Instructions

1. This homework has two parts: Q1, Q2 and Q3 are written questions and Q4 is a programming assignment with some parts requiring a written answer. Each part needs to be submitted as follows:

   - Submit the answers to the written questions as a `pdf` file on Canvas for the assignment corresponding to Homework 3 Written. This should consist answers to Q1, Q2, Q3 and the descriptive answers from Q4. Name the pdf file as- `LastName_FirstName.pdf`. We recommend students type answers with LaTeX or word processors for this part. A scanned handwritten copy would also be accepted, try to be clear as much as possible. No credit may be given to unreadable handwriting.
   - The programming assignment requires you to work on boilerplate code. Submit the code to the programming assignment in a zip that contain `ngram.ipynb`, `rnn.ipynb` `hw3_skeleton_char.py` and `hw3_skeleton_word.py`. This submission is to be made on Canvas for the assignment corresponding to Homework 3 Programming. Name the zip file as- `LastName_FirstName.zip`.
   - You first need to join this course on Gradescope. Please use your actual name as Gradescope username. Entry Code for CS 4650 is `9P32V6`. Entry Code for CS 7650 is `97ZWYV`. You should also submit a zip file to Gradescope hw3 Language Models. This file should contain `hw3_skeleton_char.py` and `hw3_skeleton_word.py`. You can see your score for 4 (a) as soon as you submit it.

2. For the written questions, write out all steps required to find the solutions so that partial credit may be awarded.

3. We generally encourage collaboration with other students. You may discuss the questions and potential directions for solving them with another student. However, you need to write your own solutions and code separately, and not as a group activity. Please list the students you collaborated with.

4. The code files needed to complete the homework are included in a zip file on Canvas.

1. The Kneser-Ney smoothing method approximates the probability of an n-gram that has not been seen using the likelihood of the (n-1)-gram to occur in diverse contexts. If we want to finish the sentence "I want to go to the movie ____" with the word "theatre" but we have not observed "theatre" very often, then we want to make sure that we can guess "theatre" based on its likelihood of combining with other words.

   The full formula for the bigram version of Kneser-Ney smoothing follows:

$$P(bigram) = \text{discounted bigram probability} + \text{joint unigram probability} \quad (1)$$

$$P(w_i|w_{i-1}) = \frac{max(C(w_{i-1}, w_i) - d, 0)}{C(w_{i-1})} + \lambda(w_{i-1}) \frac{|v : C(v, w_i) > 0|}{\sum_{w'} |v : C(v, w') > 0|} \quad (2)$$

$$\lambda(w_{i-1}) = \frac{d}{\sum_v C(w_{i-1}, v)} * |w : C(w_{i-1}w) > 0| \quad (3)$$

   Assume that you have collected the data in the following tables, and assume that all other observed counts are 0. In the bigram table, rows represent w_{i-1}, columns represent w_i: e.g. $C(\text{computer}, \text{keyboard}) = 2$.

$C(w_{i-1}, w_i)$	computer	keyboard	monitor	store
computer	0	2	4	4
keyboard	1	0	0	1
monitor	0	1	1	1
store	2	0	0	0

Table 1: Bigram frequency. Rows = w_{i-1}, columns = w_i.

computer	10
keyboard	3
monitor	6
store	5

Table 2: Unigram frequency.

   Consider the following sentence fragment S: "I shopped at the computer ____".
   You need to determine whether the sentence is more likely to end with "computer store" or "computer monitor."

   (a) Compute the raw bigram probabilities for the candidate words {*store*, *monitor*} to complete the sentence S, i.e. $P(\text{store}|\text{computer})$ and $P(\text{monitor}|\text{computer})$. Is one word more likely than the other, and if so which one? [2 pts]

   (b) Compute the Kneser-Ney smoothed bigram probability of the candidate words {*store*, *monitor*} to complete the sentence. Use $d = 0.5$ as the discount term. Is one word more likely than the other, and if so which one? If the result has changed, why do you think it changed? [5 pts]

(c) Change the discount term to $d = 0.1$ and re-compute the Kneser-Ney smoothed bigram probability of the candidate words $\{$*store*, *monitor*$\}$ to complete the sentence. Is one word more likely than the other, and if so which one? If the result has changed, why do you think it changed? [3 pts]

Solution.

NOTE: I AM USING 2 LATE DAYS FOR THIS PROJECT.

a)

$P(\text{store}|\text{computer}) = \frac{C(\text{computer,store})}{C(\text{computer})} = \frac{4}{10} = \mathbf{0.4}$

$P(\text{monitor}|\text{computer}) = \frac{C(\text{computer,monitor})}{C(\text{computer})} = \frac{4}{10} = \mathbf{0.4}$

No, both words are equally likely because they have the same counts.

b)

Let us calculate some constants beforehand that are useful in both calculations below:

$\sum_v C(\text{computer}, v)$

$= C(\text{computer, computer}) + C(\text{computer, keyboard}) + C(\text{computer, monitor}) + C(\text{computer, store})$

$= 0 + 2 + 4 + 4$

$= 10$

$|w : C(\text{computer}, w) > 0|$

$= |\{\text{keyboard, monitor, store}\}|$

$= 3$

$\sum_{w'} |v : C(v, w') > 0|$

$= |v : C(v, \text{computer}) > 0| + C(v, \text{keyboard}) > 0| + C(v, \text{monitor}) > 0| + C(v, \text{store}) > 0|$

$= |\{\text{keyboard, store}\}| + |\{\text{computer, monitor}\}| + |\{\text{computer, monitor}\}| + |\{\text{computer, keyboard, monitor}\}|$

$= 2 + 2 + 2 + 3 = 9$

Now, we can calculate the actual smoothed bigram probabilities:

$P(\text{store}|\text{computer})$

$= \frac{\max(C(\text{computer,store})-d,0)}{C(\text{computer})} + \lambda(\text{computer}) \frac{|v:C(v,\text{store})>0|}{\sum_{w'} |v:C(v,w')>0|}$

$= \frac{\max(C(\text{computer,store})-0.5,0)}{C(\text{computer})} + \frac{0.5|w:C(\text{computer},w)>0|}{\sum_v C(\text{computer},v)} \frac{|v:C(v,\text{store})>0|}{\sum_{w'} |v:C(v,w')>0|}$

$= \frac{3.5}{10} + \frac{0.5(3)}{10}\frac{3}{9}$

$= 0.35 + 0.05$

$= \mathbf{0.4}$

$P(\text{monitor}|\text{computer})$

$= \frac{\max(C(\text{computer,monitor})-d,0)}{C(\text{computer})} + \lambda(\text{computer})\frac{|v:C(v,\text{monitor})>0|}{\sum_{w'}|v:C(v,w')>0|}$

$= \frac{\max(C(\text{computer,monitor})-0.5,0)}{C(\text{computer})} + \frac{0.5|w:C(\text{computer},w)>0|}{\sum_v C(\text{computer},v)}\frac{|v:C(v,\text{monitor})>0|}{\sum_{w'}|v:C(v,w')>0|}$

$= \frac{3.5}{10} + \frac{0.5(3)}{10}\frac{2}{9}$

$= 0.35 + 0.0333$

$= \mathbf{0.3833}$

We can now see "store" is now more likely to appear than "monitor". This is because "store" appears in 3 bigrams compared to the 2 bigrams for "monitor", so it is more likely to appear in an unseen bigram in general.

c)

$P(\text{store}|\text{computer})$

$= \frac{\max(C(\text{computer,store})-d,0)}{C(\text{computer})} + \lambda(\text{computer})\frac{|v:C(v,\text{store})>0|}{\sum_{w'}|v:C(v,w')>0|}$

$= \frac{\max(C(\text{computer,store})-0.1,0)}{C(\text{computer})} + \frac{0.1|w:C(\text{computer},w)>0|}{\sum_v C(\text{computer},v)}\frac{|v:C(v,\text{store})>0|}{\sum_{w'}|v:C(v,w')>0|}$

$= \frac{3.9}{10} + \frac{0.1(3)}{10}\frac{3}{9}$

$= 0.39 + 0.01$

$= \mathbf{0.4}$

$P(\text{monitor}|\text{computer})$

$= \frac{\max(C(\text{computer,monitor})-d,0)}{C(\text{computer})} + \lambda(\text{computer})\frac{|v:C(v,\text{monitor})>0|}{\sum_{w'}|v:C(v,w')>0|}$

$= \frac{\max(C(\text{computer,monitor})-0.1,0)}{C(\text{computer})} + \frac{0.1|w:C(\text{computer},w)>0|}{\sum_v C(\text{computer},v)}\frac{|v:C(v,\text{monitor})>0|}{\sum_{w'}|v:C(v,w')>0|}$

$= \frac{3.9}{10} + \frac{0.1(3)}{10}\frac{2}{9}$

$= 0.39 + 0.0067$

$= \mathbf{0.3967}$

As we can see, the probabilities converged significantly closer to their original probabilities of 0.4, but "store" still has a higher probability than "monitor", so nothing has changed. There was no change because the only thing that changed in the calculations was a smaller smoothing constant, d. As such, by making the smoothing constant smaller, the probabilities changed in the same manner as they did in part (b), just to a lesser extent. ■

2. Consider we have a term-document matrix for four words in three documents shown in Table 3. The whole document set has $N = 20$ documents, and for each of the four words, the document frequency df_t is shown in Table 4.

   (a) Compute the *tf-idf* weights for each word car, auto, insurance and best in Doc1, Doc2, and Doc3. [6 pts]

   (b) Use the *tf-idf* weight you get from (a) to represent each document with a vector and calculate the cosine similarities between these three documents. [4 pts]

term-document	Doc1	Doc2	Doc3
car	27	4	24
insurance	3	18	0
auto	0	33	29
best	14	0	17

Table 3: Term-document Matrix

	df
car	12
insurance	6
auto	10
best	16

Table 4: Document Frequency

Solution.
a)

tfidf(car, Doc1) $=$ tf(car)idf(car) $= \log(27+1)\log(\frac{20}{12}) = 1.447(0.2218) = 0.3211$

tfidf(car, Doc2) $=$ tf(car)idf(car) $= \log(4+1)\log(\frac{20}{12}) = 0.6990(0.2218) = 0.1551$

tfidf(car, Doc3) $=$ tf(car)idf(car) $= \log(24+1)\log(\frac{20}{12}) = 1.3979(0.2218) = 0.3101$

tfidf(insurance, Doc1) $=$ tf(insurance)idf(insurance) $= \log(3+1)\log(\frac{20}{6}) = 0.6021(0.5229) = 0.3148$

tfidf(insurance, Doc2) $=$ tf(insurance)idf(insurance) $= \log(18+1)\log(\frac{20}{6}) = 1.2788(0.5229) = 0.6686$

tfidf(insurance, Doc3) $=$ tf(insurance)idf(insurance) $= \log(0+1)\log(\frac{20}{6}) = 0(0.5229) = 0$

tfidf(auto, Doc1) $=$ tf(auto)idf(auto) $= \log(0+1)\log(\frac{20}{10}) = 0(0.3010) = 0$

tfidf(auto, Doc2) $=$ tf(auto)idf(auto) $= \log(33+1)\log(\frac{20}{10}) = 1.5315(0.3010) = 0.4610$

tfidf(auto, Doc3) $=$ tf(auto)idf(auto) $= \log(29+1)\log(\frac{20}{10}) = 1.4771(0.3010) = 0.4447$

tfidf(best, Doc1) $=$ tf(best)idf(best) $= \log(14+1)\log(\frac{20}{16}) = 1.1761(0.0969) = 0.1140$

tfidf(best, Doc2) $=$ tf(best)idf(best) $= \log(0+1)\log(\frac{20}{16}) = 0(0.0969) = 0$

tfidf(best, Doc3) $=$ tf(best)idf(best) $= \log(17+1)\log(\frac{20}{16}) = 1.2553(0.0969) = 0.1216$

b)

$v_{\text{doc1}} = [0.3211, 0.3148, 0, 0.1140]$

$v_{\text{doc2}} = [0.1551, 0.6686, 0.4610, 0]$

$v_{\text{doc3}} = [0.3101, 0, 0.4447, 0.1216]$

$$\text{sim}(v_{\text{doc1}}, v_{\text{doc2}}) = \frac{v_{\text{doc1}}^T v_{\text{doc2}}}{||v_{\text{doc1}}||||v_{\text{doc2}}||} = \frac{0.2603}{0.4639(0.8268)} = 0.6786$$

$$\text{sim}(v_{\text{doc1}}, v_{\text{doc3}}) = \frac{v_{\text{doc1}}^T v_{\text{doc3}}}{||v_{\text{doc1}}||||v_{\text{doc3}}||} = \frac{0.1134}{0.4639(0.5556)} = 0.4401$$

$$\text{sim}(v_{\text{doc2}}, v_{\text{doc3}}) = \frac{v_{\text{doc2}}^T v_{\text{doc3}}}{||v_{\text{doc2}}||||v_{\text{doc3}}||} = \frac{0.2531}{0.8268(0.5556)} = 0.5509$$ ■

3. The distributional hypothesis suggests that the more similarity there is in the meaning of two words, the more distributionally similar they are, where a word's distibution refers to the context in which it appears. This motivated the work by Mikolov et al. on the skip-gram model which is an efficient way of learning high quality dense vector representations of words from unstructured text. The objective of the skip-gram model is to learn the probability distribution $P(O|I)$ where given an inside word w_I, we intend to estimate the probability that an outside word w_O lies in the context window of w_I. The basic formulation of the skip-gram model defines this using the softmax function:

$$P(O = w_O|I = w_I) = \frac{\exp(\boldsymbol{u}_{\boldsymbol{w_O}}^T.\boldsymbol{v}_{\boldsymbol{w_I}})}{\sum_{w \in \text{Vocab}} \exp(\boldsymbol{u}_{\boldsymbol{w}}^T.\boldsymbol{v}_{\boldsymbol{w_I}})} \tag{4}$$

Here, $\boldsymbol{u}_{\boldsymbol{w_O}}$ is the word vector representing the outside word o and $\boldsymbol{v}_{\boldsymbol{w_I}}$ is the word vector representing the inside word i. To update these parameters continually during training, we store these in two matrices $\mathbf{U}$ and $\mathbf{V}$. The columns of $\mathbf{V}$ are all of the inside word vectors $\boldsymbol{v}_{\boldsymbol{w_I}}$ while the columns of $\mathbf{U}$ are all the outside word vectors $\boldsymbol{u}_{\boldsymbol{w_O}}$ and both these matrices contain a vector for each word in the vocabulary.

(a) The cross entropy loss between two probability distributions p and q, is expressed as:

$$CE(p, q) = -\sum_i p_i \log(q_i) \tag{5}$$

For, a given inside word $w_I = w_k$, if we consider the ground truth distribution $\boldsymbol{y}$ to be a one-hot vector (of length same as the size of vocabulary) with a 1 only for the true outside word w_O and 0 everywhere else. The predicted distribution $\hat{\boldsymbol{y}}$ (of length same as the size of vocabulary) is the probability distribution $P(w_O|w_I = w_k)$. The i^{th} entry in these vectors is the probability of the i^{th} word being an outside word. Write down and simplify the expression for the cross entropy loss, $CE(\boldsymbol{y}, \hat{\boldsymbol{y}})$, for the skip-gram model described above for a single pair of words w_O and w_I. (Note: your answer should be in terms of $P(O = w_O|I = w_I)$.) [2 pts]

(b) Find the partial derivative of the cross entropy loss calculated in part (a) with respect to the inside word vector $\boldsymbol{v}_{\boldsymbol{w_I}}$. (Note: your answer should be in terms of $\boldsymbol{y}$, $\hat{\boldsymbol{y}}$ and $\mathbf{U}$.) [5 pts]

(c) Find the partial derivative of the cross entropy loss calculated in part (a) with respect to each of the outside word vectors $\boldsymbol{u}_{\boldsymbol{w_O}}$. (Note: Do this for both cases $w_O = O$ (true outside word) and $w_O \neq O$ (all other words). Your answer should be in terms of $\boldsymbol{y}$, $\hat{\boldsymbol{y}}$ and $\boldsymbol{v}_{\boldsymbol{w_I}}$.) [5 pts]

(d) Explain the idea of negative sampling and the use of the parameter K. Write down the loss function for this case. (Note: your answer should be in terms of $\boldsymbol{u}_{\boldsymbol{w_O}}$, $\boldsymbol{v}_{\boldsymbol{w_I}}$ and the parameter K.) [3 pts]

Solution.
a)
Let k be the ground truth index (i.e. $y_k = 1$ and $y_{m \neq k} = 0$):
$CE(p, q)$
$= -\sum_i p_i \log(q_i)$
$= -\sum_i y_i \log(\hat{y}_i)$
$= -y_k \log(\hat{y}_k)$
$= -\log(P(O = w_O | I = w_I))$
$= -\log(P(O = w_k | I = w_k))$
b) Let $w_I = w_k$ be the ground-truth, correctly predicted word at index k.
$\frac{\delta}{\delta v_{w_I}} - \log(P(O = w_O | I = w_I))$
$= \frac{\delta}{\delta v_{w_I}} - \log(P(O = w_k | I = w_k))$
$= \frac{\delta}{\delta v_{w_I}} - \log(\frac{\exp(u_{w_k}^T.v_{w_k})}{\sum_{w \in \text{Vocab}} \exp(u_w^T.v_{w_I})})$
$= \frac{\delta}{\delta v_{w_I}} \left[-\log(\exp(u_{w_k}^T.v_{w_I})) + \log(\sum_{w \in \text{Vocab}} \exp(u_w^T.v_{w_I})) \right]$
$= \frac{\delta}{\delta v_{w_I}} \left[-u_{w_k}^T.v_{w_I} + \log(\sum_{w \in \text{Vocab}} \exp(u_w^T.v_{w_I})) \right]$
$= -u_{w_k}^T + \frac{\delta}{\delta v_{w_I}} \log(\sum_{w \in \text{Vocab}} \exp(u_w^T.v_{w_I}))$
$= -u_{w_k}^T + \frac{\frac{\delta}{\delta v_{w_I}} \sum_{w \in \text{Vocab}} \exp(u_w^T.v_{w_I})}{\sum_{w \in \text{Vocab}} \exp(u_w^T.v_{w_I})}$
$= -u_{w_k}^T + \frac{\sum_{w \in \text{Vocab}} u_w^T \exp(u_w^T.v_{w_I})}{\sum_{w \in \text{Vocab}} \exp(u_w^T.v_{w_I})}$
$= -u_{w_k}^T + \sum_{w \in \text{Vocab}} u_w^T \hat{y}_w$
$= \sum_{w \in \text{Vocab}} u_w^T \hat{y}_w - u_{w_k}^T$
$= \sum_{w \in \text{Vocab}} \hat{y}_w u_w^T - y_w u_w^T$
$= \sum_{w \in \text{Vocab}} u_w^T (\hat{y}_w - y_w)$
$= U(\hat{y} - y)$
c) Let $w_I = w_k$ be the ground-truth word at index k.
$\frac{\delta}{\delta u_{w_O}} - \log(P(O = w_O | I = w_I))$
$= \frac{\delta}{\delta u_{w_O}} - \log(P(O = w_k | I = w_k))$
$= \frac{\delta}{\delta u_{w_O}} - \log(\frac{\exp(u_{w_k}^T.v_{w_k})}{\sum_{w \in \text{Vocab}} \exp(u_w^T.v_{w_k})})$
$= \frac{\delta}{\delta u_{w_O}} \left[-\log(\exp(u_{w_k}^T.v_{w_k})) + \log(\sum_{w \in \text{Vocab}} \exp(u_w^T.v_{w_k})) \right]$
$= \frac{\delta}{\delta u_{w_O}} \left[-u_{w_k}^T.v_{w_k} + \log(\sum_{w \in \text{Vocab}} \exp(u_w^T.v_{w_k})) \right]$
Now, let $w_O = w_k$ (i.e. the ground-truth correctly predicted word). We get the following derivative:

$\frac{\delta}{\delta u_{w_k}}\big[-u_{w_k}^T.v_{w_k}+\log(\sum_{w\in\text{Vocab}}\exp(u_w^T.v_{w_k}))\big]$

$= -v_{w_k}^T + \frac{\delta}{\delta u_{w_k}}\log(\sum_{w\in\text{Vocab}}\exp(u_w^T.v_{w_k}))$

$= -v_{w_k}^T + \frac{\frac{\delta}{\delta u_{w_k}}\sum_{w\in\text{Vocab}}\exp(u_w^T.v_{w_k})}{\sum_{w\in\text{Vocab}}\exp(u_w^T.v_{w_k})}$

$= -v_{w_k}^T + \frac{v_{w_k}^T\exp(u_{w_k}^T v_{w_k})}{\sum_{w\in\text{Vocab}}\exp(u_w^T.v_{w_k})}$

$= -v_{w_k}^T + v_{w_k}^T\hat{y}_k$

$= -v_{w_k}^T + y^T\hat{y}v_{w_k}^T$

$= v_{w_k}^T(y^T\hat{y}-1)$

$= v_{w_I}^T(y^T\hat{y}-1)$

Now, instead, let $w_O = w_m$ (where $m \neq k$). We get the following derivative:

$\frac{\delta}{\delta u_{w_m}}\big[-u_{w_k}^T.v_{w_k}+\log(\sum_{w\in\text{Vocab}}\exp(u_w^T.v_{w_k}))\big]$

$= \frac{\delta}{\delta u_{w_m}}\log(\sum_{w\in\text{Vocab}}\exp(u_w^T.v_{w_k}))$

$= \frac{\frac{\delta}{\delta u_{w_m}}\sum_{w\in\text{Vocab}}\exp(u_w^T.v_{w_k})}{\sum_{w\in\text{Vocab}}\exp(u_w^T.v_{w_k})}$

$= \frac{\sum_{w\in(\text{Vocab}/w_k)}v_{w_k}^T\exp(u_w^T.v_{w_k})}{\sum_{w\in\text{Vocab}}\exp(u_w^T.v_{w_k})}$

$= \sum_{w\in(\text{Vocab}/w_k)}v_{w_k}^T\hat{y}_w$

$= v_{w_k}^T\sum_{w\in(\text{Vocab}/w_k)}\hat{y}_w$

$= v_{w_k}^T\left[(1-y)^T\hat{y}\right]$

$= (1-y)^T\hat{y}v_{w_k}^T$

$= (1-y)^T\hat{y}v_{w_I}^T$

d)

For a network like word2vec, stochastic gradient descent requires us to compute the probability of each word across our entire vocabulary. As such, when we do backpropagation later on in the process, and V (the size of our vocabulary) is massive (tens of thousands or hundreds of thousands of words), this process becomes computationally expensive, especially once you consider iterating over several epochs of your data. As such, negative sampling solves this problem. We will take our target word (e.g. apricot) and choose a window positive examples around our target word as positive samples (e.g. let c = 2, then choose the 2*c surrounding words = 4 total words used as positive samples) and for each positive example, you choose K negative samples (any word in our corpus that is not our target word, as described in the "8_word_embedding.pdf" slides for our class). As such, if K = 2, and in the above example we had c = 2 (leading to 4 positive samples), then we would have 4*K = 4*2 = 8 negative samples. As such, instead of performing stochastic gradient descent across our vocabulary of 100,000 words/samples, we now only perform descent across 4 positive + 8 negative = 12 samples, significantly reducing computation time.

As such, the role of K is to increase the number of negative samples used for computation in our backpropagation steps. It goes in tandem with C, which is the size of our positive sample window, since C increases the number of positive samples, and K increases as the number of positive samples increases.

However, since we now using negative sampling, we can treat each word as a binary classification problem. Let t be the target word, c be a positive word sample, and n be a negative word sample. This binary classification problem can model a probability distribution where positive samples (labeled as similarity score s = 1) are modeled as $P(s = 1|c, t)$ and K negative samples (labeled as similarity score s = 0) are modeled as $P(s = 1|n, t)$. We want to maximize the similarity between the positive samples and minimize similarity between negative samples. First, let us define $P(s = 1|O, I) = \frac{1}{1+\exp(-u_{w_O}^T v_{w_I})} = \sigma(u_{w_O}^T v_{w_I})$. Then, we can transform the above loss function into the following (for a single positive word sample), which is the objective function we want to maximize:

$L(\theta; w, c, n)$

$= \log P(s = 1|w_c, w_I) + \sum_{i=1}^{K} \log P(s = 0|n_i, w_I)$

$= \log \sigma(u_{w_c}^T v_{w_I}) + \sum_{i=1}^{K} \log(1 - P(s = 1|n_i, w_I))$

$= \log \sigma(u_{w_c}^T v_{w_I}) + \sum_{i=1}^{K} \log \sigma(-u_{n_i}^T v_{w_I})$

■

4. In the textbook, language modeling was defined as the task of predicting the next word in a sequence given the previous words. In this assignment, you will implement character-level, word-level N-gram and character-level RNN language models. You need to both answer the questions and submit your code. You need to submit a zip file to Canvas Homework 3 Programming. This file should contain `ngram.ipynb`, `rnn.ipynb` `hw3_skeleton_char.py` and `hw3_skeleton_word.py`.

   You should also submit a zip file to the Gradescope assignment HW3 Language Models. This file should contain `hw3_skeleton_char.py` and `hw3_skeleton_word.py`.

   (a) For N-gram language models, You should complete two scripts `hw3_skeleton_char.py` and `hw3_skeleton_word.py`. Detailed instructions can be found in `ngram.ipynb`. You should also use test cases in `ngram.ipynb` and use `ngram.ipynb` to get development results for (c).

   You need to submit a zip file to Gradescope HW3 Language Models. This file should contain `hw3_skeleton_char.py` and `hw3_skeleton_word.py`. You can see the scores for your code there. Character-level N-gram language models accounts for 20 points. Word-level N-gram language models accounts for 10 points, which are bonus for CS 4650. [30pts for CS 7650, 20 pts + bonus 10pts for CS 4650]

   (b) See the generation results of your character-level and word-level N-gram language models respectively ($n \geq 1$). The paragraphs which character-level N-gram language models generate all start with *F*. The paragraphs which word-level N-gram language models generate all start with *In*. Did you get such results?

Explain what is going on. (CS 4650 can only focus on character-level N-gram language model.) [2 pts]

(c) (Bonus for CS 4650) Compare the generation results of character-level and word-level N-gram language models. Which do you think is better? Compare the perplexity of `nytimes_article.txt` when using character-level and word-level N-gram language models. Explain what you found. [2pts, bonus for CS 4650]

(d) When you compute perplexity, you can play with different sets of hyper-parameters in both character-level and word-level N-gram language models. You can tune n , k and λ. Please report here the best results and the corresponding hyper-parameters in development sets. For character-level N-gram language models, the development set is `shakespeare_sonnets.txt`. For word-level N-gram language models, the development sets are `shakespeare_sonnets.txt` and `val_e.txt`. (CS 4650 should only focus on character-level N-gram language model.) [6 pts for CS 7650, 2 pts + bonus 4 pts for CS 4650]

(e) For RNN language models, You should complete the forward method of Class RNN in `rnn.ipynb`. You need to figure out the code and tune the hyperparameters. You should also copy a paragraph generated by your model and report the perplexity on the development set `shakespeare_sonnets.txt`. Compare the results of character-level RNN language model and character-level N-gram language model. [10 pts]

Solution.
a) Refer to code.

b) The reason "First" is always being generated first is because after examining the file shakespeare_input.txt, you can see the first word is "First Citizen". As such, when this entire chunk of text is fed into the model to update the model, the only character that appears after ∼∼ is F. This would actually be cured if we passed several different Shakespeare input files into our model to update it, as each input file (each play) would start with a different word, as such, there would be more variance in the first character generated by the model. In other words, if "All" was the first speaker of the play instead of "First Citizen", the model would learn to be biased toward starting its text generation with the letter A instead of F.

The reason why "In" appears in every instance for the word-level generation is similar. All three word-level models are trained on train_e.txt, and if you look at the file, it begins with the word "In". Furthermore, if you look at the create_ngram_model() function, it simply reads in the text from a file without splitting them by lines, so the only instance where "∼" appears as the context (with 2 tildas for the 2-gram model, 3 tildas for 3-gram, etc.), is in the case where we see "∼∼ In". As such, the model is biased toward this only occurrence since "In" has an insanely high chance of occurring after the tildas.

c) This one is a no brainer: the word-level model is better. The explanation for this is because the character-level model must generate real words, and then once it is successful at generating words that actually exist, it must generate sentences that sound coherent. Meanwhile, the word-level model does not have to worry about the first issue. It already generates real words from a word bank, so the word-level model only has to perform well at stringing those words together into coherent sentences. And this can be seen by the random text generated by both 4-gram character and word-level models. The word-level models do not have any deficiencies in terms of generating realistic words. Their only deficiencies come in terms of generating realistic phrases.

Now, the quantitative measure, perplexity, was compared between the two interpolation models with the best hyperparameters found in part (d) of this question. For the character-level ngram model, a perplexity of 5.1700 was achieved. Meanwhile, for the word-level ngram model, a perplexity of 1549.9949 was achieved.

Now, with that being said, it is unfair to compare the word and character level models' perplexities because the shakespeare sonnets test file has a significant number more characters than words, so the probability, or score, is normalized by a larger amount when taking the n-th root. For example, if there were N words, the word-level model would take the N-th root, but if we assume there are 5 letters per word, then there are 5N characters, so the character model would take the (5N)-th root, producing a much smaller score. Additionally, the vocab of the word model is size 62983, and the vocab for the character model is size 67, so there will inherently be different perplexities due to that. One of the reasons this is the case is because when a context does not exist in our model and we assign a uniform probability to a given word or character existing, the word-level model will assign a smaller probability (1/62983) compared to the character level model (1/67).

d) To simplify the process, I first tuned the hyperparameters of the character-level ngram model. First, I tuned the value of n, then once I found the optimal n, I tuned the value of the lambdas, then I finally tuned the value of k. You can see the results (measured through perplexity) below:

$n = 2 \implies 10.2814$
$n = 3 \implies 7.9210$
$n = 4 \implies 6.7296$
$n = 5 \implies 6.1147$
$n = 10 \implies 6.4789$

Then, I took the 5-gram model and tuned the values of the lambdas as follows:

$[1, 0, 0, 0, 0, 0] \implies 23.4546$
$[0, 1, 0, 0, 0, 0] \implies 12.5071$
$[0, 0, 1, 0, 0, 0] \implies 8.1685$
$[0, 0, 0, 1, 0, 0] \implies 5.9825$
$[0, 0, 0, 0, 1, 0] \implies 5.5298$
$[0, 0, 0, 0, 0, 1] \implies 6.2527$
$[0.01, 0.01, 0.01, 0.01, 0.9, 0.06] \implies 5.2464$
$[0.05, 0.05, 0.05, 0.05, 0.75, 0.05] \implies 5.2661$
$[0.05, 0.05, 0.05, 0.1, 0.5, 0.25] \implies 5.1700$
$[0.1, 0.1, 0.1, 0.1, 0.4, 0.2] \implies 5.4642$

Then, I used lambdas of $\lambda_1 = 0.05, \lambda_2 = 0.05, \lambda_3 = 0.05, \lambda_4 = 0.1, \lambda_5 = 0.5, \lambda_6 = 0.25$ and tuned the values of k:

$k = 0.01 \implies 5.1700$
$k = 0.1 \implies 5.4272$
$k = 0.2 \implies 5.6311$
$k = 0.5 \implies 6.0760$
$k = 1.0 \implies 6.6078$
$k = 2.0 \implies 7.3816$

So, as seen above, the best hyperparameters for the character-level model are $n = 5$, lambdas $= [0.05, 0.05, 0.05, 0.1, 0.5, 0.25]$, $k = 0.01$. For the word-level model, I did the same amount of hyperparameter tuning as I did in the character-level model. So, for the word-level model:

$n = 2 \implies 4649.9847$
$n = 3 \implies 5209.0778$
$n = 4 \implies 5776.3654$
$n = 5 \implies 6315.5207$
$n = 10 \implies 8594.5631$

$[1, 0, 0] \implies 1549.9949$
$[0, 1, 0] \implies inf$
$[0, 0, 1] \implies inf$
$[0.9, 0.05, 0.05] \implies 1722.2166$
$[0.7, 0.15, 0.15] \implies 2214.2784$
$[0.5, 0.25, 0.25] \implies 3099.9898$

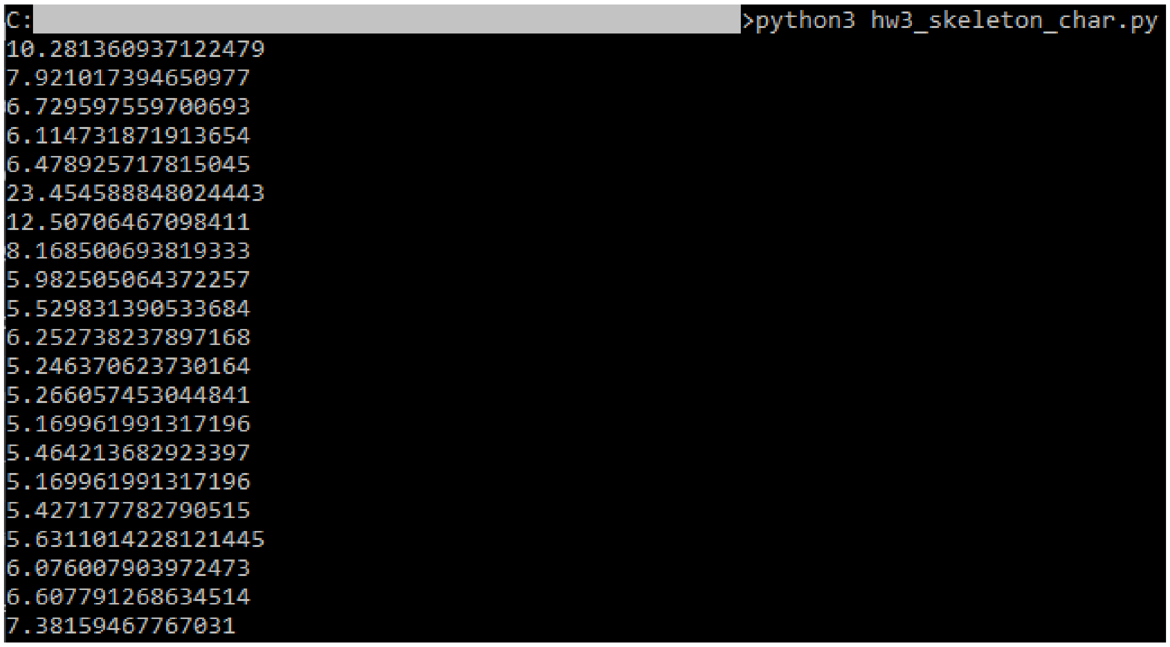
$k = 0.01 \implies 1549.9949$
$k = 0.1 \implies 1794.6906$
$k = 0.2 \implies 1897.3396$
$k = 0.5 \implies 2047.2329$
$k = 1.0 \implies 2170.6000$
$k = 2.0 \implies 2313.9570$

So, the best word-level model has $n = 2$, lambdas $= [1, 0, 0]$, $k = 0.01$. These were validated with the shakespeare_sonnett.txt set, and after checking perplexity across all parameters on the other dev set, val_e.txt, the same general patterns hold, so I am certain these parameters are the best ones for the word-level model.

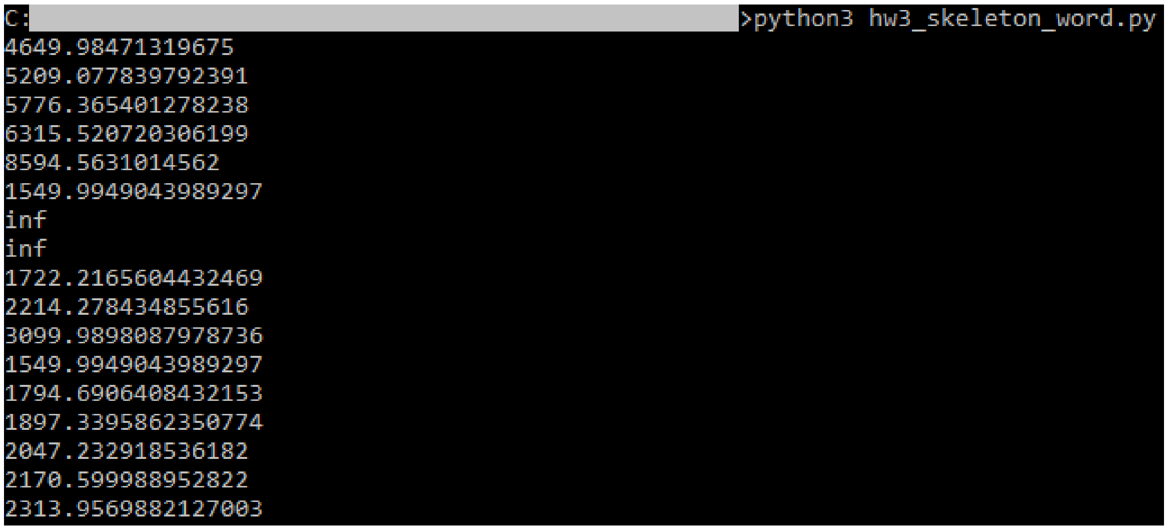
To wrap up, on the same dev set (shakespeare_sonnett.txt) for the character and word-level models, the perplexity for the character model is 5.1700 and for the word-level model is 1549.9949. For the other word-level dev set, val_e.txt, I am getting a perplexity of 393.8618 after training on train_e.txt.

If you want to specifically see these scores, or simulate the same tests, just run "python3 hw3_skeleton_word.py" and "python3 hw3_skeleton_word.py". I specifically added the tests in the main() function, so the tests will run immediately after calling the program.

Below are two screenshots. The first one shows the perplexity outputs from the character-level model in the order I presented above. The second screenshot shows the perplexities output from the word-level model in the order I presented above. Both were the output after training on shakespeare_input.txt and testing on shakespeare_sonnets.txt. I did not show results on val_e.txt for brevity's sake.



```
C:[redacted]>python3 hw3_skeleton_char.py
10.281360937122479
7.921017394650977
6.729597559700693
6.114731871913654
6.478925717815045
23.454588848024443
12.50706467098411
8.168500693819333
5.982505064372257
5.529831390533684
6.252738237897168
5.246370623730164
5.266057453044841
5.169961991317196
5.464213682923397
5.169961991317196
5.427177782790515
5.6311014228121445
6.076007903972473
6.607791268634514
7.38159467767031
```



```
C:[redacted]>python3 hw3_skeleton_word.py
4649.98471319675
5209.077839792391
5776.365401278238
6315.520720306199
8594.5631014562
1549.9949043989297
inf
inf
1722.2165604432469
2214.278434855616
3099.9898087978736
1549.9949043989297
1794.6906408432153
1897.3395862350774
2047.232918536182
2170.599988952822
2313.9569882127003
```

e) After evaluating on "shakespeare_sonnets.txt", I obtained a perplexity of 6.8739 on the character-level RNN. One sample from the character-level RNN with temperature of 0.2 is the following:

"The hath he speak he come.

SARDONA: I have he be the best the grown and he be be and for the speak.

PERDARIUS: What hath the come the come the come and hath the come.

CROSANDO: I have the come the pr"

Please ignore the quotations. I added those purely as an indicator it is output from the model, rather than something I directly typed. Now, we can compare this to a sample paragraph from the character-level 4-gram language model:

“”First Apollows-monstands. The foundly and highness’ friend.\n\nCOSTARD:\nThe would first Guard:\nJoin while he days I true:\nI have the reputation, good:\nBut a tradian a prithers.\n\nTOUCHSTONE:\nSoft,\nthough they lief talend mud indeed, double your treason ””

Once again, the pair of quotations inserted at the beginning and end of the paragraph were inserted by me, but all the other quotations were generated by the model. In general, both models perform terribly in terms of generating coherent text that is easily readable by English speakers. However, I will say the 4-gram model performed slightly better in terms of coherency, although marginal. The RNN generated sequences such as “the come the come the come and hath the come” which makes no sense at all, while the 4-gram model generated phrases such as “I have the reputation, good”, which is significantly easier to read and comprehend. Meanwhile, the RNN generated more realistic paragraphs in the sense that the RNN-generated text looks like it just came out of a playbook, because it is well-structured (i.e. a character, “SARDONA”, is specifically speaking at a certain time). Now, with that being said, I will not paste the paragraphs for the 3-gram or 2-gram ngram character models, but those are significantly worse than the character RNN. As such, I will say the 4-gram character model works better than the RNN character model, in terms of qualitative analysis, but the RNN character model performs leaps and bounds better than the 2-gram and 3-gram models in terms of generating coherent paragraphs, text, and words.

With that being said, when comparing the two models quantitatively, namely through perplexity, the ngram model performs better. Through the character-level ngram model, the perplexity is 5.1700 (as mentioned in question 4d). Meanwhile, the RNN generated a sentence with perplexity of 6.8739, which is worse than the ngram model. As such, I believe the ngram model does a better job at selecting words, under uncertainty, that actually have a chance of appearing. Therefore, since I believe the 4-gram ngram model does a better job qualitatively, and it has shown to perform better quantitatively, the 4-gram model is better than the RNN. ■