distribution}[3][('\text{<s>}', w_1, w_2)] \\
p_3 &= P(\ ('\text{<s>}', w_1, w_2, w_3)\ ) \ = \ \text{Ngram\_distribution}[4][('\text{<s>}', w_1, w_2, w_3)] \\
&\ldots \\
p_i &= P(\ (w_{i-3}, w_{i-2}, w_{i-1}, w_i)\ ) \ = \ \text{Ngram\_distribution}[4][(w_{i-3}, w_{i-2}, w_{i-1}, w_i)] \\
&\ldots \\
p_n &= P(\ (w_{n-3}, w_{n-2}, w_{n-1}, w_n)\ ) \ = \ \text{Ngram\_distribution}[4][(w_{n-3}, w_{n-2}, w_{n-1}, w_n)]
\end{aligned}$$

Finally, let

$$P('\text{<s>}', w_1, w_2, \cdots, w_n) \ = \ p_1 * p_2 * \cdots * p_n.$$

In other words, as each step, we use as much left context as we have available, up to N = 4.

What could possibly go wrong?

Well, if our sentence is from our training set, nothing! All the probabilities will have been calculated for all the possible N-grams.

However, when we have a separate training set, we have to account for the fact that **some N-grams (and even some tokens) may occur in the testing set which do not occur in the training set, and so their probability will be 0.**

There are various solutions, which we discussed in lectures 5 and 6, but the simplest (and very effective for large data sets) is "Stupid Backoff," recursively defined as follows for quadrigrams (and analogously for bigrams and trigrams):

```
P(w1, w2, w3, w4) = brown_N_grams[4][(w1, w2, w3, w4)] if this
is not 0, else:
                  = 0.4 * P(w2, w3, w4)              if this
is not 0, else:
                  = 0.4^2 * P(w3, w4)                if this
is not 0, else:
                  = 0.4^3 * P(w4)                    if this
is not 0, else:
                  = (0.4)^4 * (frequency of w4 in corpus / #
tokens in corpus)
```

The "discount factor" 0.4 was proposed by the originators of the method, and seems to work well in practice.

This calculation is unnecessary in generative models, since then we will train on the entire corpus, and only use available N-grams to produce sentences.

Problem 2 (40 pts)

Now we will calculate the probability of a sequence of tokens. We will warm up by considering the simple case, where no probabilities can be 0, and then consider the more complex case, where "stupid backoff" will be used.

Part A

For this part, complete the following template to create a function which will calculate the probability of a sequence of tokens, assuming that no probabilities are 0, for example, if you test a sentence from the training set.

Tests are provided following the cell in which you will write your code.

In [ ]:

```
# Probability of a list of tokens using N-grams
# W a list of tokens

# This assumes no probabilities are 0, e.g., if using on the training

# Now deal with growing prefixes, W must be of length at least 2

def P(N,W):
  answer = 1
  if N == 4:
    answer *= P(3, W[:3])
    for x in range(0, len(W) - N + 1):
      value = Ngram_distribution[N].get(W[:4])
      if value is None:
        value = 0
        answer = answer * value
      else:
        answer = answer * value
      W = W[1:]
  elif N == 3:
    answer *= P(2, W[:2])
    for x in range(0, len(W) - N + 1):
      value = Ngram_distribution[N].get(W[:3])
      if value is None:
        value = 0
        answer = answer * value
      else:
        answer = answer * value
      W = W[1:]
  elif N == 2:
    for x in range(0, len(W) - N + 1):
      value = Ngram_distribution[N].get(W[:2])
      if value is None:
        value = 0
        answer = answer * value
      else:
        answer = answer * value
      W = W[1:]

  return answer
```

The following are tests to make sure your code is working properly. The values printed should be the same.

In [ ]:
```
a = Ngram_distribution[2][('to','the',)]
b = Ngram_distribution[2][('the','house',)]
c = Ngram_distribution[2][('house','.')]

print('a*b*c:    ',a*b*c)
print('P(2,...):', P(2,('to','the','house','.')))
```

```
a*b*c:    1.5798085081133677e-11
P(2,...): 1.5798085081133677e-11
```

In [ ]:
```
a = Ngram_distribution[2][('to','the',)]
b = Ngram_distribution[3][('to','the','house',)]
c = Ngram_distribution[3][('the','house','.')]

print('a*b*c:    ',a*b*c)
print('P(3,...):', P(3,('to','the','house','.')))
```

```
a*b*c:    6.8984809553359e-13
P(3,...): 6.8984809553359e-13
```

In [ ]:
```
a = Ngram_distribution[2][('to','the',)]
b = Ngram_distribution[3][('to','the','house',)]
c = Ngram_distribution[4][('to',f'the','house','.')]

print('a*b*c:    ',a*b*c)
print('P(4,...):', P(4,('to','the','house','.')))
```

```
a*b*c:    7.256797296866889e-14
P(4,...): 7.256797296866889e-14
```

Part B

Now we will develop the probability for a sequence with the possibility that some N-grams, or even some tokens, are not in the training set. We will use the idea of "stupid backoff" explained in lecture.

Complete the following template and verify that it passes all the tests. Note that `P_stupid_backoff` is the same as `P` except that you use `PN_with_stupid_backoff` instead of `Ngram_distribution` .

In [ ]:
```
# Probability with stupid backoff
# same as previous, but have to use recursive (or iterative) method in
# calling Ngram_distribution directly

# W a list of tokens

num_all_tokens = len(brown.words())
```

```
# This returns backed-off probability for single N-gram
# len(W) must be N, this will try whole N-gram, then last N-1 tokens,

# Assumes W is a tuple

def PN_with_stupid_backoff(N,W):
  answer = 1
  if N == 4:
    for x in range(0, len(W) - N + 1):
      value = Ngram_distribution[N].get(W[:4])
      if value is None:
        value = 0.4 * PN_with_stupid_backoff(N-1, W[1:])
        answer = answer * value
      else:
        answer = answer * value
  elif N == 3:
    for x in range(0, len(W) - N + 1):
      value = Ngram_distribution[N].get(W[:3])
      if value is None:
        value = 0.4 * PN_with_stupid_backoff(N-1, W[1:])
        answer = answer * value
      else:
        answer = answer * value
  elif N == 2:
    for x in range(0, len(W) - N + 1):
      value = Ngram_distribution[N].get(W[:2])
      if value is None:
        value = 0.4 * PN_with_stupid_backoff(N-1, W[1:])
        answer = answer * value
      else:
        answer = answer * value
  elif N == 1:
    for x in range(0, len(W) - N + 1):
      value = Ngram_distribution[N].get(W[:1])
      if value is None:
        words = brown.words()
        value = list(brown.words()).count(W[0])/num_all_tokens
        answer = answer * value
      else:
        answer = answer * value

  return answer

# Now just substitute previous for call to Ngram_distribution

# Note that for training set, this will be same as P(N,W) since all pr

def P_stupid_backoff(N,W):
  answer = 1
  if N == 4:
    answer *= P_stupid_backoff(3, W[:3])
    for x in range(0, len(W) - N + 1):
      value = PN_with_stupid_backoff(N,W[:4])
      # value = Ngram_distribution[N].get(W[:4])
      if value is None:
        value = 0
        answer = answer * value
      else:
        answer = answer * value
      W = W[1:]
  elif N == 3:
    answer *= P_stupid_backoff(2, W[:2])
    for x in range(0, len(W) - N + 1):
      value = PN_with_stupid_backoff(N,W[:3])
      # value = Ngram_distribution[N].get(W[:3])
      if value is None:
        value = 0
        answer = answer * value
      else:
        answer = answer * value
      W = W[1:]
  elif N == 2:
    for x in range(0, len(W) - N + 1):
      value = PN_with_stupid_backoff(N,W[:2])
      # value = Ngram_distribution[N].get(W[:2])
      if value is None:
        value = 0
        answer = answer * value
      else:
        answer = answer * value
      W = W[1:]
  elif N == 1:
    for x in range(0, len(W) - N + 1):
      value = PN_with_stupid_backoff(N,W[:1])
      # value = Ngram_distribution[N].get(W[:1])
      if value is None:
        value = 0
        answer = answer * value
      else:
        answer = answer * value
      W = W[1:]

  return answer

# tests

P_stupid_backoff(2,('to','the','house','.'))
```

Out[74]:

```
1.5798085081133677e-11
```

In [ ]:

```
PN_with_stupid_backoff(4,('to','his','house','in'))
```

Out[75]:

```
2.234297284753796e-06
```

In [ ]:

```
P(4,('to','his','house','in'))
```

Out[76]:

```
0.0
```

In [ ]:

```
a = P(3,('his','house','in'))
a
```

Out[77]:

```
0.0
```

In [ ]:

```
b = P(3,('house','in'))
b
```

Out[78]:

```
1.3964358029711225e-05
```

In [ ]:

```
0.4*0.4*b
```

Out[79]:

```
2.2342972847537965e-06
```

In [ ]:

```
# 'grandstand' is in testing set but not in the training set

P(2,('where','is','the','grandstand'))
```

Out[80]:

```
0.0
```

In [ ]:

```
P_stupid_backoff(2,('where','is','the','grandstand'))
```

Out[81]:

```
9.088177927478767e-16
```

In [ ]:

```
a = P(2,('where','is'))
b = P(2,('is','the'))
c = P(2,('the','grandstand'))    # this is 0, so use 0.4*d instead of
d = list(brown.words()).count('grandstand') / len(brown.words())
a*b*0.4*d
```

Out[82]:

```
9.088177927478767e-16
```

Part C

Now we will implement the notion of *perplexity* as explained in lecture. Refer to the formula presented there to complete the following template, and verify that it passes all the tests.

Note: When we calculate perplexity, we do not count the start-of-sentence symbol `<s>` as a token. So the length of

```
'<s>' 'the' 'man' 'was' 'tall' '.' '</s>'
```

would be 6, and you would use the exponent $-(1/6)$ in the calculation.

In [ ]:
```
# Perplexity

# We assume that W starts with <s>, may not end with </s>
"""
def PP(N,W):
  length = len(W)
  word = W
  probs = []
  sum_in_log = 0

  for x in W:
    if x == "<s>":
      length -= 1

  for x in range(1, N):
    if x == 1:
      probs.append(P_stupid_backoff(x, word[:x]))
    else:
      probs.append(P_stupid_backoff(x, word[:x]) / P_stupid_backoff(x

  if N > len(word):
    for p in probs:
      sum_in_log += math.log(p)
    result = math.exp(sum_in_log)
    if result < 2.2250738585072014e-308:
      result = 2.2250738585072014e-308
    return result ** (-(1/(length)))

  for x in range(0, len(W)- N + 1):
    probOne = P_stupid_backoff(N,W[x:N+x])
    probTwo = P_stupid_backoff(N-1, W[x:N-1+x])
    probs.append(probOne)

  for p in probs:
    sum_in_log += math.log(p)
  result = math.exp(sum_in_log)
  if result < 2.2250738585072014e-308:
    result = 2.2250738585072014e-308
  return result ** (-1/length)
"""

def PP(N,W):
  length = len(W)
  word = W
  probs = []
  sum_in_log = 0

  for x in W:
    if x == "<s>":
      length -= 1

  if N == 3:
    if len(word) > 1:
      probs.append(P_stupid_backoff(2, word[:2]))
  if N == 4:
    if len(word) > 1:
      probs.append(P_stupid_backoff(2, word[:2]))
    if len(word) > 2:
      probs.append(P_stupid_backoff(3, word[:3])/P_stupid_backoff(2, w

  if N > len(word):
    for p in probs:
      sum_in_log += math.log(p)
    sum_in_log = sum_in_log * -1/length
    result = math.exp(sum_in_log)
    return result

  for x in range(0, len(W)- N + 1):
    probOne = P_stupid_backoff(N,W[x:N+x])
    if N > 2:
      probTwo = P_stupid_backoff(N-1, W[x:N-1+x])
    else:
      probTwo = 1
    probs.append(probOne/probTwo)

  for p in probs:
    sum_in_log += math.log(p)
  sum_in_log = sum_in_log * -1/length
  result = math.exp(sum_in_log)

  return result
```

```
In [ ]: PP(2,('<s>','the'))
Out[84]: 3901.8749999999886
```

```
In [ ]: PP(2,('<s>','the','man','went'))
Out[85]: 35708.79331810148
```

```
In [ ]: PP(2,('<s>','the','man','went','to','the', 'house','.','</s>'))
Out[86]: 5410.502630293523
```

```
In [ ]: PP(3,('<s>','the'))
Out[87]: 3901.8749999999886
```

```
In [ ]: PP(3,('<s>','the','man','went'))
Out[88]: 239724.60904832865
```

```
In [ ]: PP(3,('<s>','the','man','went','to','the', 'house','.','</s>'))
Out[89]: 73539.4118507504
```

Part D

Print out the first ten sentences in the training set, with their perplexities. Then do the same for the testing set.

Print out the text of the sentences in a readable form, e.g., for a sentence `w`, print it out using

```
' '.join(w[1:-1)
```

In [ ]:

```
start = 0
for x in range(10):
  sent = training_sents[x][1:len(training_sents[x])-1]
  sentence = ' '.join(sent)
  train = tuple(training_sents[x])
  print(f"{round(PP(2, train),2)} \t {sentence}")
```

```
75558.8          Muscle weakness did not improve , and the patient ne
eded first a cane , then crutches .
41114.8          He replaced the flashlight where it had been stowed
, got into his own car and backed it out of the garage .
64851.91         When he had given the call a few moments thought , h
e went into the kitchen to ask Mrs. Yamata to prepare tea and sushi f
or the visitors , using the formal English china and the silver tea s
ervice which had been donated to the mission , then he went outside t
o inspect the grounds .
54823.54         -- On the basis of a differentiability assumption in
function space , it is possible to prove that , for materials having
the property that the stress is given by a functional of the history
of the deformation gradients , the classical theory of infinitesimal
viscoelasticity is valid when the deformation has been infinitesimal
for all times in the past .
131936.22        She said sharks have no bones and shrimp swam backwa
rd .
102780.31        T. V. Barker , who developed the classification-angl
e system , was about to begin the systematic compilation of the index
when he died in 1931 .
15584.1          He was then in man's hands .
2517.48          4 .
22932.71         `` Fifteen minutes , then ! !
26299.58         Thus the cocktail party would appear to be the ideal
system , but there is one weakness .
```

In [ ]:

```
start = 0
for x in range(10):
  sent = testing_sents[x][1:len(testing_sents[x])-1]
  sentence = ' '.join(sent)
  test = tuple(testing_sents[x])
  print(f"{round(PP(3, test),2)} \t {sentence}")
```

```
66836.37         It is at least as important as the more dramatic att
empts to break down barriers of inequality in the South .
131885.57        the car's far windshield panel turned into a silver
web with a dark hole in the center .
109489.65        `` I was just thinking how things have changed .
92107.21         She smiled , and the teeth gleamed in her beautifull
y modeled olive face .
149699.23        `` There isn't a chance of Myra's letting anything l
ike that happen .
217504.84        On the other hand , many a pastor is so absorbed in
ministering to the intimate , personal needs of individuals in his co
ngregation that he does little or nothing to lead them into a sense o
f social responsibility and world mission .
34612.33         We live down by the Base commissary .
140178.39        For example , the BBB has reported it was receiving
four times as many inquiries about quack devices and 10 times as many
complaints compared with two years ago .
43182.19         As a result , life had become a kind of continuous m
ake-ready .
179499.92        Some of the poems express a mood of joy in a newly d
iscovered love ; ;
```

Part E

Finally, we will find the perplexities of the the testing set with bigrams, trigrams, and quadrigrams. Complete the following template the verify that your results are consistent with the test results.

In [ ]:

```
# Find all the probabilities of the sentences in the testing set, mult
# and take the $K^{th}$ root, where K is the number of tokens, excludi
# So, K = (sum of length of sentences) - (# of sentences)

# We need to take the product of many small probabilities, so use math
# underflow (the product would then simply be 0.0).

# Print out the perplexity as an integer.

import math
new_test = tuple(t for sublist in testing_sents for t in sublist)

def all_tests(N,W):
```

```
def all_tests(N,W):
  length = len(W)
  for x in W:
    if x == "<s>":
      length -= 1
  probs = []
  sum_in_log = 0
  probOne = 1
  probTwo = 1
  for x in testing_sents:
    if N == 3:
      if len(x) > 1:
        probOne *= P_stupid_backoff(2, tuple(x[:2]))
    if N == 4:
      if len(x) > 1:
        probOne *= P_stupid_backoff(2, tuple(x[:2]))
      if len(x) > 2:
        probOne *= P_stupid_backoff(3, tuple(x[:3]))/P_stupid_backoff(
    if N > len(x):
      probs.append(probOne)
    else:
      for y in range(0,len(x) - N + 1):
        probOne *= P_stupid_backoff(N,tuple(x[y:N+y]))
        if N > 2:
          probOne /= P_stupid_backoff(N-1, tuple(x[y:N-1+y]))
        else:
          probTwo = 1
      probs.append(probOne)
    probOne = 1
  return probs, length

def helper(probs, length):
  sum_in_log = 0
  for p in probs:
    sum_in_log += math.log(p)
  sum_in_log = sum_in_log * -1/length
  result = math.exp(sum_in_log)
  return result


one,two = all_tests(2,new_test)
print(f"The perplexity of the testing sets for 2-grams is {int(helper(

one,two = all_tests(3,new_test)
print(f"The perplexity of the testing sets for 3-grams is {int(helper(

one,two = all_tests(4,new_test)
print(f"The perplexity of the testing sets for 4-grams is {int(helper(
```

```
The perplexity of the testing sets for 2-grams is 20011
The perplexity of the testing sets for 2-grams is 29011.

The perplexity of the testing sets for 3-grams is 124608.

The perplexity of the testing sets for 4-grams is 294296.
```

Problem 3: Generative N-Gram Model (25 pts)

Now we will consider how to generate sentences using our N-gram model.

The idea is fairly simple. Suppose we have model using N=4 (quadrigrams -- the algorithm for bigrams and trigrams is analogous):

1. To get w_1 , choose a bigram $(\text{"<s>"}, w_1)$ randomly according the probability distribution stored in

$$\text{Ngram\_distribution}[2][\ (\text{"<s>"},)\ ].$$

2. To get w_2 , choose a bigram $(\text{"<s>"}, w_1, w_2)$ randomly according the probability distribution stored in

$$\text{Ngram\_distribution}[3][\ (\text{"<s>"}, w_1)\ ].$$

3. To get w_3 , choose a trigram $(\text{"<s>"}, w_1, w_2, w_3)$ randomly according the probability distribution stored in

$$\text{Ngram\_distribution}[4][\ (\text{"<s>"}, w_1, w_2)\ ].$$

4. Thereafter, for a sequence $(\text{"<s>"}, w_1, w_2, \ldots, w_{i-2}, w_{i-1})$, to get w_i , choose a quadrigram $(w_{i-3}, w_{i-2}, w_{i-1}, w_i)$ randomly according the probability distribution stored in

$$\text{Ngram\_distribution}[4][\ (w_{i-3}, w_{i-2}, w_{i-1})\ ].$$

5. When we generate the end of sentence marker `<\s>` we stop.

Part A

The first step, under the assumption that we are working with N-grams for N = 1,2,3, or 4, is to build a data structure that can sample from the distribution of next tokens given an (N-1)-gram of left context.

The best choice here is a nested default dictionary for N-grams for N = 2,3,4, the outer dictionary containing keys consisting of the first N-1 tokens (we'll call this the *prefix*), with the value being an inner dictionary holding a probability distribution for the last token (we'll call this *wn*).

For this problem, you need to redo the construction of the list of N-grams and the distributions for each N, using the *entire* set of sentences, not just the testing set you used for the previous problems. You can easily do this by copying and pasting code from above.

In [ ]:

```
All_Ngrams = [None]*5    # first slot is empty, then Ngram[1] will hol
                         # Ngram[2] will hold bigrams, etc. for ALL se

# your code here

# now create for N = 1,2,3,4

All_Ngram_distribution = [None]*5
all_sents = testing_sents + training_sents

unigrams_for_all = []
for x in all_sents:
  unigrams_for_all.append(get_Ngrams_for_sentence(1,x))
unigrams_for_all = [tup for listlist in unigrams_for_all for tup in li

bigrams_for_all = []
for x in all_sents:
  bigrams_for_all.append(get_Ngrams_for_sentence(2,x))
bigrams_for_all = [tup for listlist in bigrams_for_all for tup in list

trigrams_for_all = []
for x in all_sents:
  trigrams_for_all.append(get_Ngrams_for_sentence(3,x))
trigrams_for_all = [tup for listlist in trigrams_for_all for tup in li

quadgrams_for_all = []
for x in all_sents:
  quadgrams_for_all.append(get_Ngrams_for_sentence(4,x))
quadgrams_for_all = [tup for listlist in quadgrams_for_all for tup in 

All_Ngrams[1] = unigrams_for_all
All_Ngrams[2] = bigrams_for_all
All_Ngrams[3] = trigrams_for_all
All_Ngrams[4] = quadgrams_for_all


All_Ngram_distribution[1] = get_Ngram_distribution(1, All_Ngrams[1])
All_Ngram_distribution[2] = get_Ngram_distribution(2, All_Ngrams[2])
All_Ngram_distribution[3] = get_Ngram_distribution(3, All_Ngrams[3])
All_Ngram_distribution[4] = get_Ngram_distribution(4, All_Ngrams[4])

# your code here
```

Part B

Now we must build a date structure to solve the following problem: If we are working in an N-gram model for N = 2, 3, or 4, given a sequence of tokens

$$< s > \;\; w_1 \; w_2 \; w_3 \;\; \cdots \; w_i$$

generate a sample word w_{i+1} using the distribution of the N-grams of the form

$$w_{i-N+1} \;\; \cdots \; w_{i-1} \; w_i \; w_{i+1}.$$

In other words, we use the last $N - 1$ tokens of the sequence to determine a likely next token, given the distribution of N-grams starting with those $N - 1$ tokens.

The best way to do this is to build a nested dictionary. Let us call the first $N - 1$ tokens in an N-gram the *prefix* and the last token wn. Then the outer dictionary is a `defaultdict` whose keys are the prefixes and values are an inner `defaultdict` whose keys are the wn and whose values form a probability distribution for the ways that the prefix can be completed with a token wn.

To give a simple example, suppose that in our corpus there are only the following bigrams whose first token is 'the':

```
('the','boy'),    ('the','baby'),     ('the','baby'),    ('the','man')
```

Then our outer dictionary would have the prefix

```
('the',)
```

as a key, and the inner dictionary would store the probability that each of 'boy', 'baby', and 'man' would follow 'the', as shown in the next code cell.

Note that the prefix is an N-gram (a tuple) and wn is simply a token.

In [ ]:

```
D = defaultdict(lambda: None)

D[('the',)] = defaultdict(lambda: 0)
D[('the',)]['boy'] = 0.25
D[('the',)]['man'] = 0.25
D[('the',)]['baby'] = 0.5

D
```

Out[94]:

```
defaultdict(<function __main__.<lambda>()>,
            {('the',): defaultdict(<function __main__.<lambda>()>,
                         {'boy': 0.25, 'man': 0.25, 'baby': 0.5})})
```

Your task is to complete the following template and build a nested default dictionary giving the probability distributions for completions for (N-1)-grams.

Hint: For each N, you must create a `defaultdict` for all N-grams, whose key is the prefix and whose values are an inner `defaultdict`. For each N-gram, you should store the probability of the N-gram prefix+wn under the key wn. Then you must normalize these probabilities so that their sum is 1.0.

The end result will be that if you lookup a prefix (N-1 tokens), you will have a probability distribution of possible wn which can be used for the next token.

In [ ]:

```


def get_Ngram_dict(N):
  D = defaultdict(lambda: None)
  len_prefix = N-1
  for x in All_Ngrams[N]:
    if x[:N-1] in D:
      D[(x[:N-1])][x[N-1]] += 1
    else:
      D[(x[:N-1])] = defaultdict(lambda: 0)
      D[(x[:N-1])][x[N-1]] += 1
  count = 0
  for x,y in D.items():
    count = 0
    for a, b in y.items():
      count += b
    for a, b in y.items():
      y[a] = b/count


  return D                    # your code here

Ngram_nested_dict = [None]*5

for n in range(2,5):
    Ngram_nested_dict[n] = get_Ngram_dict(n)
```

In [105]:

```
# tests: these should sum to (close to) 1.0
sum(Ngram_nested_dict[2][('<s>',)].values())
```

Out[105]:

```
1.0000000000000742
```

In [106]:

```
sum(Ngram_nested_dict[3][('<s>','The')].values())
```

Out[106]:

```
0.9999999999999438
```

In [107]:

```
sum(Ngram_nested_dict[4][('<s>','When', 'the')].values())
```

Out[107]:

```
1.0000000000000007
```

Part C

Now that we have a way of sampling the next likely word, we will write a function which will predict the next word. You must sample from the probability distribution given a prefix, to choose a likely next word.

Hint: read about `numpy.random.choice` , in particular how you can set the parameter `p` to determine the probability of selecting a given key from the dictionary.

In [ ]:

```
# given a prefix, randomly choose next token using the appropriate pro

#  len(prefix) must be 1, 2, or 3 only

def next_word(prefix):
  length = len(prefix) + 1
  word = []
  prob = []
  dict = Ngram_nested_dict[length]
  count = 0
  for x,y in dict.items():
    if x == prefix:
      for a,b in y.items():
        word.append(a)
        prob.append(b)
  nextWord = np.random.choice(word, size=1, p=prob)[0]
  return nextWord

            # your code here
```

In [ ]:

```
# tests; these are random, your results may vary

next_word(('<s>',))
```

Out[100]:

```
'In'
```

In [ ]:

```
next_word(('<s>','The'))
```

Out[101]:

```
'Dark'
```

In [ ]:

```
next_word(('<s>','The','man'))
```

Out[102]:

```
'moved'
```

Part D

Complete the following template to generate a random sentence by starting with the unigram `('<s>',)` and extending it by sampling until you generate the token `</s>` .

In [ ]:

```
# N is the parameter in N-gram
"""
# string version
def generate_sentence(N):
  first_word = '<s>'
  last_word = '</s>'
  middle_words = first_word + " "
  current_word = ('<s>',)
  count = 0
  while not(last_word in middle_words):

    word = next_word(current_word)

    middle_words += word + " "
    words = middle_words.split()
    if len(words) < N - 1:
      current_word = tuple(words)
    else:
      current_word = tuple(words[len(words)- N + 1:len(words)])
  return middle_words

# list version
def generate_sentence(N):
  first_word = '<s>'
  last_word = '</s>'
  middle_words = [first_word]
  current_word = ('<s>',)
  count = 0
  while not(last_word in middle_words):

    word = next_word(current_word)

    middle_words.append(word)
    if len(middle_words) < N - 1:
      current_word = tuple(middle_words)
    else:
      current_word = tuple(middle_words[len(middle_words)- N + 1:len(m
  return middle_words
"""
# tuple version
def generate_sentence(N):
  first_word = '<s>'
  last_word = '</s>'
  middle_words = first_word + " "
  current_word = ('<s>',)
  count = 0
  while not(last_word in middle_words):

    word = next_word(current_word)

    middle_words += word + " "
    words = middle_words.split()
    if len(words) < N - 1:
      current_word = tuple(words)
    else:
      current_word = tuple(words[len(words)- N + 1:len(words)])
  return tuple(middle_words.split())
```

In [ ]:

```
# tests; these are random, your results may vary


w1 = generate_sentence(2)
print(int(PP(2,w1)), ' '.join(w1[1:-1]))

w2 = generate_sentence(3)
print(int(PP(3,w2)), ' '.join(w2[1:-1]))

w3 = generate_sentence(4)
print(int(PP(4,w3)), ' '.join(w3[1:-1]))
```

```
32745 Death ! !
246992 He held the razor , a lone figure .
379224 You don't need worry , Angelo .
```

Part E

Experiment with generating sentences for various values of N = 2, 3, 4. Do you see a difference in the quality of the sentences? How well does it do with punctuation and quotes? The typical view is that for larger values of N, the model is just "memorizing" the corpus. Do you think this is true? (You might look through the corpus to see what relationship your generated sentences, say for N = 4, have with the sentences in the corpus.

After the experiments, I discovered numerous things. N = 4 produced the best quality of the sentences, the closest to believing that it is created by humans. However, this might be because the model is simply "memorizing" the corpus and producing the contents in the corpus without generating a new sentence using the data.

The punctuations and quotes for N = 4 was good overall but when it came to N = 3 and especially N = 2, the punctuations were in places that should not have existed.

I would agree and disagree regarding whether the corpus simply memorizing the corpus. It certainly can generate sentences that exist inside the corpus but there existed variations within parts of the sentences that were not exactly the same as the corpus. However, it still looks identical in a few specific areas where the occurrences of some words in sequence is unique.

Overall, I believe that the value of N = 3 produces the best sentences that looks as it is human generated (even though some parts do not make sense in terms of sentences) and it also does not simply memorize the corpus and generate the exact same sentences in the corpus.

In [ ]: