Homework 6 report
Sanad Saha

Evaluating Entropy (2 pts)

1. Like EX2, please first evaluate the trigram entropies (base and large) on EX2 test.txt
Q: Is the large version intuitively better than the small version? If so, why? If not, why?

Answer:
I've evaluated the trigram entropy for both base and large on test.txt from EX2 using the following command.

```
cat test.txt | sed -e 's/ /_/g;s/\(.\)/\1 /g' | awk '{printf("<s> %s </s>\n", $0)}' | carmel -sribI trigram.(base/large).wfsa.norm
```

For base trigram the entropy was 2.9148 and for the large trigram wfsa it was 2.93042.
Here the large trigram version is performing poorly than the smaller version because the large trigram model was trained on wiki-2 corpus, which contains lots of lines outside of train.txt, which by a small margin impacts the entropy on test.txt; which is related to the train.txt

2. Now write a short program to evaluate the entropy of NLMs on the same test.txt.
Hint: they should be around 2.6 and 2.0, respectively.
Q: Which version (base or large) is better? Why?

Answer:
My eval_entropy.py calculates the entropy of the test.txt using the NLMs. Using the large version we get entropy of 1.9726965712505402 and using the base version we get entropy of 2.692813611492875

I used the following command to run the eval_entropy.py

```
cat test.txt | /scratch/anaconda3/bin/python3 eval_entropy.py
```

The large version of the NLM is performing significantly better than the base version. As the large version was trained on large corpus of wiki-2; the large n-gram from NLM has both seen more cases and probabilities calculated are more accurate.

3. Is NLM better than trigram in terms of entropy? Does it make sense?

Answer:
Yes, NLM is better than trigram in terms of entropy. This makes perfect sense because NLM has a n-gram language model, which is more accurate in predicting the next letter. Generally in English language it's easier to effectively predict the next letter when you have better model than trigrams (3-gram). The higher the n, in the n-gram language model, it gives better prediction.

3 Random Generation (2 pts)

1. Random generation from n-gram models.

Use carmel -GI 10 <your_wfsa> to stochastically generate character sequences. Show the results. Do these results make sense?

Answer:

Using the base trigram model, the following sequences were stochastically generated. I have added (char_seq_trigram.txt) which contains the results in a .txt file. The command is given below:

carmel -GI 10 trigram.base.wfsa.norm > gen_seq_trigram.txt

```
pelican04 ~/Courses/CS-539/hw6-data/nlm 161% carmel -GI 10 trigram.base.wfsa.norm


Using random seed -R 1659018150
<s> a n d f o r e _ d o _ t o _ b y _ t h a s _ e x p e r s _ o f _ c o f f e d w a n _ i t e n t h e _ r e c a n _ m e
u r t _ w a l _ a l l y i n </s> e^-145.267935669945
<s> s h o s _ b y _ t h e _ o f f e v i s t _ w r _ h e _ i n _ s t a n d _ r e _ p r e _ d i a n d s _ g r e _ v o u </
s> e^-114.956481700927
<s> t _ t o _ i m p a n _ t h e r c h o b _ t o _ t h e s _ i s _ d o c a l l _ c o n _ o n d _ c a l a n _ a n y _ d o
e s c o t _ t r i n _ t h e _ t h e _ m u s p e r s h i t _ h a t t y _ i t h r i p s e c t h e a t _ o f _ t h e c t o
f _ a n _ a _ m y _ j o n _ d u c t o l l y _ b y _ f o r _ l i e s t _ s a w i t _ a m e _ f o n </s> e^-328.9276132136
17
<s> a t i f t i o n e s </s> 1.47113912911454e-11
<s> t h e _ g r o w n _ _ a n d _ b l e d _ i n s _ v e r e v i o n d _ t o _ p r o s t r a c e _ s u r e _ c a l _ p a
r m _ a l l _ b e e p _ i n t _ s a p _ t h o e s </s> e^-165.620802753212
<s> r e _ r i p s e _ i d e r e _ a r i a _ g a r e s _ b a n d _ b e f t _ i s t a g </s> e^-96.281934071179
<s> v i r _ k i c i s _ i d _ o f _ a n d _ a _ b e e t _ w a l l _ s o _ c a f t i g a r e g a m i s t r o d _ y o c c
o m e _ b a u m _ t h i s _ w i t h e _ d e _ b a d _ a r e _ m e m o r i n s p i e s _ t h e _ b e n d _ t h e _ w h _
f r o u n d _ a t _ t h e _ s e _ i s _ o n l y _ i s _ f o r k e _ i t c h o m _ t o _ e n t a r i n e _ i n g a m _ a
r h o l l _ c u r _ d r o v e _ i n d _ t a n y _ a n _ h i l d _ w i t _ s t r o b v i c a r e d _ n e _ a l l _ w h e
a _ s t y _ u n w o _ r i t e r _ c e s _ t h e v e </s> e^-534.064279710993
<s> t h e _ u n c l i v i n g s _ f r o d _ t h o _ d e n _ w i l l _ t r a g a t s _ t h e i v a n i c i n g _ t o l f
</s> e^-118.312238445224
<s> w e e k _ f i n d _ i n _ t h e r _ d o d u c t e _ r a l s o m _ a n d _ c o u n d _ l a s _ a p p o e s </s> e^-99
.4168337324265
<s> r e l p _ o m _ s p o c a t r u e _ d o e s </s> 1.2206204177118e-26
```

The results makes sense. As we have done the same experiment in EX2, we know that trigram model generates some English like words but they are not actually correct English. Small English words such as week, find, the, for, and etc. are correctly generated; but the trigram model is not good enough to generate English words having more than 4 letters, let alone English sequence.

2. Write a short code to randomly generate 10 sentences (from <s> to </s>) from NLMs.

Hint: use random.choices() to draw the random sample from a distribution.

Answer:

My (gen_seq.py) generates 10 sentences using NLMs. I've provided 2 files char_seq_ngram_base.txt and char_seq_ngram_large.txt, which contains 10 sentences generated using base, large NLM model respectively.

random.choices() was used to draw random samples from the distribution. The following screenshot shows some sentences generated using gen_seq.py.

```
pelican04 ~/Courses/CS-539/hw6-data/nlm 163% /scratch/anaconda3/bin/python3 gen_seq.py
<s> s t r i c t _ a n d _ t h e _ t h e _ d i s s i l e d _ a n d _ a l l _ o n _ t h e _ m o v e _ t h e _ c o m p l e
t e s _ a n d _ t h e _ p e l s e d _ a _ b a d _ m a n y _ h a d _ s t r o l d a n _ w e r e _ t o _ t h e _ p r o m a
s s i o n _ a n d _ o n e _ e x p e r t i o n _ t o _ f e l l _ b e a n _ t o _ l i k e _ a n d _ d o e t _ t h e _ r e
c o n t e r s _ i n _ t h e _ p r o m a t i o n s _ a n d _ w h e n _ t h e _ i n t i v e s _ w i t h _ t h e _ s t o r
i e s _ t o _ t h e _ b r e g e r _ p a r t s _ p r u m e n t s _ t o _ r e p o u t _ a _ w e r e _ o f _ t h e _ s c o
m p i l i e s _ t h e _ w i l l _ m a y _ y o u _ c o u r s e _ a l l o r i e </s>

<s> t h e r e _ h i s _ h e _ t h e _ p o l i t i c e _ a n d _ o n _ t h e _ m o r e _ f o r _ t h e _ n e w _ c o m p
l a t e _ t h e _ s t a r t _ i n _ t h e _ p r o p e r i c a n _ l a l l e d _ t h e _ c o m e n i n g _ h i t h _ t h
e _ m o r e _ d e c o n d e s t e r s _ o f _ m o s t _ b e _ m a r k e _ t h e _ n o t _ g e t _ i n _ a _ p r e v e r
y _ a n d _ t h e _ w a s _ t h e _ m a r k e r _ f r o m _ t h e _ t h e _ b i t _ o f _ t h e _ b a d _ m a n y _ c o
n t e r i a n s _ t o _ b e _ p o l i t i c e _ a n d _ a s t i n g _ t h e _ c o m p e s e d _ a n d _ s i m p e n _ t
h e _ s i n d e r _ t o _ t h e _ n o t _ w h o _ g o t _ s t r a t e s _ t o _ a l l o u s _ s e e t _ s t a r d _ t h
e _ d i s t a t e _ a n d _ c o n t e r t i c a _ h a s _ a n g e r e _ t o _ u n i c e _ t h a t _ w i t h _ h a l l _
a n d _ t h e _ m o s t _ i n _ t h e _ p r o p e r e d _ t h e _ a n d _ k e r t i o n _ i n _ t h e _ g r o w _ t h e
_ b e t i o n e r _ d i s t r a t e _ b e _ a r e _ t r e s t _ w i t h _ b e t o r e _ o f _ t h e _ p l o s e s _ t h
e _ r e a d _ t h e _ b e s t _ i n _ a _ c o m p e r i e s _ w i t h _ t h e _ s t a m e _ o f _ t h e _ p o l a n _ s
u n d _ t h e _ s t i t n _ o f _ s h i c k _ t h e _ c o n d e d _ w a n t _ t h e _ m o r e _ t h e _ b e c e n s _ o
f _ s e a s e d _ a _ c o m e n </s>

<s> w h i n g e s _ t o _ f o r _ t h e _ p r o v e s t e d _ w h e n _ t h e _ r a d e _ i n _ t h e _ m i s t _ a n d
_ g o o d _ d e a n t _ t h e _ s t r i k e _ s o n o r e d _ t h e _ a r e _ t h e _ r e a l _ t h e _ s t e p t _ t h
e _ i n d e r e d _ i n _ t h e _ u r e r _ p l a n t _ t o _ g e t _ s t r i n e _ t o _ b e _ a r e _ h a s _ m a n _
i _ c o m p l i c i o n s </s>
```

3. Compare the results between NLMs and trigrams.

Answer:
If we look closely the generated sentences between NLM and trigrams, we can see that NLM does exceedingly better job in generating English like sentences. Although the generated sentences are not grammatically perfect, but there are far more correctly spelled bigger words than the trigram model. For example: strict, there, strike etc. words were not possible to generate for a trigram model.

4
Restoring Spaces (4 pts)
1. Recall from EX2 that you can use LMs to recover spaces:

therestcanbeatotalmessandyoucanstillreaditwithoutaproblem
thisisbecausethehumanminddoesnotreadeveryletterbyitselfbutthewordasawhole.

First, redo the trigram experiments using our provided trigrams, and using Carmel.
What are the precisions, recalls, and F-scores? (use eval_space.py).

Answer:

I have redone the recover space experiment of EX2, using both of our large and base trigram models in Carmel.
Carmel returned sentences that had <s>, </s> as starting and ending symbol and _ were used instead of spaces, with each character being separated by spaces. So, I used the following command to convert the results before evaluating with the test.txt file.

```
cat space_restored_large.txt | sed -e 's/ //g;s/_/ /g;s/<s>//g;s/<\/s>//g' > space_restored_test.trigram.large
```

Using the base trigram model we achieve the following scores:
Precision = 60.8% Recall = 59.8% F1-score = 60.3%

```
atukan ~/Courses/CS-539/hw6-data 172% python eval_space.py test.txt space_restored_test.trigram.base
recall= 0.598 precision= 0.608 F1= 0.603
```

And using the large trigram model the following scores were achieved:
Precision = 65.2% Recall = 62.5% F1-score = 63.8%

```
atukan ~/Courses/CS-539/hw6-data 173% python eval_space.py test.txt space_restored_test.trigram.large
recall= 0.625 precision= 0.652 F1= 0.638
```

2. Then design an algorithm to recover spaces using NLMs. Note: you can't use dynamic programming any more due to the infinite history that NLM remembers. You can use beam search instead. Describe your algorithm in English and pseudocode, and analyze the complexity.

Answer:
The algorithm to recover spaces using NLM is described below using pseudocode. For each line, we would conduct Beam search to retrieve missing spaces for that line.

First we load our language model into h and initialize our Beam (beam) which doesn't have any entry at the beginning.

Pseudocode and description:

```
step1:

    for each line in the given file:
        h = load the language model
        beam = [(0, h)] (Initialize the beam)

Step2:
```

Now for each character in the given line, we have 2 decisions to make,

```
        for each character in the current line:

            temp = [] (Temporary Beam)
Step3:
```

For top K-best answer in the beam in the previous step

```
            for (p, h) in beam:
```

1. We can add a space and then add the current character c.

```
                prob1 = p + math.log(h.next_prob("_"))
                h1 = h + "_"
                prob1 += math.log(h1.next_prob(c))
```

```
h1 = h1 + c
temp.append((prob1, h1))
```

or,

2. We can add the current character c without adding space.

```
prob2 = p + math.log(h.next_prob(c))
h2 = h + c
temp.append((prob2, h2))
```

We calculate probability and update the new language model given these two possible decisions are made. Then we add these to our temporary beam. This temporary beam holds all possible scenarios where spaces can be added, alongside their probability.

`Step4:`
Now holding all possible scenarios of adding spaces, is not efficient. We can discard those with low probabilities of occuring. So, we update our original Beam with the top 20 results from the temporary beam.

```
beam = sorted(temp, reverse=True)[:beam_len]
```

After, continuing step 3 until the end of the line, we finally get our top 20 candidates lines with spaces. And as we are sorting our beam at each step, the 1st answer in the Beam is the sentence which has the best probability of occuring with spaces.

```
p, h = beam[0]
```

We retrieve that and print the history of that sentence's language model, which gives us the line with restored spaces.

```
print(("".join(h.history)).replace("_"," ").replace("<s>",""))
```

The above algorithm has a complexity of O(n * m * b * 2),
where n is the number of lines, m is the number of characters per line, b is the beam size, and for the 2 decisions we multiply it by 2.
As, 2 is a constant factor we can write the complexity as O(n * m * b)

3. Implement it, and report the precision, recalls, and F-scores.

Answer:
Our implementation of the algorithm (restore_spaces.py) described in part 2, achieves the following scores [using Beam size 20]:
Recall = 82.7% Precision = 80.0% F1-score = 81.3%

```
atukan ~/Courses/CS-539/hw6-data 188% python eval_space.py test.txt spaces_restored_base.txt
recall= 0.827 precision= 0.800 F1= 0.813
```

Recall = 96.7% Precision = 95.1% F1-score = 95.9%

```
atukan ~/Courses/CS-539/hw6-data 189% python eval_space.py test.txt spaces_restored_large.txt
recall= 0.967 precision= 0.951 F1= 0.959
```

5

Restoring vowels (4 pts)

1. Redo trigram vowel recovery and report the accuracy. (use eval_vowels.py)

Answer:

I used the following command to get restore vowel using the given trigram models and carmel.

```
cat test.txt.novowels | sed -e 's/ /_/g;s/\(.\)/\1 /g' | awk '{printf("<s> %s </s>\n", $0)}' | carmel -sribIEWk 1 trigram.large.wfsa.norm remove-vowels.fst
```

then I used the following command to remove _, <s>, </s>, and extra spaces before evaluating the restored file using eval_vowels.py

```
cat vowel_restored.trigram.base | sed -e 's/ //g;s/_/ /g;s/<s>//g;s/<\/s>//g' > vowel_restored_test.trigram.base
```

Using the base trigram model, we got accuracy of 42.6% and using the large trigram model 41.0% accuracy.

```
pelican02 ~/Courses/CS-539/hw6-data 198% python eval_vowels.py test.txt vowel_restored_test.trigram.large
word acc= 0.410
```

```
pelican02 ~/Courses/CS-539/hw6-data 199% python eval_vowels.py test.txt vowel_restored_test.trigram.base
word acc= 0.426
```

2. Now design an algorithm to recover spaces using NLMs.
Describe your algorithm in English and pseudocode, and analyze the complexity.

Answer:

The algorithm for recovering spaces is described below:

Pseudocode and descriptions:

Step1:

```
vowel = ['a', 'e', 'i', 'o', 'u']

for i in vowel:
    perm.append([i])
for i in vowel:
```

```
    for j in vowel:
        perm.append([i, j])
for i in vowel:
    for j in vowel:
        for k in vowel:
            perm.append([i, j, k])
```

In step1, we generate all possible combinations of vowels, upto 3 consecutive vowels and store them in a list (perm)
When we are running our algorithm, we need to search for the missing vowel combination. So, we'll be using the perm list.

step2:

```
    for each line in the given file:
        h = load the language model
        beam = [(0, h)] (#Initialize the Beam)

        for each character c in the current line:
            if c == ' ': (#Replacing the spaces with _)
                c = '_'

            temp = [] (#initializing temporary beam)
```

```
Step3:
```
For top K-best answer stored in the beam, in the previous step

```
            for (p, h) in beam:
```

We have several options here,

1. We can ignore adding any vowels and just add the current character

```
prob = p + math.log(h.next_prob(c))
h_temp = h + c
temp.append((prob, h_temp))
```

2. We can add vowel before the current character. We have stored all possible vowels (upto 3 consecutive) in the list (perm). So, we would add each one of those to our top K-beams from the previous step and compute the new language model and probability of occurrence. As the probabilities can be very small, we are using log to avoid underflow.

```
for permutes in perm:
    h_temp = h + ""
    prob = p
```

```
                    for j in permutes:
                        prob += math.log(h_temp.next_prob(j))
                        h_temp += j

                    prob += math.log(h_temp.next_prob(c))
                    h_temp += c
                    temp.append((prob, h_temp))
```

Step4:

Now holding all possible scenarios of adding spaces, is not efficient. We can discard those with low probabilities of occuring. So, we update our original Beam with the top 20 results from the temporary beam.

```
beam = sorted(temp, reverse=True)[:beam_len]
```

After, continuing step 3 until the end of the line, we finally get our top 40 [beam_len] candidates lines with spaces. And as we are sorting our beam at each step, the 1st answer in the Beam is the sentence which has the best probability of occuring with spaces.

```
p, h = beam[0]
```

We retrieve that and print the history of that sentence's language model, which gives us the line with restored spaces.

```
print(("".join(h.history)).replace("_", " ").replace("<s>", ""))
```

The above algorithm has a complexity of O (n * m * b * v),
where n is the number of lines, m is the number of characters per line, b is the beam size, and v is the number of candidate vowels.
V can change according to the number of vowel candidate, In our implementation we used maximum of 2 consecutive vowels as candidate. Because considering 3 consecutive vowels would add another 125 candidates, and it was taking a lot of time to train in pelican.

So, the complexity of the aforementioned algorithm is O (n * m * b * v)

3. Implement it, and report the Precisions, Recalls, and F-scores.

Answer:

The implementation of the algorithm described above for recovering vowel: restore_vowels.py is attached with this report.
For the experiment, only 30 vowel candidate was used (upto 2 consecutive vowel combinations). As the training time of the models (with 3 consecutive vowel combinations) was unexpectedly long in Pelican, I wasn't able to complete it. Nonetheless using 1, and 2 consecutive vowel candidates, give close to the intended accuracy.

I got accuracy of 52.3% using the Base NLM.

```
pelican04 ~/Courses/CS-539/hw6-data 169% python eval_vowels.py test.txt vowels_restored_base.txt
word acc= 0.523
```

Using the Large NLM I got 76.5% accuracy.

```
pelican04 ~/Courses/CS-539/hw6-data 170% python eval_vowels.py test.txt restore_vowels_large.txt
word acc= 0.765
```

I could have achieved 54% and 80% accuracy respectively if 3 consecutive vowel combination could be used. Because in the test.txt file there were words that have 3 consecutive vowels [e.g. [**'i'**, **'i'**, **'o'**], [**'i'**, **'o'**, **'u'**], [**'e'**, **'o'**, **'u'**], [**'e'**, **'e'**, **'i'**]], which explains the small reduction in accuracy.

In this vowel, restoration problem F1 score, recall and precision are all the same, so they weren't reported individually. I've added both the recovered file with this report.