CS 505 Homework 03: N-Gram Modelling

Due Monday 10/9 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.

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.

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.

I discussed the homework with Vineet Raju, Dominic Maglione, and Phillip Tran. I used Stack Overflow for minor debugging. I used chatGPT to debug potential issues with my code, but did not implement any of its suggestions since it was incorrect.

In [1]:
```
import math
import numpy as np
from numpy.random import shuffle, seed, choice
import nltk
from tqdm import tqdm
from collections import defaultdict

# 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]   Unzipping corpora/brown.zip.
```

Out[1]:
```
True
```

Problem One: Bag of N-Grams

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 [2]:
```
# First, shuffle the set of sentences

seed(0)

# your code here
sentences = list(brown.sents())
```

```
shuffled_sentences = sentences.copy()
shuffle(shuffled_sentences)

train_end_idx = int(len(shuffled_sentences) * 0.999)
training_sents = shuffled_sentences[:train_end_idx]
testing_sents = shuffled_sentences[train_end_idx:]

print(f"Length training set: {len(training_sents)}.")
print(f"Length of testing set: {len(testing_sents)}.")

print(f"Start of training set:\n{training_sents[0]}")
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', 'dramati
c', 'attempts', 'to', 'break', 'down', 'barriers', 'of', 'inequality', '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 [3]:
```
# put `<s>` at beginning and `</s>` at end of all sentences.

def bracket_sentence(sent):
  return ["<s>"] + sent + ["</s>"]   # your code here


# your code here
training_sents = list(map(bracket_sentence, training_sents))
testing_sents = list(map(bracket_sentence, testing_sents))

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

```
Start of training set:
['<s>', 'Muscle', 'weakness', 'did', 'not', 'improve', ',', 'and', 'the', 'p
atient', 'needed', 'first', 'a', 'cane', ',', 'then', 'crutches', '.', '</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 [4]:
```
# 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 (w1,w2

def get_Ngrams_for_sentence(N,s):
  return list(tuple(s[i:i+N]) for i in range(len(s) - N + 1))    # your code


# your code here
first_train_sentence = training_sents[0]
unigram = get_Ngrams_for_sentence(1, first_train_sentence)
bigram = get_Ngrams_for_sentence(2, first_train_sentence)
trigram = get_Ngrams_for_sentence(3, first_train_sentence)
four_gram = get_Ngrams_for_sentence(4, first_train_sentence)

print(unigram)
print(bigram)
print(trigram)
print(four_gram)
```

```
[('<s>',), ('Muscle',), ('weakness',), ('did',), ('not',), ('improve',),
(',',), ('and',), ('the',), ('patient',), ('needed',), ('first',), ('a',),
('cane',), (',',), ('then',), ('crutches',), ('.',), ('</s>',)]
[('<s>', 'Muscle'), ('Muscle', 'weakness'), ('weakness', 'did'), ('did', 'no
t'), ('not', 'improve'), ('improve', ','), (',', 'and'), ('and', 'the'), ('t
he', 'patient'), ('patient', 'needed'), ('needed', 'first'), ('first', 'a'),
('a', 'cane'), ('cane', ','), (',', 'then'), ('then', 'crutches'), ('crutche
s', '.'), ('.', '</s>')]
[('<s>', 'Muscle', 'weakness'), ('Muscle', 'weakness', 'did'), ('weakness',
'did', 'not'), ('did', 'not', 'improve'), ('not', 'improve', ','), ('improve
', ',', 'and'), (',', 'and', 'the'), ('and', 'the', 'patient'), ('the', 'pat
ient', 'needed'), ('patient', 'needed', 'first'), ('needed', 'first', 'a'),
('first', 'a', 'cane'), ('a', 'cane', ','), ('cane', ',', 'then'), (',', 'th
en', 'crutches'), ('then', 'crutches', '.'), ('crutches', '.', '</s>')]
[('<s>', 'Muscle', 'weakness', 'did'), ('Muscle', 'weakness', 'did', 'not'),
('weakness', 'did', 'not', 'improve'), ('did', 'not', 'improve', ','), ('not
', 'improve', ',', 'and'), ('improve', ',', 'and', 'the'), (',', 'and', 'the
', 'patient'), ('and', 'the', 'patient', 'needed'), ('the', 'patient', 'need
ed', 'first'), ('patient', 'needed', 'first', '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 [5]:

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


# your code here
for n in range(1, 5):
  Ngrams[n] = []
  for s in training_sents:
    Ngrams[n].extend(get_Ngrams_for_sentence(n, s))


for n in range(1, 5):
  print(f"There are {len(Ngrams[n])} N-grams in Ngrams[{n}] and the first 5
  print(Ngrams[n][:5])
  print()
```

```
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'), ('did', 'no
t'), ('not', 'improve')]

There are 1160103 N-grams in Ngrams[3] and the first 5 are:
[('<s>', 'Muscle', 'weakness'), ('Muscle', 'weakness', 'did'), ('weakness',
'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', 'improve', ','), ('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.

Note that we have set the probability of the unigram `<s>` to 1.0. This is because we are interested in calculating the probability of sentences, which *always* begin with the token `'<s>'` .

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 [6]:
```
# Create a defaultdict with the frequency distribution for the training set
def get_Ngram_distribution(N, Ngrams):
  # your code here
  Ngram_dist = defaultdict(int)

  for gram in Ngrams[N]:
    Ngram_dist[gram] += 1

  for key in Ngram_dist.keys():
    Ngram_dist[key] /= len(Ngrams[N])

  return Ngram_dist
```

```
# your code here
# now create for N = 1,2,3,4
Ngram_distribution = [None]*5

for n in range(1, 5):
  Ngram_distribution[n] = get_Ngram_distribution(n, Ngrams)

# Since all sentences will start with <s>, its probability should be 1.0

Ngram_distribution[1][('<s>',)] = 1.0

# tests

print(f"\nThe Probability of ('<s>',) is {np.around(Ngram_distribution[1][('
print(f"\nThe Probability of ('to',) is {np.around(Ngram_distribution[1][('t
print(f"\nThe Probability of ('to','the') is {np.around(Ngram_distribution[2
print(f"\nThe Probability of ('to','the','house') is {np.around(Ngram_distri
print(f"\nThe Probability of ('to','the','house','.')  is {np.around(Ngram_d
```

```
The Probability of ('<s>',) is 1.0.

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

and explicated in this StackOverflow post:

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

$$w_1, w_2, \cdots, w_n,$$

Assume we have calculated all the N-gram probabilities in `Ngram_distribution[N]` for N = 1,2,3,4.

We will calculate the probability of each successive token w_i in as much left context as we have, up to 3 (the last token in the 4-gram being w_i).

$$\begin{aligned}
p_1 &= \text{Ngram\_distribution}[1][(w_1,)] \\
p_2 &= \text{Ngram\_distribution}[2][(w_1, w_2)] \ / \ \text{Ngram\_distribution}[1][(w_1,)] \\
p_3 &= \text{Ngram\_distribution}[3][(w_1, w_2, w_3)] \ / \ \text{Ngram\_distribution}[2][(\text{'<s>'}, w_1, w_2)] \\
p_4 &= \text{Ngram\_distribution}[4][(w_1, w_2, w_3, w_4)] \ / \ \text{Ngram\_distribution}[3][(\text{'<s>'}, w_1, \\
&\ \ \cdots \\
p_i &= \text{Ngram\_distribution}[4][(w_{i-3}, w_{i-2}, w_{i-1}, w_i)] \ / \ \text{Ngram\_distribution}[3][(w_{i-3}, \\
&\ \ \cdots \\
p_n &= \text{Ngram\_distribution}[4][(w_{n-3}, w_{n-2}, w_{n-1}, w_n)] \ / \ \text{Ngram\_distribution}[3][(w_{n-3}
\end{aligned}$$

Note that if w_1 is `<s>`, then $p_1 = 1.0$.

Finally, let

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

One messy detail is dealing with the possibility of 0 counts (if that is possible); thus, if the numerator in the above expressions is 0, then the entire product is 0 (you want to avoid a divide by 0 error in the denominator).

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.** However, we want to make the best estimate of the probability we can!

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 bigrams, trigrams, and quadrigrams, and **using the probability calculations shown above.**

```
PN_stupid_backoff(w1) = p1   as defined above          # if
this is not 0, else:
                      = (frequency of w1 in whole corpus /
number of tokens in whole corpus)

PN_stupid_backoff(w1, w2) = p2   as defined above        #
if this is not 0, else:
                          = 0.4 * P_stupid_backoff(w2)    #
```

```
recursive, use previous definition

PN_stupid_backoff(w1, w2, w3) = p3   as defined above
# if this is not 0, else:
                              = 0.4 * P_stupid_backoff(w2,
w3)  # recursive, use definition above


PN_stupid_backoff(w1, w2, w3, w4) = p4  as defined above
# if this is not 0, else:
                                    = 0.4 *
P_stupid_backoff(w2, w3, w4)   # recursive, previous
definition
```

This accounts for how to backoff when trying to find the probability of some w_i in a left context of 1, 2, or 3 tokens.

Then we use these calculations instead of p_1, p_2 as in the previous algorithm, for trigrams it would be:

```
 P_stupid_backoff(w1, w2, w3, w4, ..., wn) =
PN_stupid_backoff(w1)
                                             *
PN_stupid_backoff(w1, w2)
                                             *
PN_stupid_backoff(w1, w2, w3)
                                             *
PN_stupid_backoff(w2, w3, w4)
                                             ...
                                             *
PN_stupid_backoff(w(n-2), w(n-1), wn)
```

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

Now we will calculate the probability of a sequence of tokens. We will warm up by considering the simple case, 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.

Since you may want to use this in the stupid backoff version, you should make sure that if the numerator in the conditional probability calculation is 0, immediately return 0, so that there is no possibility of divide by 0 in the denominator.

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

In [7]:
```python
# Probability of a list of tokens using N-grams
# W a list of tokens

# Calculate the probability of the last token in W, as we did in the calcula
# check numerator: if it is 0, return 0 immediately and do not do the divisi
# division by 0)

# N == len(W)

def PN(N,W):
  # your code here
  # precondition: N == len(W)

  numerator = Ngram_distribution[N].get(W, 0)

  if Ngram_distribution[N-1]:
    denominator = Ngram_distribution[N-1].get(W[:-1], 0)
  else:
    # unigram case
    return numerator

  # prevent division by 0
  if denominator == 0:
    return 0

  return numerator / denominator


# Now calculate for a whole sequence: use PN(..) for bigram, trigram, etc. u
# model, then slide PN(...) across the sequence, as shown above.

def P(N,W):
  # your code here
  proba = 1.0

  for i in range(2, N):
    proba *= PN(i, W[:i])

  for i in range(len(W) - N + 1):
    proba *= PN(N, W[i:i+N])

  return proba
```

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

In [8]:
```
#  Sentence:  He frowned.
#  Using a bigram model

a = Ngram_distribution[2][('<s>','He')]
b = Ngram_distribution[2][('He','frowned')] / Ngram_distribution[1][('He',)]
c = Ngram_distribution[2][('frowned','.')] / Ngram_distribution[1][('frowned
d = Ngram_distribution[2][('.','</s>')] / Ngram_distribution[1][('.',)]


print('a=', a)
print('  ', PN(2,('<s>','He')))
print('b =', b)
print('  ', PN(2,('He','frowned')))
print('c=', c)
print('  ', PN(2,('frowned','.')))
print('d=', d)
print('  ', PN(2,('.','</s>')))

print('\na*b*c*d:  ',a*b*c*d)
print('P(2,...): ', P(2,('<s>','He','frowned','.','</s>')))
```

```
a= 0.002346012148991486
   0.002346012148991486
b = 0.000351478118528877
   0.000351478118528877
c= 0.39264499316157175
   0.39264499316157175
d= 1.0470533150975245
   1.0470533150975245

a*b*c*d:   3.389982137368031e-07
P(2,...):  3.389982137368031e-07
```

In [9]:
```
#  Sentence:  He frowned.
#  Using a trigram model

a = Ngram_distribution[2][('<s>','He')]
b = Ngram_distribution[3][('<s>','He','frowned')] / Ngram_distribution[2][('
c = Ngram_distribution[3][('He','frowned','.')] / Ngram_distribution[2][('He
d = Ngram_distribution[3][('frowned','.','</s>')] / Ngram_distribution[2][('

print('a*b*c*d:  ',a*b*c*d)
print('P(3,...): ', P(3,('<s>','He','frowned','.','</s>')))
```

```
a*b*c*d:   9.492186072583041e-07
P(3,...):  9.492186072583041e-07
```

In [10]:
```
#  Sentence:  He frowned.
#  Using a quadrigram model

a = Ngram_distribution[2][('<s>','He')]
b = Ngram_distribution[3][('<s>','He','frowned')] / Ngram_distribution[2][('
c = Ngram_distribution[4][('<s>','He','frowned','.')] / Ngram_distribution[3
d = Ngram_distribution[4][('He','frowned','.','</s>')] / Ngram_distribution[

print('a*b*c*d:  ',a*b*c*d)
print('P(4,...): ', P(4,('<s>','He','frowned','.','</s>')))
```

```
a*b*c*d:   9.538640808692934e-07
P(4,...):  9.538640808692934e-07
```

In [11]:
```
# Sentence:  I love NLP
# using trigrams

# This shows that you should test the numerator to see if it is 0, because t
# you can return 0 immediately, and not have the possibility of division by
# which would occur in the cases d below (c is 0 but would not cause divisio

a  = Ngram_distribution[2][('<s>','I')]
b  = Ngram_distribution[3][('<s>','I','love')] / Ngram_distribution[2][('<s>
c  = Ngram_distribution[3][('I','love','NLP')] / Ngram_distribution[2][('I',
d1 = Ngram_distribution[3][('love','NLP','</s>')]
d2 = Ngram_distribution[2][('love','NLP')]

print('a=',a,'\nb=',b,'\nc=',c,'\nd1=',d1,'\nd2=',d2,'\n')

print('a*b*d*d1:  ',a*b*c*d1)
print('P(3,...): ', P(3,('<s>','I','love','NLP','</s>')))
```

```
a= 0.001128648701930778
b= 0.0015274769289326804
c= 0.0
d1= 0
d2= 0

a*b*d*d1:   0.0
P(3,...):  0.0
```

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.

In [12]:
```
# Probability with stupid backoff
# same as previous, but have to use recursive (or iterative) method instead
# calling Ngram_distribution directly

# W a list of tokens
# 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, then N

# Assumes W is a tuple

# calculate for a particular length N-gram
# must have N == len(W)

all_words = list(brown.words())


def PN_with_stupid_backoff(N,W):
  # your code here
  proba = PN(N, W)

  if proba > 0:
    return proba

  elif N == 1:
    return all_words.count(W[0]) / len(all_words)

  return 0.4 * PN_with_stupid_backoff(N-1,W[1:])




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

# Really the same as P(N,W) except using the previous function, and no need
# since it never returns 0 for a sequence using words from the corpus



def P_stupid_backoff(N,W):
  # your code here
  proba = 1.0

  for i in range(2, N):
    proba *= PN_with_stupid_backoff(i, W[:i])

  for i in range(len(W) - N + 1):
    proba *= PN_with_stupid_backoff(N, W[i:i+N])

  return proba
```

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

In [13]:
```
# 'grandstand' is in testing set but not in the training set
# This uses bigrams

P(2,('<s>','where','is','the','grandstand'))
```

Out[13]:
```
0.0
```

In [14]:
```
a = PN(2,('<s>','where'))
b = PN(2,('where','is'))
c = PN(2,('is','the'))
d2 = PN(2,('the','grandstand'))     # this is 0, so use less context
d1 = PN(1,('grandstand'))           # this is 0, so use 0.4*d instead of d2
d  = list(brown.words()).count('grandstand') / len(brown.words())

print('a =',a,'\nb =',b,'\nc =',c,'\nd2=',d2,'\nd1=',d1,'\nd =',d,'\n')
print(a*b*c*(0.4*d))
print(P_stupid_backoff(2,('<s>','where','is','the','grandstand')))
```

```
a = 1.6428656505542618e-06
b = 0.006166391726133832
c = 0.08182229363508187
d2= 0.0
d1= 0
d = 8.611840246918683e-07

2.855359302895301e-16
2.855359302895301e-16
```

In [15]:
```
# 'grandstand' is in testing set but not in the training set
# This uses trigrams

P(3,('<s>','where','is','the','grandstand'))
```

Out[15]:
```
0.0
```

In [16]:
```
a  = PN(2,('<s>','where'))                 # <= this works
b3 = PN(3,('<s>','where','is'))           # this is 0, so try less context
b2 = PN(2,('where','is'))                 # <= this works
c  = PN(3,('where','is','the'))            # <= this works

d3 = PN(3,('is','the','grandstand'))      # this is 0, so try less context
d2 = PN(2,('the','grandstand'))           # this is 0, so try less context
d1 = PN(1,('grandstand'))                 # this is 0, so use probability in w
d  = list(brown.words()).count('grandstand') / len(brown.words())     # <= t

print('a =',a,'\nb3=',b3,'\nb2=',b2,'\nc =',c,'\nd3=',d3,'\nd2=',d2,'\nd1=',

# every time we try less context, must multiply by 0.4, so we
# use 0.4*b2 instead of b3 and 0.4*0.4*e instead of d3
print(a*(0.4*b2)*c*(0.4*0.4*d))
print(P_stupid_backoff(3,('<s>','where','is','the','grandstand')))
```

```
a = 1.6428656505542618e-06
b3= 0.0
b2= 0.006166391726133832
c = 0.4197506600707006
d3= 0.0
d2= 0.0
d1= 0
d = 8.611840246918683e-07

2.3436917228933544e-16
2.3436917228933544e-16
```

In [17]:
```
a  = PN(2,('<s>','The'))                      #   <= this works

b3 = PN(3,('<s>','The','grandstand'))         # this is 0, so try less context
b2 = PN(2,('The','grandstand'))               # this is 0, so try less context
b1 = PN(1,('grandstand',))                    # this is 0, so have to use frequ
b  = list(brown.words()).count('grandstand') / len(brown.words())    #   <=

c3 = PN(3,('The','grandstand','fell'))        # this is 0, so try less context
c2 = PN(2,('grandstand','fell'))              # this is 0, so try less context
c1 = PN(1,('fell',))                          # ok, this works

d3 = PN(3,('grandstand','fell','down'))     # this is 0, so try less context
d2 = PN(2,('fell','down'))                  #   <= this works

e3 = PN(3,('fell','down','</s>'))           # this is 0, so try less context
e2 = PN(2,('down','</s>'))                  #  <= this works

# every time we tried a smaller N, we have to multiply by 0.4
# so we use a, then 0.4*0.4*b0, then 0.4*0.4*c1, then 0.4*d2, then 0.4*e2e.

print('a =',a,'\nb3=',b3,'\nb2=',b2,'\nb1=',b1,'\nb =',b)
print('c3=',c3,'\nc2=',c2,'\nc1=',c1,'\nd3=',d3,'\nd2=',d2,'\ne3=',e3,'\ne2=

print(a * (0.4*0.4*b) * (0.4*0.4*c1) * (0.4*d2) * (0.4*e2) )
print(P_stupid_backoff(3,('<s>','The','grandstand','fell','down','</s>')))
```

```
a = 0.005372992110137713
b3= 0.0
b2= 0.0
b1= 0
b = 8.611840246918683e-07
c3= 0
c2= 0
c1= 7.217571334317l2e-05
d3= 0
d2= 0.01138101429453831
e3= 0.0
e2= 0.0011817757506744071

1.839836556015632e-20
1.839836556015632e-20
```

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.

In [18]:
```
# Perplexity

# We assume that W starts with <s>, may not end with </s>

def PP(N,W):
  # your code here
  p = P_stupid_backoff(N, W)
  if p == 0.0:
    return 0.0
  return p**( -1 / (len(W)-1) )
```

If we know that no probabilities can be 0, then P is same as P_stupid_backoff

```
In [19]: PP(2,('<s>','The'))

Out[19]: 186.11603730316466

In [20]: P(2,('<s>','The'))**(-1/1)

Out[20]: 186.11603730316466

In [21]: P_stupid_backoff(2,('<s>','The'))**(-1/1)

Out[21]: 186.11603730316466

In [22]: PP(2,('<s>','The','man','went'))

Out[22]: 308.91157551766736

In [23]: P(2,('<s>','The','man','went'))**(-1/3)

Out[23]: 308.91157551766736

In [24]: P_stupid_backoff(2,('<s>','The','man','went'))**(-1/3)

Out[24]: 308.91157551766736

In [25]: PP(2,('<s>','The','man','went','to','the', 'house','.','</s>'))

Out[25]: 35.03849995731908

In [26]: P(2,('<s>','The','man','went','to','the', 'house','.','</s>'))**(-1/8)

Out[26]: 35.03849995731908

In [27]: P_stupid_backoff(2,('<s>','The','man','went','to','the', 'house','.','</s>')

Out[27]: 35.03849995731908
```

When probabilities may be 0, we must use stupid backoff

```
In [28]: PP(2,('<s>','where','is','the','grandstand'))

Out[28]: 7692.8065013245405

In [29]: P_stupid_backoff(2,('<s>','where','is','the','grandstand'))**(-1/4)

Out[29]: 7692.8065013245405
```

```
In [30]: PP(3,('<s>','where','is','the','grandstand'))

Out[30]: 8082.112259400884

In [31]: P_stupid_backoff(3,('<s>','where','is','the','grandstand'))**(-1/4)

Out[31]: 8082.112259400884

In [32]: PP(3,('<s>','The','grandstand','fell','down','</s>'))

Out[32]: 8852.055970670379

In [33]: P_stupid_backoff(3,('<s>','The','grandstand','fell','down','</s>'))**(-1/5)

Out[33]: 8852.055970670379

In [34]: PP(4,('<s>','The','grandstand','fell','down','</s>'))

Out[34]: 15339.392368477242

In [35]: P_stupid_backoff(4,('<s>','The','grandstand','fell','down','</s>'))**(-1/5)

Out[35]: 15339.392368477242
```

Part D

Print out the first ten sentences in the training set, with their perplexities to 2 decimal places, using trigrams. 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)
```

Notice the perplexities of the training set are generally smaller than the testing set!

```
In [36]: # your code here
         for w in training_sents[:10]:
           sent = " ".join(w[1:-1])
           perplexity = np.round(PP(N=3, W=tuple(w)), 2)
           print(f"{perplexity}\t{sent}")
```

```
10.76   Muscle weakness did not improve , and the patient needed first a can
e , then crutches .
10.74   He replaced the flashlight where it had been stowed , got into his o
wn car and backed it out of the garage .
8.42    When he had given the call a few moments thought , he went into the
kitchen to ask Mrs. Yamata to prepare tea and sushi for the visitors , using
the formal English china and the silver tea service which had been donated t
o the mission , then he went outside to inspect the grounds .
10.04   -- 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 de
formation has been infinitesimal for all times in the past .
4.88    She said sharks have no bones and shrimp swam backward .
9.62    T. V. Barker , who developed the classification-angle system , was a
bout to begin the systematic compilation of the index when he died in 1931 .
10.32   He was then in man's hands .
32.57   4 .
15.57   `` Fifteen minutes , then ! !
11.27   Thus the cocktail party would appear to be the ideal system , but th
ere is one weakness .
```

In [37]:
```python
# your code here
for w in testing_sents[:10]:
  sent = " ".join(w[1:-1])
  perplexity = np.round(PP(N=3, W=tuple(w)), 2)
  print(f"{perplexity}\t{sent}")
```

```
176.79  It is at least as important as the more dramatic attempts to break d
own barriers of inequality in the South .
1306.74 the car's far windshield panel turned into a silver web with a dark
hole in the center .
69.22   `` I was just thinking how things have changed .
1071.75 She smiled , and the teeth gleamed in her beautifully modeled olive
face .
438.24  `` There isn't a chance of Myra's letting anything like that happen
.
174.63  On the other hand , many a pastor is so absorbed in ministering to t
he intimate , personal needs of individuals in his congregation that he does
little or nothing to lead them into a sense of social responsibility and wor
ld mission .
923.7   We live down by the Base commissary .
484.32  For example , the BBB has reported it was receiving four times as ma
ny inquiries about quack devices and 10 times as many complaints compared wi
th two years ago .
204.82  As a result , life had become a kind of continuous make-ready .
210.44  Some of the poems express a mood of joy in a newly discovered love ;
;
```

Part E

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

In [38]:
```
# Find all the probabilities of the sentences in the testing set, multiply t
# and take the $K^{th}$ root, where K is the number of tokens, excluding the
# So, K = (sum of length of sentences) - (# of sentences)

# We need to take the product of many small probabilities, so use math.log a
# underflow.

# Print out the perplexity as an integer.

import math

# your code here
K = sum(len(s) for s in testing_sents) - len(testing_sents)

perplexity_bigrams = int(np.exp(-1/K * np.sum(
                     np.log([P_stupid_backoff(2, tuple(testing_sents[i]))
                     for i in range(len(testing_sents))]))))

perplexity_trigrams = int(np.exp(-1/K * np.sum(
                      np.log([P_stupid_backoff(3, tuple(testing_sents[i]))
                      for i in range(len(testing_sents))]))))

perplexity_four_grams = int(np.exp(-1/K * np.sum(
                        np.log([P_stupid_backoff(4, tuple(testing_sents[i]))
                        for i in range(len(testing_sents))]))))

print(f"The perplexity of the testing set for 2-grams is {perplexity_bigrams
print(f"The perplexity of the testing set for 3-grams is {perplexity_trigram
print(f"The perplexity of the testing set for 4-grams is {perplexity_four_gr
```

```
The perplexity of the testing set for 2-grams is 387.
The perplexity of the testing set for 3-grams is 496.
The perplexity of the testing set for 4-grams is 957.
```

Problem 3: Generative N-Gram Model

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

$$\text{Ngram\_distribution}[2][\ (\text{"<s>"}, w_1)\ ].$$

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

$$\text{Ngram\_distribution}[3][\ (\text{"<s>"}, w_1, w_2)\ ]\ /\ \text{Ngram\_distribution}[2][\ (\text{"<s>"}, w_1)\ ].$$

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

$$\text{Ngram\_distribution}[4][\ (\text{"<s>"}, w_1, w_2, w_3)\ ]\ /\ \text{Ngram\_distribution}[3][\ (\text{"<s>"}, w_1,$$

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

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

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

The problem is that this is difficult if we simply use the formulae given above: what we need is a separate probability distribution for each prefix.

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 [39]:
```
All_Ngrams = [None]*5    # first slot is empty, then Ngram[1] will hold unig
                         # Ngram[2] will hold bigrams, etc.

# add <s> </s> tokens
sentences = list(map(bracket_sentence, sentences))

for n in range(1, 5):
  All_Ngrams[n] = []
  for s in sentences:
    All_Ngrams[n].extend(get_Ngrams_for_sentence(n, s))

# your code here
def get_All_Ngram_distribution(N, All_Ngrams):
  All_Ngram_dist = defaultdict(int)

  for gram in All_Ngrams[N]:
    All_Ngram_dist[gram] += 1

  return All_Ngram_dist
```

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

All_Ngram_distribution = [None]*5

for n in range(1, 5):
  All_Ngram_distribution[n] = get_All_Ngram_distribution(n, All_Ngrams)
```

Part B

Now we must build a data 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 [40]:
```
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[40]: 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 look up a prefix (N-1 tokens), you will have a probability distribution of possible wn which can be used for the next token.

```
In [41]: # now build a dictionary of lists of completions, with probabilities
         # For N-gram, key is prefix, of length N-1, which returns a dictionary
         # with keys that are tokens (the last token wn in the N-gram), with
         # values the number of times that N-gram (prefix),wn occurs.

         # N here is the length of the whole N-gram, so the prefix is of length N-1

         def get_Ngram_dict(N):
           # your code here
           nested = defaultdict(defaultdict)
           for ngram in All_Ngram_distribution[N]:
             inner_key = ngram[-1]
             inner_value = All_Ngram_distribution[N].get(ngram, 0.0)
             total = All_Ngram_distribution[N-1].get(ngram[:-1], 0)
             inner_value = inner_value if total == 0 else inner_value / total
             nested[ngram[:-1]].update({inner_key: inner_value})
           return nested

         Ngram_nested_dict = [None]*5

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

```
In [42]: # tests: these should sum to (close to) 1.0

         sum(Ngram_nested_dict[2][('<s>',)].values())
```

```
Out[42]: 1.0000000000000744
```

```
In [43]: sum(Ngram_nested_dict[3][('<s>','The')].values())
```

```
Out[43]: 0.9999999999999502
```

```
In [44]: sum(Ngram_nested_dict[4][('<s>','When', 'the')].values())
```

```
Out[44]: 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 [45]:
```
# given a prefix, randomly choose next token using the appropriate probabili

#  len(prefix) must be 1, 2, 3, or 4

def next_word(prefix):
  # your code here
  N = len(prefix)+1
  choices, probas = list(zip(*Ngram_nested_dict[N][prefix].items()))
  return str(np.random.choice(choices, p=probas))
```

In [46]:
```
# tests

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

Out[46]:
```
'It'
```

In [47]:
```
next_word(('<s>','The'))
```

Out[47]:
```
'subjects'
```

In [48]:
```
next_word(('<s>','The','man'))
```

Out[48]:
```
'stood'
```

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 [49]:
```
# N is the parameter in N-gram

def generate_sentence(N):
  # your code here
  generated = ('<s>',)
  idx = 0

  while generated[-1] != '</s>':
    if len(generated) >= N:
      idx += 1
    generated += (next_word(generated[idx:]),)

  return generated
```

In [50]:
```
# tests  -- run this cell many times!



w1 = generate_sentence(2)

print(np.around(PP(2,w1),2), ' '.join(w1[1:-1]),'\n')

w2 = generate_sentence(3)

print(np.around(PP(3,w2),2), ' '.join(w2[1:-1]),'\n')

w3 = generate_sentence(4)

print(np.around(PP(4,w3),2), ' '.join(w3[1:-1]),'\n')
```

```
63.1 Determine if we become so '' , it , residence in the process .

15.18 He went into the pool too .

3.41 a middle distance often containing the major motif ; ;
```

Part E

Experiment with generating sentences for various values of N = 2, 3, 4. How do the perplexities compare? 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.

In [51]:
```
w = generate_sentence(2)

print('N = 2:', np.around(PP(2,w),2), ' '.join(w[1:-1]),'\n')

w = generate_sentence(3)
```

```
print('N = 3:', np.around(PP(3,w),2), ' '.join(w[1:-1]),'\n')

w = generate_sentence(4)

print('N = 4:', np.around(PP(4,w),2), ' '.join(w[1:-1]),'\n')
```

```
N = 2: 47.83 He started laughing .

N = 3: 5.09 Do patriots everywhere know enough to send Lieutenant-Colonel He
nry Leavenworth from Detroit as a single day to reassure me .

N = 4: 1.89 The total of these three declarative statements and detailed des
criptions of the formats and functions of each of these , as distinct feelin
gs and concepts , to say the least of it .
```

In [52]:

```
num_generations = 1000
counts = 0

# check how much the model is just "memorizing" the corpus for N=4
for _ in range(num_generations):
  w = generate_sentence(4)
  counts += sentences.count(list(w))

print(f"In {num_generations} generations with N = 4, there are {counts}\
 sentences that exactly match the corpus.")
```

```
In 1000 generations with N = 4, there are 522 sentences that exactly match t
he corpus.
```

Based off of initial exploration of generation I notice that the greater the value of N, the greater the generation length typically is. This is interesting because perpexlity decreases as the length of the generated output increases. After running multiple generations of $N = 2, 3, 4$ I like the quality of $N = 2$ the best because it generates shorter sentences that make more sense. Probablistically, it makes sense that the bigrams would make more sense because they typically generate less tokens making it easier to generate short text that makes more sense and are able to sample from a greater space since they are the most prevalent n-gram greater than 1. As dataset size increases, however, I believe we would see larger N being better since they will naturally have more examples. Punctuation greatly varies between generations with some generations being reasonable while others having extraneous punctuation such as "!!, :, ????, etc." After running the above code to check how much the model is just "memorizing" the corpus for 1000 generations, I do think it may be memorizing since a good portion of the sentences are exact copies of the sentences in the corpus.