CS 6320.002: Natural Language Processing
Fall 2019
Homework 1

Xiaojie Zhu
XXZ180012

Writeup Question 1.1: What data types did you use for the two counters and the vocabulary, and did you initialize the vocabulary to be empty or already containing some token(s)? Explain why. You may want to wait to answer this question until after you complete the programming parts, in case you change your mind.
Answer:
I use dictionaries to store the *self.ngram_counts* and the *self.context_counts*. I initialized the vocabulary with "<s>".

The reason to use dictionaries to store the counters is that dictionary is a key-to-value map and I can use it to store and retrieve the values in $O(1)$ time complexity, which would be very convenient when I need to use the data to compute the possibilities.

The vocabulary was initialized to contain the word "<s>" because the text itself does not contain <s> and thus we need to initialize it into the vocabulary set.

Writeup Question 1.2: Why do we have a special case for unseen contexts? Why do we set the probability to be $1/|V|$?
Answer:
We need to have special case for unseen contexts because the probability of the n-gram $(word, context)$ is calculated by the equation:

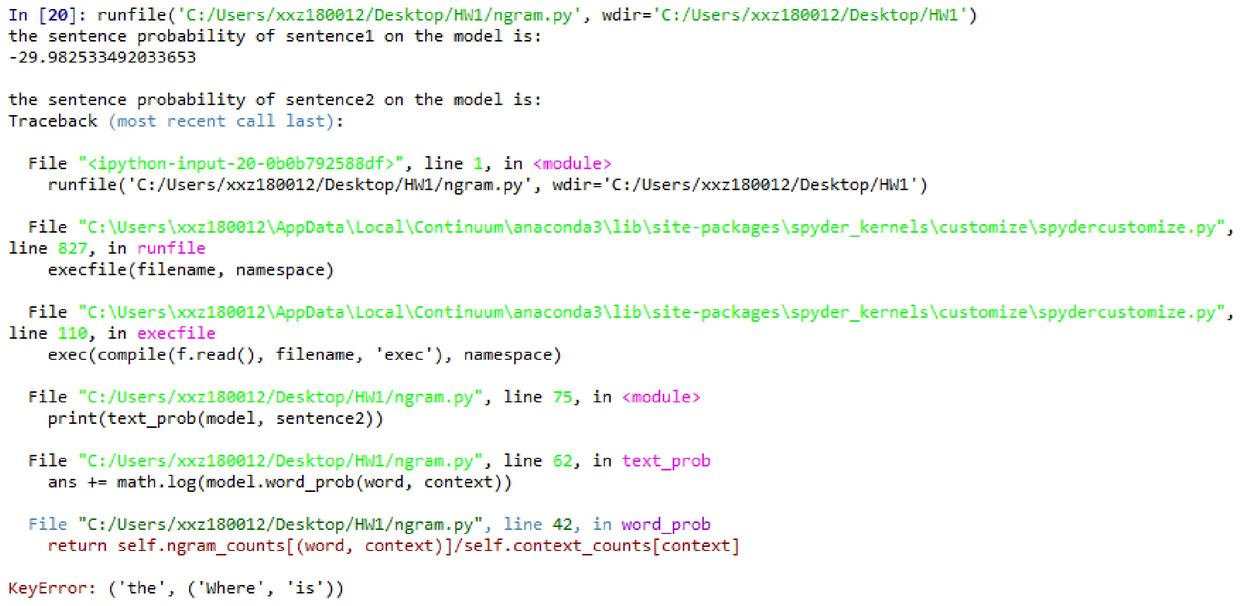
$$p(word|context) = \frac{c(context, word)}{c(context)}$$

If we do not have seen the context before, then our value of $c(context)$ would be 0 or $undefined$, which cannot be used as a divisor.

We set the probability to be $1/|V|$ because this is a relatively small value, which could be used to represent the scarce occurrence of the probability of this n-gram.

Writeup Question 1.3: What are your model's predicted probabilities for these two sentences? Did anything unusual happen when you ran the second sentence? Explain what happened and why. (If you're not sure or can't remember from what we talked about in class, try stepping through your code to see what's going on.)

Answer:

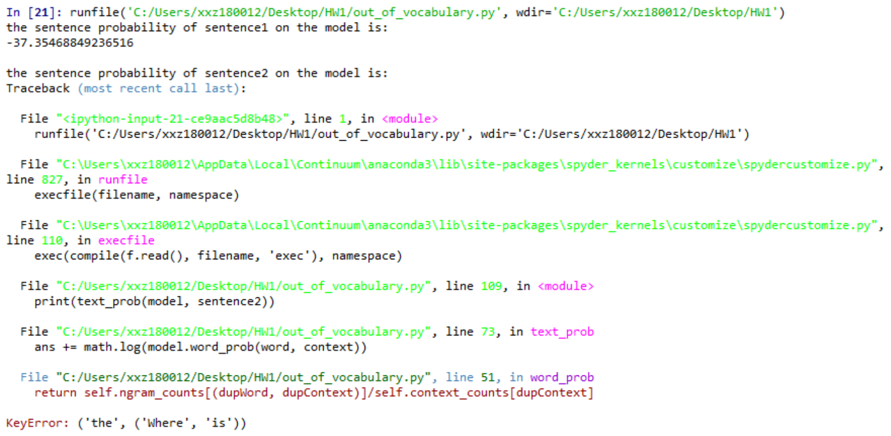


```
In [20]: runfile('C:/Users/xxz180012/Desktop/HW1/ngram.py', wdir='C:/Users/xxz180012/Desktop/HW1')
the sentence probability of sentence1 on the model is:
-29.982533492033653

the sentence probability of sentence2 on the model is:
Traceback (most recent call last):

  File "<ipython-input-20-0b0b792588df>", line 1, in <module>
    runfile('C:/Users/xxz180012/Desktop/HW1/ngram.py', wdir='C:/Users/xxz180012/Desktop/HW1')

  File "C:\Users\xxz180012\AppData\Local\Continuum\anaconda3\lib\site-packages\spyder_kernels\customize\spydercustomize.py",
line 827, in runfile
    execfile(filename, namespace)

  File "C:\Users\xxz180012\AppData\Local\Continuum\anaconda3\lib\site-packages\spyder_kernels\customize\spydercustomize.py",
line 110, in execfile
    exec(compile(f.read(), filename, 'exec'), namespace)

  File "C:/Users/xxz180012/Desktop/HW1/ngram.py", line 75, in <module>
    print(text_prob(model, sentence2))

  File "C:/Users/xxz180012/Desktop/HW1/ngram.py", line 62, in text_prob
    ans += math.log(model.word_prob(word, context))

  File "C:/Users/xxz180012/Desktop/HW1/ngram.py", line 42, in word_prob
    return self.ngram_counts[(word, context)]/self.context_counts[context]

KeyError: ('the', ('Where', 'is'))
```

As we can see from the results, the probabilities for the first sentence is -29.982533492033653. As for the second sentence, there is an error. The reason for the second sentence to be an error is that we have a trigram that is unseen in our ngram_counts, thus the dictionary will return a keyError and we cannot carry out the division calculation.

Writeup Question 2.1: Try predicting the log probability of that second sentence again. Have we fixed whatever was going on? Why or why not?

Answer:



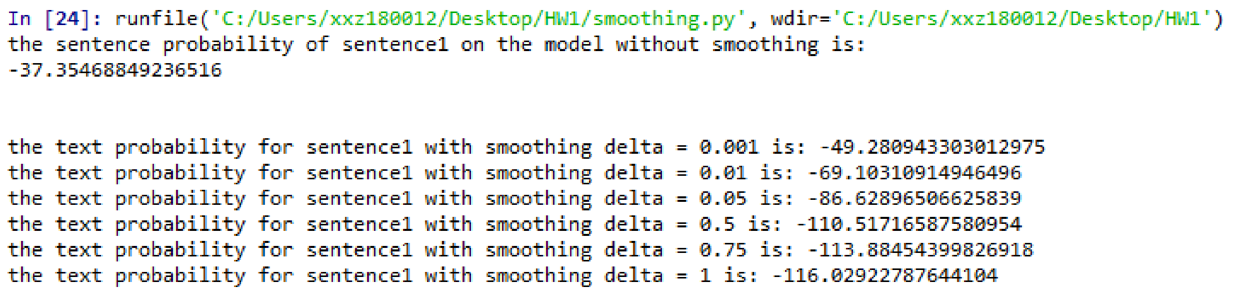
```
In [21]: runfile('C:/Users/xxz180012/Desktop/HW1/out_of_vocabulary.py', wdir='C:/Users/xxz180012/Desktop/HW1')
the sentence probability of sentence1 on the model is:
-37.35468849236516

the sentence probability of sentence2 on the model is:
Traceback (most recent call last):

  File "<ipython-input-21-ce9aac5d8b48>", line 1, in <module>
    runfile('C:/Users/xxz180012/Desktop/HW1/out_of_vocabulary.py', wdir='C:/Users/xxz180012/Desktop/HW1')

  File "C:\Users\xxz180012\AppData\Local\Continuum\anaconda3\lib\site-packages\spyder_kernels\customize\spydercustomize.py",
line 827, in runfile
    execfile(filename, namespace)

  File "C:\Users\xxz180012\AppData\Local\Continuum\anaconda3\lib\site-packages\spyder_kernels\customize\spydercustomize.py",
line 110, in execfile
    exec(compile(f.read(), filename, 'exec'), namespace)

  File "C:/Users/xxz180012/Desktop/HW1/out_of_vocabulary.py", line 109, in <module>
    print(text_prob(model, sentence2))

  File "C:/Users/xxz180012/Desktop/HW1/out_of_vocabulary.py", line 73, in text_prob
    ans += math.log(model.word_prob(word, context))

  File "C:/Users/xxz180012/Desktop/HW1/out_of_vocabulary.py", line 51, in word_prob
    return self.ngram_counts[(dupWord, dupContext)]/self.context_counts[dupContext]

KeyError: ('the', ('Where', 'is'))
```

As seen from the result, we can compute the log probability of the first sentence, but now it’s smaller. This is understandable, since we removed a lot of rare words, the value will change.

As for the second sentence, we still have errors, and the cause is still because of the missing ngram, the out-of-vocabulary method is not sufficient enough to solve this problem, since “Where”, “is”, and “the” are all very common words, it’s very likely that they will not be replaced by “<unk>”.

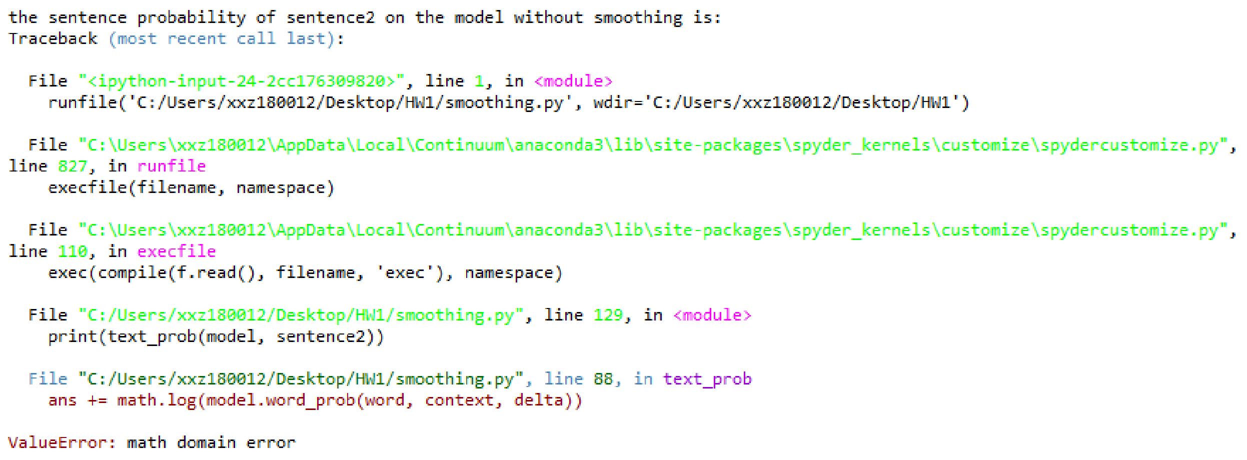
Writeup Question 2.2: How did you modify the Laplace smoothing formula? Explain why the modification was necessary.
Answer: Omit

Writeup Question 2.3: Try predicting the log probabilities of both sentences again using different values for delta. What do you get? How does the value of delta affect the predicted log probabilities? Based on these examples, do you think Laplace smoothing works well for n-gram language models? Why or why not?



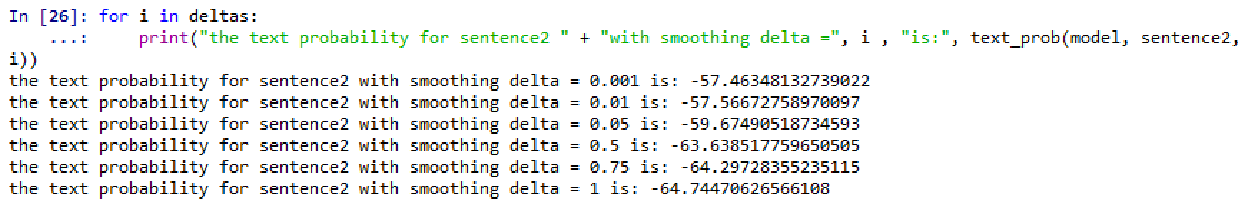
```
In [24]: runfile('C:/Users/xxz180012/Desktop/HW1/smoothing.py', wdir='C:/Users/xxz180012/Desktop/HW1')
the sentence probability of sentence1 on the model without smoothing is:
-37.35468849236516


the text probability for sentence1 with smoothing delta = 0.001 is: -49.280943303012975
the text probability for sentence1 with smoothing delta = 0.01 is: -69.10310914946496
the text probability for sentence1 with smoothing delta = 0.05 is: -86.62896506625839
the text probability for sentence1 with smoothing delta = 0.5 is: -110.51716587580954
the text probability for sentence1 with smoothing delta = 0.75 is: -113.88454399826918
the text probability for sentence1 with smoothing delta = 1 is: -116.02922787644104
```

For sentence 1, this is what I got by setting difference values of δ. We can see that the log probability of the sentence yields a negative correlation with the size of δ. Also, there are huge gaps in log probability among deltas in the range of [0, 1].



```
the sentence probability of sentence2 on the model without smoothing is:
Traceback (most recent call last):

  File "<ipython-input-24-2cc176309820>", line 1, in <module>
    runfile('C:/Users/xxz180012/Desktop/HW1/smoothing.py', wdir='C:/Users/xxz180012/Desktop/HW1')

  File "C:\Users\xxz180012\AppData\Local\Continuum\anaconda3\lib\site-packages\spyder_kernels\customize\spydercustomize.py",
line 827, in runfile
    execfile(filename, namespace)

  File "C:\Users\xxz180012\AppData\Local\Continuum\anaconda3\lib\site-packages\spyder_kernels\customize\spydercustomize.py",
line 110, in execfile
    exec(compile(f.read(), filename, 'exec'), namespace)

  File "C:/Users/xxz180012/Desktop/HW1/smoothing.py", line 129, in <module>
    print(text_prob(model, sentence2))

  File "C:/Users/xxz180012/Desktop/HW1/smoothing.py", line 88, in text_prob
    ans += math.log(model.word_prob(word, context, delta))

ValueError: math domain error
```

As before, for sentence2, I cannot compute the text probability with $\delta = 0$, since not all n-grams in the sentence are seen in the training data.



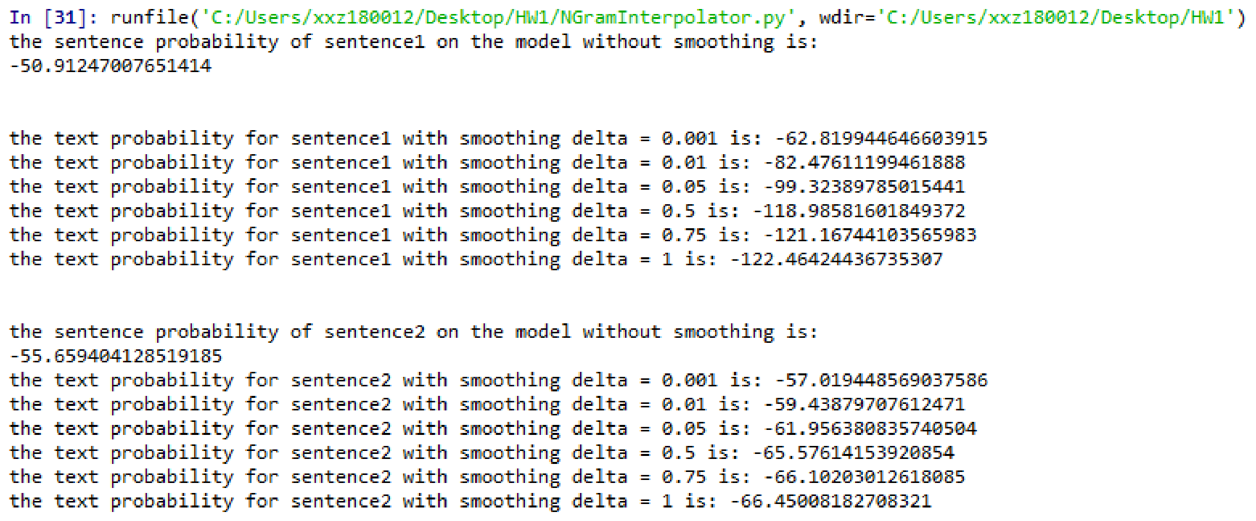
```
In [26]: for i in deltas:
    ...:     print("the text probability for sentence2 " + "with smoothing delta =", i , "is:", text_prob(model, sentence2,
i))
the text probability for sentence2 with smoothing delta = 0.001 is: -57.46348132739022
the text probability for sentence2 with smoothing delta = 0.01 is: -57.56672758970097
the text probability for sentence2 with smoothing delta = 0.05 is: -59.67490518734593
the text probability for sentence2 with smoothing delta = 0.5 is: -63.638517759650505
the text probability for sentence2 with smoothing delta = 0.75 is: -64.29728355235115
the text probability for sentence2 with smoothing delta = 1 is: -64.74470626566108
```

For $\delta > 0$, we have our values above. We can see that the decrease of log probability of sentence 2 is less aggressive than sentence 1 under same smoothing delta, presumably because for shorter sentence, the sentence yields higher possibility than the longer sentence.

From our result, we can see that the results of the log probability on each sentence relies heavily on the picking of value for δ and thus are unstable. Since Laplace smoothing assumes a uniform prior probability over all words in the vocabulary, it is unable to work well for n-gram language models.

Writeup Question 2.4: Train a trigram NGramInterpolator with lambdas = [0.33, 0.33, 0.33] and use it to predict the log probabilities of the two example sentences. What do you get? How does its compare with the base NGramLM, both with and without smoothing?

Answer:



```
In [31]: runfile('C:/Users/xxz180012/Desktop/HW1/NGramInterpolator.py', wdir='C:/Users/xxz180012/Desktop/HW1')
the sentence probability of sentence1 on the model without smoothing is:
-50.91247007651414


the text probability for sentence1 with smoothing delta = 0.001 is: -62.819944646603915
the text probability for sentence1 with smoothing delta = 0.01 is: -82.47611199461888
the text probability for sentence1 with smoothing delta = 0.05 is: -99.32389785015441
the text probability for sentence1 with smoothing delta = 0.5 is: -118.98581601849372
the text probability for sentence1 with smoothing delta = 0.75 is: -121.16744103565983
the text probability for sentence1 with smoothing delta = 1 is: -122.46424436735307


the sentence probability of sentence2 on the model without smoothing is:
-55.659404128519185
the text probability for sentence2 with smoothing delta = 0.001 is: -57.019448569037586
the text probability for sentence2 with smoothing delta = 0.01 is: -59.43879707612471
the text probability for sentence2 with smoothing delta = 0.05 is: -61.956380835740504
the text probability for sentence2 with smoothing delta = 0.5 is: -65.57614153920854
the text probability for sentence2 with smoothing delta = 0.75 is: -66.10203012618085
the text probability for sentence2 with smoothing delta = 1 is: -66.45008182708321
```

Above are the results we have by training a trigram NGramInterpolator for both sentences.

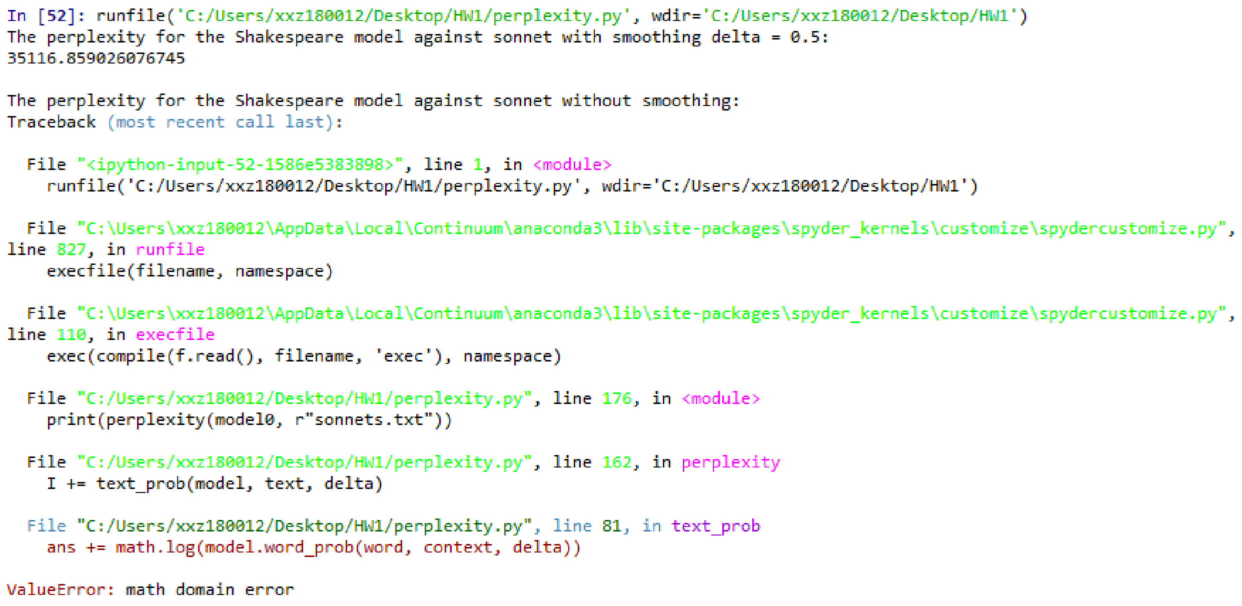
For sentence 1, we can see that the decrease of the value results still varies largely, even more largely than the base NGramLM, with the increase of δ.

For sentence 2, now we can see that the model works even without smoothing. The log probability are affected heavily by the delta as well, but not as severe as sentence 1.

We can also notice that the value of NGramInterpolator are also generally smaller than the value calculated by the base NGramLM. The reason is that now we have a monogram, a bigram, and a trigram, but the weight for each model is the same, and thus we trust each value equally, this cannot dilute the unevenness of the probability distribution.

Writeup Question 3.1: Train two trigram NGramLMs on shakespeare.txt, one with smoothing (use $delta = 0.5$) and one without. (As you have probably noticed, the NGramInterpolator is slower because it builds multiple models, so we will be using plain NGramLMs for the rest of the homework.) Evaluate the two models' perplexities using sonnets.txt as the test data. What do you get? Does anything unusual happen?
Explain what and why.

Answer:

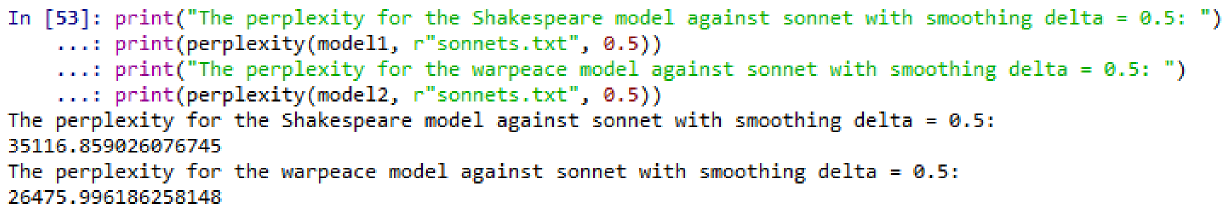


```
In [52]: runfile('C:/Users/xxz180012/Desktop/HW1/perplexity.py', wdir='C:/Users/xxz180012/Desktop/HW1')
The perplexity for the Shakespeare model against sonnet with smoothing delta = 0.5:
35116.859026076745

The perplexity for the Shakespeare model against sonnet without smoothing:
Traceback (most recent call last):

  File "<ipython-input-52-1586e5383898>", line 1, in <module>
    runfile('C:/Users/xxz180012/Desktop/HW1/perplexity.py', wdir='C:/Users/xxz180012/Desktop/HW1')

  File "C:\Users\xxz180012\AppData\Local\Continuum\anaconda3\lib\site-packages\spyder_kernels\customize\spydercustomize.py",
line 827, in runfile
    execfile(filename, namespace)

  File "C:\Users\xxz180012\AppData\Local\Continuum\anaconda3\lib\site-packages\spyder_kernels\customize\spydercustomize.py",
line 110, in execfile
    exec(compile(f.read(), filename, 'exec'), namespace)

  File "C:/Users/xxz180012/Desktop/HW1/perplexity.py", line 176, in <module>
    print(perplexity(model0, r"sonnets.txt"))

  File "C:/Users/xxz180012/Desktop/HW1/perplexity.py", line 162, in perplexity
    I += text_prob(model, text, delta)

  File "C:/Users/xxz180012/Desktop/HW1/perplexity.py", line 81, in text_prob
    ans += math.log(model.word_prob(word, context, delta))

ValueError: math domain error
```

From the results we can see that, with smoothing, the model compute the perplexity with the value 35116.859026076745, however, without smoothing, we have an error, and the reason is the same as before, that there are trigrams that are unseen in the model's ngram_counts.

Writeup Question 3.2: Evaluate the perplexities of a smoothed ($delta = 0.5$) tri-gram NGramLM trained on *shakespeare.txt* and one trained on *warpeace.txt*. Use sonnets.txt as the test data for both. What do you get? Which one performs better, and why do you think that's the case?

Answer:



```
In [53]: print("The perplexity for the Shakespeare model against sonnet with smoothing delta = 0.5: ")
    ...: print(perplexity(model1, r"sonnets.txt", 0.5))
    ...: print("The perplexity for the warpeace model against sonnet with smoothing delta = 0.5: ")
    ...: print(perplexity(model2, r"sonnets.txt", 0.5))
The perplexity for the Shakespeare model against sonnet with smoothing delta = 0.5:
35116.859026076745
The perplexity for the warpeace model against sonnet with smoothing delta = 0.5:
26475.996186258148
```

From the results we can see that warpeace model performs better. This is against the common sense, after all, since both were written by Shakespeare it should be the Shakespeare model outperforming warpeace model.

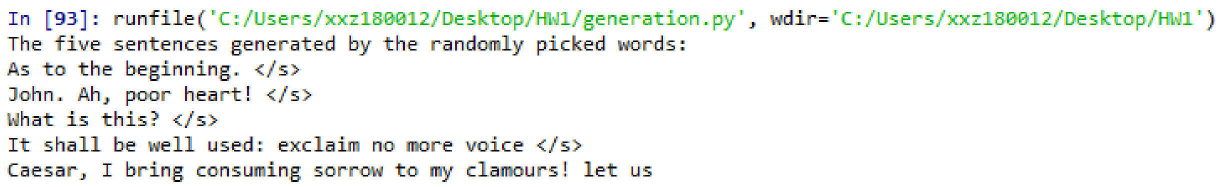
However, the sonnets are very different in structure and word-picking from plays, it’s possibly that the perplexity is high between the two pieces of works even if they were written by the same author.

Writeup Question 3.3: Authorship identification is an important task in NLP. Can you think of a way to use language models to determine who wrote an unknown piece of text? Explain your idea and how it would work (you don't need to implement it).

Answer:

We can use the n-gram models (NGramLM, NGramInterpolator, etc) to train some models with different authors’ works. Then run the perplexity against the unknown piece of text with those models, and compare the values. The one with the best values could be considered as the most possible candidate author who wrote the text.

Writeup Question 4.1: Train a trigram model on shakespeare.txt and generate 5 sentences with max length = 10. What did you generate? Are they good (Elizabethan) English sentences? What are some problems you see with the generated sentences?



```
In [93]: runfile('C:/Users/xxz180012/Desktop/HW1/generation.py', wdir='C:/Users/xxz180012/Desktop/HW1')
The five sentences generated by the randomly picked words:
As to the beginning. </s>
John. Ah, poor heart! </s>
What is this? </s>
It shall be well used: exclaim no more voice </s>
Caesar, I bring consuming sorrow to my clamours! let us
```

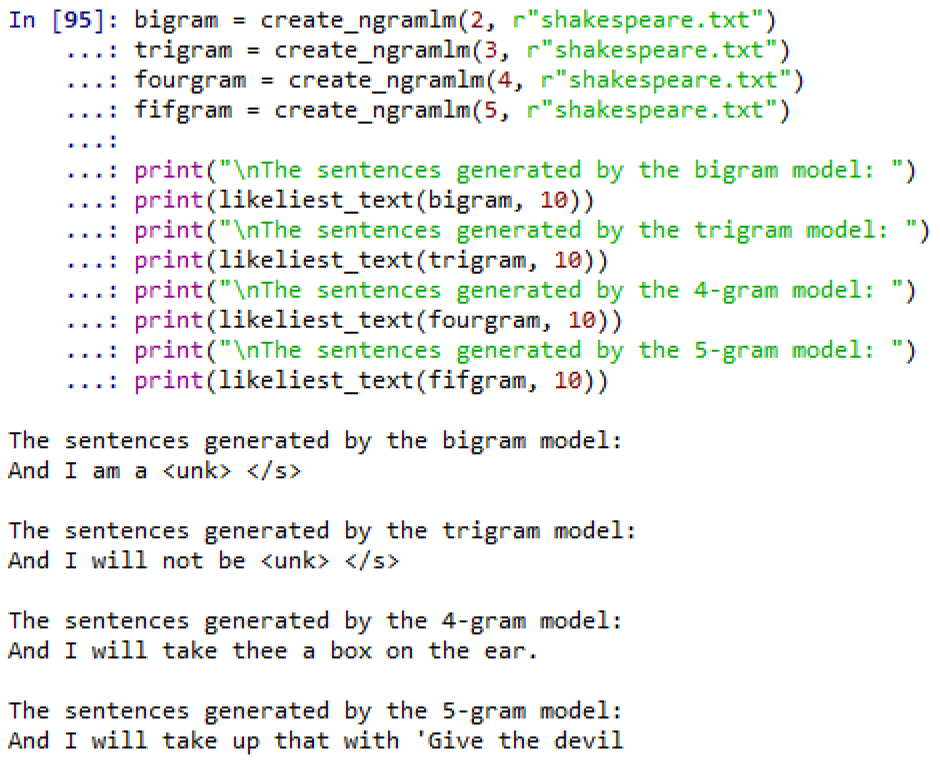
Answer:

Above are the sentences we created, as we can see, the first 4 sentences ends because we meet the “</s>”, surprisingly, these sentence were decent enough, although the 4th sentence does not make too much sense. The last sentence ends because it reaches the length limit and the sentence itself seems unfinished, the first part of the sentence was well generated, and also grammar correct.

The major problems of the sentences generated randomly, though, are that they very often ends quickly, most sentences didn't reach the maximum length limit of 10 words. Moreover, all those generated sentences are not really similar, the model can basically generate any sentence from the vocabulary and it will not automatically pick the most possible one.

Writeup Question 4.2: Train four models-one bigram, one trigram, one 4-gram, and one 5-gram-on shakespeare.txt and generate the likeliest sentence for each one using max length = 10. What did you generate? Do you notice anything about these sentences? How do they compare to each other? How do they compare to the randomly-generated sentences?

Answer:



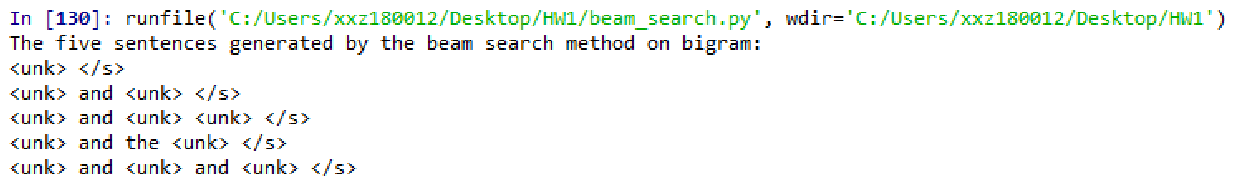
```
In [95]: bigram = create_ngramlm(2, r"shakespeare.txt")
    ...: trigram = create_ngramlm(3, r"shakespeare.txt")
    ...: fourgram = create_ngramlm(4, r"shakespeare.txt")
    ...: fifgram = create_ngramlm(5, r"shakespeare.txt")
    ...:
    ...: print("\nThe sentences generated by the bigram model: ")
    ...: print(likeliest_text(bigram, 10))
    ...: print("\nThe sentences generated by the trigram model: ")
    ...: print(likeliest_text(trigram, 10))
    ...: print("\nThe sentences generated by the 4-gram model: ")
    ...: print(likeliest_text(fourgram, 10))
    ...: print("\nThe sentences generated by the 5-gram model: ")
    ...: print(likeliest_text(fifgram, 10))

The sentences generated by the bigram model:
And I am a <unk> </s>

The sentences generated by the trigram model:
And I will not be <unk> </s>

The sentences generated by the 4-gram model:
And I will take thee a box on the ear.

The sentences generated by the 5-gram model:
And I will take up that with 'Give the devil
```

As shown in above, the first couple words of each sentence are always the same. From the qualities of the sentences, we might be able to summarize that the larger the n is, the more likely the sentence would be well organized and meaningful. Compared with randomly-generated sentences, the sentence generated with this method is more likely to have identical structure, if we increase the size of the n-gram, we are more likely to generate a structurally and semantically correct sentence, but the sentence will share similarity with the ones that are generated by smaller n-grams, unlike the randomly picked sentences, which are generated merely by randomness.

Writeup Question 5.1: How long did this homework take you to complete (not counting extra credit)?
Answer: it took me around 9 hours in total (6 hours to write the codes plus the writeup questions for the homework parts, and another 3 hours to modify and comment my codes) to complete the homework

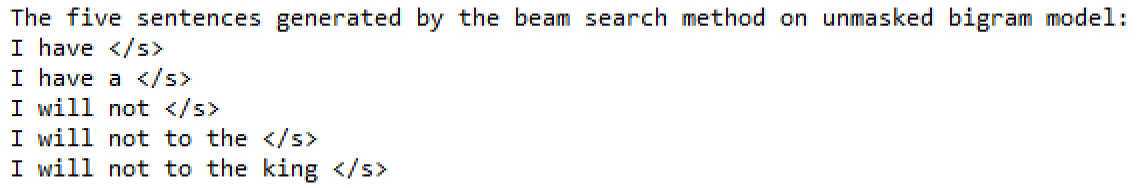
Writeup Question 5.2: Did you discuss this homework with anyone?
Answer: yes, I discussed the homework, specifically the first part, with Hongzheng Wang

Writeup Question 6.1b: For beam search, generate 5 sentences with a trigram model trained on shakespeare.txt and max length = 10, using k = 5, and put the generated sentences in your writeup. How do they compare to the generated sentences from Questions 4.1 and 4.2?
Answer:
We use $\delta = 0.5$



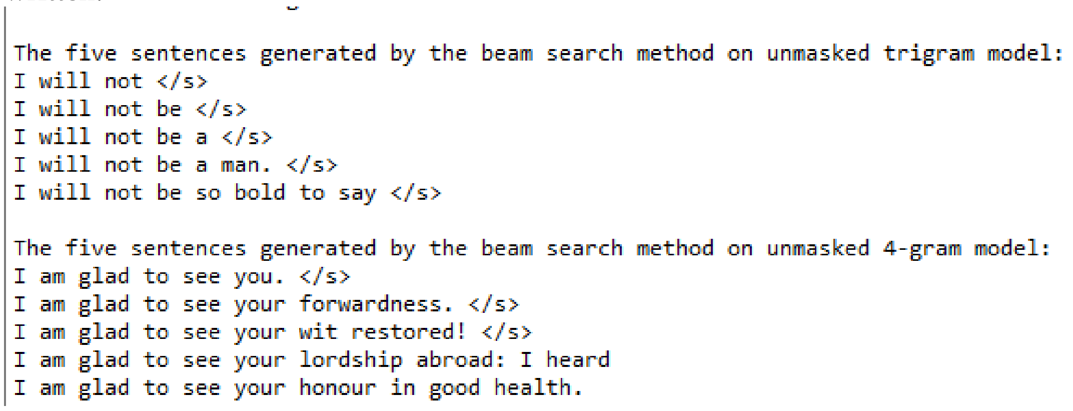
```
In [130]: runfile('C:/Users/xxz180012/Desktop/HW1/beam_search.py', wdir='C:/Users/xxz180012/Desktop/HW1')
The five sentences generated by the beam search method on bigram:
<unk> </s>
<unk> and <unk> </s>
<unk> and <unk> <unk> </s>
<unk> and the <unk> </s>
<unk> and <unk> and <unk> </s>
```

Above are the 5 sentences that with the highest log probabilities. As we can see, those sentences also share a very similar pattern. The token "<unk>" occurred very often in the sentences and thus shortened the length of the sentence. The problem might be solved if we do not mask the rare words.



```
The five sentences generated by the beam search method on unmasked bigram model:
I have </s>
I have a </s>
I will not </s>
I will not to the </s>
I will not to the king </s>
```

We can see that the unmask version of the bigram model still perform not as good as the likeliest model. However, if we increase the size of the n-gram, the sentence will be more and more well written:



```
The five sentences generated by the beam search method on unmasked trigram model:
I will not </s>
I will not be </s>
I will not be a </s>
I will not be a man. </s>
I will not be so bold to say </s>

The five sentences generated by the beam search method on unmasked 4-gram model:
I am glad to see you. </s>
I am glad to see your forwardness. </s>
I am glad to see your wit restored! </s>
I am glad to see your lordship abroad: I heard
I am glad to see your honour in good health.
```