1004365_mini_project

June 17, 2022

##

50.040 Natural Language Processing, Summer 2021

Due 17 June 2021, 5pm

Mini Project

Write your student ID and name

0.0.1 STUDENT ID: 1004365

0.0.2 Name: Lee Jet Xuen

0.0.3 Students with whom you have discussed (if any):

1. Tay Sze Chang 1004301
2. Brandon Chong Wah Jin 1004104

1 Introduction

Language models are very useful for a wide range of applications, e.g., speech recognition and machine translation. Consider a sentence consisting of words $x_1, x_2, ..., x_m$, where m is the length of the sentence, the goal of language modeling is to model the probability of the sentence, where $m \geq 1$, $x_i V $ and V is the vocabulary of the corpus:

$$p(x_1, x_2, ..., x_m)$$

In this project, we are going to explore both statistical language model and neural language model on the Wikitext-2 datasets. Download wikitext-2 word-level data and put it under the `data` folder.

1.1 Statistical Language Model

A simple way is to view words as independent random variables (i.e., zero-th order Markovian assumption). The joint probability can be written as:

$$p(x_1, x_2, ..., x_m) = \prod_{i=1}^{m} p(x_i)$$

However, this model ignores the word order information, to account for which, under the first-order Markovian assumption, the joint probability can be written as:

$$p(x_0, x_1, x_2, ..., x_m) = \prod_{i=1}^{m} p(x_i \mid x_{i-1})$$

Under the second-order Markovian assumption, the joint probability can be written as:

$$p(x_{-1}, x_0, x_1, x_2, ..., x_m) = \prod_{i=1}^{m} p(x_i \mid x_{i-2}, x_{i-1})$$

Similar to what we did in HMM, we will assume that $x_{-1} = START, x_0 = START, x_m = STOP$ in this definition, where $START, STOP$ are special symbols referring to the start and the end of a sentence.

1.1.1 Parameter estimation

Let's use $count(u)$ to denote the number of times the unigram u appears in the corpus, use $count(v, u)$ to denote the number of times the bigram v, u appears in the corpus, and $count(w, v, u)$ the times the trigram w, v, u appears in the corpus, $u \in V \cup STOP$ and $w, v \in V \cup START$.

And the parameters of the unigram, bigram and trigram models can be obtained using maximum likelihood estimation (MLE).

- In the unigram model, the parameters can be estimated as:

  $$p(u) = \frac{count(u)}{c}$$

  , where c is the total number of words in the corpus.
- In the bigram model, the parameters can be estimated as:

  $$p(u \mid v) = \frac{count(v, u)}{count(v)}$$

- In the trigram model, the parameters can be estimated as:

  $$p(u \mid w, v) = \frac{count(w, v, u)}{count(w, v)}$$

1.1.2 Smoothing the parameters

Add-k Smoothing Note, it is likely that many parameters of bigram and trigram models will be 0 because the relevant bigrams and trigrams involved do not appear in the corpus. If you don't have a way to handle these 0 probabilities, all the sentences that include such bigrams or trigrams will have probabilities of 0.

We'll use a Add-k Smoothing method to fix this problem, the smoothed parameters can be estimated as:

$$p_{add-k}(u) = \frac{count(u) + k}{c + k|V^*|} \tag{1}$$

$$p_{add-k}(u \mid v) = \frac{count(v, u) + k}{count(v) + k|V^*|} \tag{2}$$

$$p_{add-k}(u \mid w, v) = \frac{count(w, v, u) + k}{count(w, v) + k|V^*|} \tag{3}$$

where $k \in (0, 1)$ is the parameter of this approach, and $|V^*|$ is the size of the vocabulary V^*, here $V^* = V \cup STOP$. One way to choose the value of k is by optimizing the perplexity of the development set, namely to choose the value that minimizes the perplexity. #### Interpolation There is another way for smoothing which is named as **interpolation**. In interpolation, we always mix the probability estimates from all the n-gram estimators, weighing and combining the trigram, bigram, and unigram counts. In simple linear interpolation, we combine different order n-grams by linearly interpolating all the models. Thus, we estimate the trigram probability $p(w_n|w_{n-2}, w_{n-1})$ by mixing together the unigram, bigram, and trigram probabilities, each weighted by a λ:

$$\hat{p}(w_n|w_{n-2}, w_{n-1}) = \lambda_1 p(w_n|w_{n-2}, w_{n-1}) + \lambda_2 p(w_n|w_{n-1}) + \lambda_3 p(w_n) \tag{4}$$

such that the λs sum to 1:

$$\sum_i \lambda_i = 1 \tag{5}$$

In addition, $\lambda_1, \lambda_2, \lambda_3 \geq 0$.

1.1.3 Perplexity

Given a test set D' consisting of sentences $X^{(1)}, X^{(2)}, ..., X^{(|D'|)}$, each sentence $X^{(j)}$ consists of words $x_1^{(j)}, x_2^{(j)}, ..., x_{n_j}^{(j)}$, we can measure the probability of each sentence $X^{(j)}$, and the quality of the language model would be the probability it assigns to the entire set of test sentences, namely:

$$\prod_{j=1}^{|D'|} p(X^{(j)}) \tag{6}$$

Let's define average log_2 probability as:

$$l = \frac{1}{c'} \sum_{j=1}^{|D'|} log_2 p(X^{(j)}) \tag{7}$$

c' is the total number of words in the test set, $|D'|$ is the number of sentences. And the perplexity is defined as:

$$perplexity = 2^{-l} \tag{8}$$

The lower the perplexity, the better the language model.

[1]:
```
from google.colab import drive
drive.mount('/content/drive')
%cd drive/MyDrive/nlp_mini_project
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
/content/drive/MyDrive/nlp_mini_project
```

[2]:
```
from collections import Counter, namedtuple, defaultdict
from nltk import ngrams
import itertools
import numpy as np
import sys
```

```
[3]: with open("./data/wikitext-2/wiki.train.tokens", 'r', encoding='utf8') as f:
         text = f.readlines()
         train_sents = [line.lower().strip('\n').split() for line in text]
         train_sents = [s for s in train_sents if len(s)>0 and s[0] != '=']
```

```
[4]: print(train_sents[1])
```

```
['the', 'game', 'began', 'development', 'in', '2010', ',', 'carrying', 'over',
'a', 'large', 'portion', 'of', 'the', 'work', 'done', 'on', 'valkyria',
'chronicles', 'ii', '.', 'while', 'it', 'retained', 'the', 'standard',
'features', 'of', 'the', 'series', ',', 'it', 'also', 'underwent', 'multiple',
'adjustments', ',', 'such', 'as', 'making', 'the', 'game', 'more', '<unk>',
'for', 'series', 'newcomers', '.', 'character', 'designer', '<unk>', 'honjou',
'and', 'composer', 'hitoshi', 'sakimoto', 'both', 'returned', 'from',
'previous', 'entries', ',', 'along', 'with', 'valkyria', 'chronicles', 'ii',
'director', 'takeshi', 'ozawa', '.', 'a', 'large', 'team', 'of', 'writers',
'handled', 'the', 'script', '.', 'the', 'game', "'s", 'opening', 'theme', 'was',
'sung', 'by', 'may', "'n", '.']
```

1.1.4 Question 1 [code]

1. Implement the function **"compute_ngram"** that computes n-grams in the corpus. (Do not take the START and STOP symbols into consideration for now.)
2. List 10 most frequent unigrams, bigrams and trigrams as well as their counts.(Hint: use the built-in function .most_common in Counter class)

```
[5]: def compute_ngram(sents, n):
         '''
         Compute n-grams that appear in "sents".
         param:
             sents: list[list[str]] --- list of list of word strings
             n: int --- "n" gram
         return:
             ngram_set: set{str} --- a set of n-grams (no duplicate elements)
             ngram_dict: dict{ngram: counts} --- a dictionary that maps each ngram␣
     ↪to its number occurence in "sents";
             This dict contains the parameters of our ngram model. E.g. if n=2,␣
     ↪ngram_dict={('a','b'):10, ('b','c'):13}

             You may need to use "Counter", "tuple" function here.
         '''
         ngram_set = None
         ngram_dict = None
         ### YOUR CODE HERE

         ngram_set = set()
         ngram_dict = defaultdict(int)
```

```
    for word in sents:
        for i in range(len(word)-n+1):
            ngram_set.add(tuple(word[i: i+n]))
            ngram_dict[tuple(word[i: i+n])] += 1

    ngram_dict = Counter(ngram_dict)

    ### END OF YOUR CODE
    return ngram_set, ngram_dict
```

[6]:
```
unigram_set, unigram_dict = compute_ngram(train_sents, 1)
print('unigram: %d' %(len(unigram_set)))
bigram_set, bigram_dict = compute_ngram(train_sents, 2)
print('bigram: %d' %(len(bigram_set)))
trigram_set, trigram_dict = compute_ngram(train_sents, 3)
print('trigram: %d' %(len(trigram_set)))
```

```
unigram: 28910
bigram: 577343
trigram: 1344047
```

[7]:
```
# List 10 most frequent unigrams, bigrams and trigrams as well as their counts.
### YOUR CODE HERE
print("~" * 117)
print("unigram:")
print(unigram_dict.most_common(10))
print("~" * 117)
print("bigram:")
print(bigram_dict.most_common(10))
print("~" * 117)
print("trigram:")
print(trigram_dict.most_common(10))
print("~" * 117)
### END OF YOUR CODE
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
unigram:
[(('the',), 130519), ((',',), 99763), (('.',), 73388), (('of',), 56743),
(('<unk>',), 53951), (('and',), 49940), (('in',), 44876), (('to',), 39462),
(('a',), 36140), (('"',), 28285)]
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
bigram:
[(('of', 'the'), 17242), (('in', 'the'), 11778), ((',', 'and'), 11643), (('.',
'the'), 11274), ((',', 'the'), 8024), (('<unk>', ','), 7698), (('to', 'the'),
```

```
6009), (('on', 'the'), 4495), (('the', '<unk>'), 4389), (('and', 'the'), 4331)]
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
trigram:
[((',', 'and', 'the'), 1393), ((',', '<unk>', ','), 950), (('<unk>', ',',
'<unk>'), 901), (('one', 'of', 'the'), 866), (('<unk>', ',', 'and'), 819),
(('.', 'however', ','), 775), (('<unk>', '<unk>', ','), 745), (('.', 'in',
'the'), 726), (('.', 'it', 'was'), 698), (('the', 'united', 'states'), 666)]
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

1.1.5 Question 2 [code]

In this part, we take the START and STOP symbols into consideration. So we need to pad the **train_sents** as described in "Statistical Language Model" before we apply "compute_ngram" function. For example, given a sentence "I like NLP", in a bigram model, we need to pad it as "START I like NLP STOP", in a trigram model, we need to pad it as "START START I like NLP STOP". For unigram model, it should be paded as "I like NLP STOP".

1. Implement the `pad_sents` function.
2. Pad `train_sents`.
3. Apply `compute_ngram` function to these padded sents.
4. Implement `ngram_prob` function. Compute the probability for each n-gram in the variable **ngrams** according equations in **"Parameter estimation"**. List down the n-grams that have 0 probability.

[8]:
```
###############################################
ngrams = list()
with open('data/ngram.txt','r') as f:
    for line in f:
        ngrams.append(line.strip('\n').split())
print(ngrams)
###############################################
```

```
[['the', 'computer'], ['go', 'to'], ['have', 'had'], ['and', 'the'], ['can',
'sea'], ['a', 'number', 'of'], ['with', 'respect', 'to'], ['in', 'terms', 'of'],
['not', 'good', 'bad'], ['first', 'start', 'with']]
```

[9]:
```
START = '<START>'
STOP = '<STOP>'
###################################
def pad_sents(sents, n):
    '''
    Pad the sents according to n.
    params:
        sents: list[list[str]] --- list of sentences.
        n: int --- specify the padding type, 1-gram, 2-gram, or 3-gram.
    return:
        padded_sents: list[list[str]] --- list of padded sentences.
```

```python
    '''
    padded_sents = []
    ### YOUR CODE HERE

    for sent in sents:
        new_sent = sent[:]

        if n == 1:
            new_sent.append(START)

        elif n == 2:
            new_sent.append(STOP)
            new_sent.insert(0, START)

        elif n == 3:
            new_sent.append(STOP)
            new_sent.insert(0, START)
            new_sent.insert(0, START)

        padded_sents.append(new_sent)
#
    ### END OF YOUR CODE
    return padded_sents
```

[10]:
```python
uni_sents = pad_sents(train_sents, 1)
bi_sents = pad_sents(train_sents, 2)
tri_sents = pad_sents(train_sents, 3)
```

[11]:
```python
unigram_set, unigram_dict = compute_ngram(uni_sents, 1)
bigram_set, bigram_dict = compute_ngram(bi_sents, 2)
trigram_set, trigram_dict = compute_ngram(tri_sents, 3)
```

[12]:
```python
len(unigram_set),len(bigram_set),len(trigram_set)
```

[12]: (28911, 580825, 1363266)

[13]:
```python
num_words = sum([v for _,v in unigram_dict.items()])
print(num_words)
```

2024702

[14]:
```python
def ngram_prob(ngram, num_words, unigram_dic, bigram_dic, trigram_dic):
    '''
    params:
        ngram: list[str] --- a list that represents n-gram
        num_words: int --- total number of words
```

```
        unigram_dic: dict{ngram: counts} --- a dictionary that maps each 1-gram␣
↪to its number of occurences in "sents";
        bigram_dic: dict{ngram: counts} --- a dictionary that maps each 2-gram␣
↪to its number of occurence in "sents";
        trigram_dic: dict{ngram: counts} --- a dictionary that maps each 3-gram␣
↪to its number occurence in "sents";
    return:
        prob: float --- probability of the "ngram"
    '''
    prob = None
    ### YOUR CODE HERE

    n = len(ngram)

    try:
        if n == 1:
            u = tuple(ngram)
            prob = unigram_dic[u]/num_words

        elif n == 2:
            bi = tuple(ngram)
            uni = tuple([ngram[1]])
            num = bigram_dic[bi]
            deno = unigram_dic[uni]
            prob = num/deno

        elif n == 3:
            tri = tuple(ngram)
            bi = tuple(ngram[1:])
            num = trigram_dic[tri]
            deno = bigram_dic[bi]
            prob = num/deno
    except ZeroDivisionError:
        return 0

    ### END OF YOUR CODE
    return prob
```

[15]:
```
ngram_prob(ngrams[0], num_words, unigram_dict, bigram_dict, trigram_dict)
```

[15]: 0.0962962962962963

[16]:
```
### List down the n-grams that have 0 probability.
### YOUR CODE HERE
for ngram in ngrams:
    if ngram_prob(ngram, num_words, unigram_dict, bigram_dict, trigram_dict) ==␣
↪0:
```

```
        print(ngram)
### END OF YOUR CODE
```

```
['can', 'sea']
['not', 'good', 'bad']
['first', 'start', 'with']
```

1.1.6 Question 3 [code]

1. Implement `add_k_smoothing_ngram` function to estimate ngram probability with `add-k` smoothing technique.
2. Implement `interpolation_ngram` function to estimate ngram probability with `interpolation` smoothing technique.
3. Implement `perplexity` function to compute the perplexity of the corpus "**valid_sents**" according to "**Perplexity**" section. The computation of $p(X^{(j)})$ depends on the n-gram model you choose.

[17]:
```
with open('data/wikitext-2/wiki.valid.tokens', 'r', encoding='utf8') as f:
    text = f.readlines()
    valid_sents = [line.lower().strip('\n').split() for line in text]
    valid_sents = [s for s in valid_sents if len(s)>0 and s[0] != '=']

uni_valid_sents = pad_sents(valid_sents, 1)
bi_valid_sents = pad_sents(valid_sents, 2)
tri_valid_sents = pad_sents(valid_sents, 3)
```

[18]:
```
def add_k_smoothing_ngram(ngram, k, num_words, unigram_dic, bigram_dic,␣
↪trigram_dic):
    '''
    params:
        ngram: list[str] --- a list that represents n-gram
        k: float
        num_words: int --- total number of words
        unigram_dic: dict{ngram: counts} --- a dictionary that maps each 1-gram␣
↪to its number of occurences in "sents";
        bigram_dic: dict{ngram: counts} --- a dictionary that maps each 2-gram␣
↪to its number of occurence in "sents";
        trigram_dic: dict{ngram: counts} --- a dictionary that maps each 3-gram␣
↪to its number occurence in "sents";
    return:
        s_prob: float --- probability of the "ngram"
    '''
    s_prob = None
    V = len(unigram_dic)
    ### YOUR CODE HERE

    num = len(ngram)
```

```python
    try:
      if num == 1:
          uni = tuple(ngram)
          nume = unigram_dic[uni] + k
          deno = num_words + k * V
          s_prob = nume / deno

      elif num == 2:
          bi = tuple(ngram)
          uni = tuple([ngram[1]])
          nume = bigram_dic[bi] + k
          deno = unigram_dic[uni] + k * V
          s_prob = nume / deno

      elif num == 3:
          tri = tuple(ngram)
          bi = tuple(ngram[1:])
          nume = trigram_dic[tri] + k
          deno = bigram_dic[bi] + k * V
          s_prob = nume / deno

    except ZeroDivisionError:
        return 0

    ### END OF YOUR CODE
    return s_prob
```

[19]:
```python
def interpolation_ngram(ngram, lam, num_words, unigram_dic, bigram_dic,␣
 ↪trigram_dic):
    '''
    params:
        ngram: list[str] --- a list that represents n-gram
        lam: list[float] --- a list of length 3.lam[0], lam[1] and lam[2] are␣
 ↪correspondence to trigram, bigram and unigram,repectively.
                                  If len(ngram) == 1, lam[0]=lam[1]=0, lam[2]=1. If␣
 ↪len(ngram) == 2, lam[0]=0. lam[0]+lam[1]+lam[2] = 1.
        num_words: int --- total number of words
        unigram_dic: dict{ngram: counts} --- a dictionary that maps each 1-gram␣
 ↪to its number of occurences in "sents";
        bigram_dic: dict{ngram: counts} --- a dictionary that maps each 2-gram␣
 ↪to its number of occurence in "sents";
        trigram_dic: dict{ngram: counts} --- a dictionary that maps each 3-gram␣
 ↪to its number occurence in "sents";
    return:
        s_prob: float --- probability of the "ngram"
    '''
```

```
    s_prob = None
    ### YOUR CODE HERE
    num = len(ngram)

    if num == 1:
        u = tuple(ngram)
        s_prob = unigram_dic[u]/num_words

    elif num == 2:
        bi_num = tuple(ngram)
        uni_deno = tuple([ngram[1]])
        bi = bigram_dic[bi_num] / unigram_dic[uni_deno]

        uni_num = tuple([ngram[0]])
        uni = unigram_dic[uni_num]/ num_words

        s_prob = lam[1] * bi + lam[2] * uni

    elif num == 3:
      tri_num = tuple(ngram)
      bi_deno = tuple(ngram[1:])
      tri = trigram_dic[tri_num]/bigram_dic[bi_deno]

      bi_num = tuple(ngram[0:2])
      uni_deno = tuple([ngram[0]])
      bi  = bigram_dic[bi_num]/unigram_dic[uni_deno]

      uni_num = tuple([ngram[0]])
      uni  = unigram_dic[uni_num]/num_words

      s_prob = lam[0] * tri +\
               lam[1] * bi +\
               lam[2] * uni

    ### END OF YOUR CODE
    return s_prob
```

[20]:
```
add_k_prob = add_k_smoothing_ngram(ngrams[5], 0.01, num_words, unigram_dict,␣
 ↪bigram_dict, trigram_dict)
interpolation_prob = interpolation_ngram(ngrams[5], [0.6,0.3,0.1], num_words,␣
 ↪unigram_dict, bigram_dict, trigram_dict)
print(ngrams[5])
print(add_k_prob, interpolation_prob)
```

```
['a', 'number', 'of']
0.3368808402441772 0.2975092541237132
```

[21]:
```python
def perplexity(n, method, num_words, valid_sents, unigram_dic, bigram_dic,␣
↪trigram_dic, k=0, lam=[0,0,1]):
    '''
    params:
        n: int --- n-gram model you choose
        method: int ---- method == 0, use add_k_smoothing; method != 0, use␣
↪interpolation method.
        num_words: int --- total number of words
        valid_sents: list[list[str]] --- list of sentences
        unigram_dic: dict{ngram: counts} --- a dictionary that maps each 1-gram␣
↪to its number of occurences in "sents";
        bigram_dic: dict{ngram: counts} --- a dictionary that maps each 2-gram␣
↪to its number of occurence in "sents";
        trigram_dic: dict{ngram: counts} --- a dictionary that maps each 3-gram␣
↪to its number occurence in "sents";
        k: float --- The parameter of add_k_smoothing
        lam: list[float] --- a list of length 3. The parameter of interpolation.
↪
    return:
        ppl: float --- perplexity of valid_sents
    '''
    ppl = None
    ### YOUR CODE HERE

    loss = 0

    for s in valid_sents:
      ngram_set, ngram_dict = compute_ngram([s], n)
      for ngram in ngram_set:
        if method == 0:
          loss += np.log2(add_k_smoothing_ngram(ngram, k, num_words,␣
↪unigram_dic, bigram_dic, trigram_dic))
        else:
          loss += np.log2(interpolation_ngram(ngram, lam, num_words,␣
↪unigram_dic, bigram_dic, trigram_dic))

    loss = 1/num_words * loss
    ppl = np.exp2(-loss)

    ### END OF YOUR CODE
    return ppl
```

[22]:
```python
perplexity(1, 0, num_words, uni_valid_sents, unigram_dict, bigram_dict,␣
↪trigram_dict, k=0.1, lam=[0,0,1])
```

[22]: 1.633207766611748

1.1.7 Question 4 [code][written]

1. Based on add-k smoothing method, try out different $k \in [0.0001, 0.001, 0.01, 0.1, 0.5]$ and different n-gram model (unigram, bigram and trigram). Find the model and k that gives the best perplexity on "**valid_sents**" (smaller is better).
2. Based on interpolation method, try out different λ where $\lambda_1 = \lambda_2$ and $\lambda_3 \in [0.1, 0.2, 0.4, 0.6, 0.8]$. Find the λ that gives the best perplexity on "**valid_sents**" (smaller is better).
3. Based on the methods and parameters we provide, choose the method that peforms best on the validation data.

[23]:
```python
n = [1,2,3]
k = [0.0001, 0.001, 0.01, 0.1, 0.5]
### YOUR CODE HERE (add-k smoothing method)

best_result = sys.maxsize
best_n = None
best_k = None
for kidx in k:
    for nidx in n:
        ppl  = perplexity(nidx, 0, num_words, valid_sents, unigram_dict,
 ↪bigram_dict, trigram_dict, kidx, lam=[0,0,1])
        print(f"n: {nidx:.5f}\tk: {kidx:.5f}\tppl: {ppl}")
        if ppl < best_result:
            best_result = ppl
            best_k = kidx
            best_n = nidx
print(f"Best configuration:\tn={best_n}\tk={best_k}\tppl:{best_result}")

### END OF YOUR CODE
```

```
n: 1.00000      k: 0.00010      ppl: 1.6262061644707124
n: 2.00000      k: 0.00010      ppl: 1.9009594911957326
n: 3.00000      k: 0.00010      ppl: 2.286756189006186
n: 1.00000      k: 0.00100      ppl: 1.626205830873074
n: 2.00000      k: 0.00100      ppl: 1.8404684943161416
n: 3.00000      k: 0.00100      ppl: 2.2174616406014214
n: 1.00000      k: 0.01000      ppl: 1.6262025053794784
n: 2.00000      k: 0.01000      ppl: 1.8370860337051367
n: 3.00000      k: 0.01000      ppl: 2.2569516903801032
n: 1.00000      k: 0.10000      ppl: 1.6261702813659682
n: 2.00000      k: 0.10000      ppl: 1.9153013516752593
n: 3.00000      k: 0.10000      ppl: 2.380594422527923
n: 1.00000      k: 0.50000      ppl: 1.626048002285084
n: 2.00000      k: 0.50000      ppl: 2.0173626347004725
n: 3.00000      k: 0.50000      ppl: 2.497675008075584
Best configuration:     n=1     k=0.5   ppl:1.626048002285084
```

[24]:
```
lambda_3 = [0.1, 0.2, 0.4, 0.6, 0.8]
### YOUR CODE HERE (interpolation method)

best_result = sys.maxsize
best_lam = None

for lam in lambda_3:
    lam_12 = (1 - lam)/2
    lamb_list = [lam_12, lam_12, lam]

    result = perplexity(best_n, 1, num_words, valid_sents, unigram_dict,␣
 ↪bigram_dict, trigram_dict, k=best_k, lam=lamb_list)
    if result < best_result:
        best_result = result
        best_lam = lamb_list

print(f"Best configuration:\tlamba:{lamb_list}\tppl:{best_result}")

### END OF YOUR CODE
```

```
Best configuration:     lamba:[0.09999999999999998, 0.09999999999999998, 0.8]
ppl:1.6262062015488201
```

Based on the methods and parameters we provide, choose the method that peforms best on the validation data (**write your answer**):
The method that performs best on the validation data is **k-smoothing**, with k = 0.5, and n = 1. The perplexity of k-smoothing method is 1.6260 as compared to the perplexity of interpolation method, 1.6262, with lambda = [0.09999999999999998, 0.09999999999999998, 0.8]

1.1.8 Question 5 [code]

Evaluate the perplexity of the test data **test_sents** based on the best model you choose in **Question 4**.

[25]:
```
with open('data/wikitext-2/wiki.test.tokens', 'r', encoding='utf8') as f:
    text = f.readlines()
    test_sents = [line.lower().strip('\n').split() for line in text]
    test_sents = [s for s in test_sents if len(s)>0 and s[0] != '=']

uni_test_sents = pad_sents(test_sents, 1)
bi_test_sents = pad_sents(test_sents, 2)
tri_test_sents = pad_sents(test_sents, 3)
```

[26]:
```
### YOUR CODE HERE
uni_test_ppl = perplexity(1, 0, num_words, uni_test_sents, unigram_dict,␣
 ↪bigram_dict, trigram_dict, k=best_k, lam=best_lam)
bi_test_ppl = perplexity(2, 0, num_words, bi_test_sents, unigram_dict,␣
 ↪bigram_dict, trigram_dict, k=best_k, lam=best_lam)
```

```
tri_test_ppl = perplexity(3, 0, num_words, tri_test_sents, unigram_dict,␣
 ↪bigram_dict, trigram_dict, k=best_k, lam=best_lam)
print(f"Perplexity of the test data for unigram: \t{uni_test_ppl}")
print(f"Perplexity of the test data for bigram: \t{bi_test_ppl}")
print(f"Perplexity of the test data for trigram: \t{tri_test_ppl}")
### END OF YOUR CODE
```

```
Perplexity of the test data for unigram:        1.7195364564464566
Perplexity of the test data for bigram:         2.1983492534392517
Perplexity of the test data for trigram:        2.8512624014200725
```

1.2 Neural Language Model

We will create a LSTM language model as shown in figure and train it on the Wikitext-2 dataset. The data generators (train_iter, valid_iter, test_iter) have been provided. The word embeddings together with the parameters in the LSTM model will be learned from scratch.

Pytorch and torchtext are required in this part. Do not make any changes to the provided code unless you are requested to do so.

1.2.1 Question 6 [code]

- Implement the `__init__` function in `LangModel` class. *Note: the code implementation should allow switching between unidirectional LSTM and bidirectional LSTM easily*
- Implement the `forward` function in `LangModel` class.
- Complete the training code in train function and the testing code in test function.
- Train two models - **Unidirectional LSTM** and **Bidirectional LSTM**. Compute the perplexity of the test data "test_iter" using the trained models. The test perplexity of both trained models should be below 150.

Important Note: Make sure that "torchtext <= 0.11", as newer version might have torchtext.legacy removed

[27]:
```
!pip install -U torchtext==0.10.0
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Requirement already satisfied: torchtext==0.10.0 in
/usr/local/lib/python3.7/dist-packages (0.10.0)
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-
packages (from torchtext==0.10.0) (2.23.0)
Requirement already satisfied: tqdm in /usr/local/lib/python3.7/dist-packages
(from torchtext==0.10.0) (4.64.0)
Requirement already satisfied: torch==1.9.0 in /usr/local/lib/python3.7/dist-
packages (from torchtext==0.10.0) (1.9.0)
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages
(from torchtext==0.10.0) (1.21.6)
Requirement already satisfied: typing-extensions in
/usr/local/lib/python3.7/dist-packages (from torch==1.9.0->torchtext==0.10.0)
```

```
(4.2.0)
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in
/usr/local/lib/python3.7/dist-packages (from requests->torchtext==0.10.0)
(1.24.3)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.7/dist-packages (from requests->torchtext==0.10.0)
(2022.5.18.1)
Requirement already satisfied: chardet<4,>=3.0.2 in
/usr/local/lib/python3.7/dist-packages (from requests->torchtext==0.10.0)
(3.0.4)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-
packages (from requests->torchtext==0.10.0) (2.10)
```

[28]:
```
import torchtext
import torch
import torch.nn.functional as F
from torchtext.legacy.datasets import WikiText2
from torch import nn, optim
from torchtext.legacy import data
from nltk import word_tokenize
import nltk
import numpy as np
nltk.download('punkt')
torch.manual_seed(222)
```

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

[28]:
```
<torch._C.Generator at 0x7fa8991ae830>
```

[29]:
```
def tokenizer(text):
    '''Tokenize a string to words'''
    return word_tokenize(text)

START = '<START>'
STOP = '<STOP>'
#Load and split data into three parts
TEXT = data.Field(lower=True, tokenize=tokenizer, init_token=START,␣
 ↪eos_token=STOP)
train, valid, test = WikiText2.splits(TEXT)
```

[30]:
```
#Build a vocabulary from the train dataset
TEXT.build_vocab(train)
print('Vocabulary size:', len(TEXT.vocab))
```

```
Vocabulary size: 28907
```

[31]:
```
BATCH_SIZE = 64
# the length of a text feeding to the RNN layer
BPTT_LEN = 32
# train, validation, test data
train_iter, valid_iter, test_iter = data.BPTTIterator.splits((train, valid,␣
 ↪test),
                                                              ␣
 ↪batch_size=BATCH_SIZE,
                                                              ␣
 ↪bptt_len=BPTT_LEN,
                                                               repeat=False)
```

[32]:
```
#Generate a batch of train data
batch = next(iter(train_iter))
text, target = batch.text, batch.target
print('Size of text tensor',text.size())
print('Size of target tensor',target.size())
```

```
Size of text tensor torch.Size([32, 64])
Size of target tensor torch.Size([32, 64])
```

[33]:
```
class LangModel(nn.Module):
    def __init__(self, lang_config):
        super(LangModel, self).__init__()
        self.vocab_size = lang_config['vocab_size']
        self.emb_size = lang_config['emb_size']
        self.hidden_size = lang_config['hidden_size']
        self.num_layer = lang_config['num_layer']
        self.bidirectional = lang_config['bidirectional']

        self.embedding = None
        self.lstm = None
        self.linear = None

        ### TODO:
        ###    1. Initialize 'self.embedding' with nn.Embedding function and 2␣
 ↪variables we have initialized for you
        ###    2. Initialize 'self.lstm' with nn.LSTM function and 4 variables␣
 ↪we have initialized for you
        ###    3. Initialize 'self.linear' with nn.Linear function and 2␣
 ↪variables we have initialized for you
        ### Reference:
        ###        https://pytorch.org/docs/stable/nn.html

        ### YOUR CODE HERE (3 lines)
        self.embedding = nn.Embedding(self.vocab_size, self.emb_size)
```

```
        self.lstm = nn.LSTM(self.emb_size, self.hidden_size, self.num_layer,␣
↪bidirectional=self.bidirectional)
        self.linear = nn.Linear(2*self.hidden_size if self.bidirectional else␣
↪self.hidden_size, self.vocab_size)

        ### END OF YOUR CODE

    def forward(self, batch_sents, hidden=None):
        '''
        params:
            batch_sents: torch.LongTensor of shape (sequence_len, batch_size)
        return:
            normalized_score: torch.FloatTensor of shape (sequence_len,␣
↪batch_size, vocab_size)
        '''
        normalized_score = None
        hidden = hidden
        ### TODO:
        ###      1. Feed the batch_sents to self.embedding
        ###      2. Feed the embeddings to self.lstm. Remember to pass "hidden"␣
↪into self.lstm, even if it is None. But we will
        ###         use "hidden" when implementing greedy search.
        ###      3. Apply linear transformation to the output of self.lstm
        ###      4. Apply 'F.log_softmax' to the output of linear transformation
        ###
        ### YOUR CODE HERE (4 lines)

        x = self.embedding(batch_sents)
        out, hidden = self.lstm(x, hidden)
        score = self.linear(out)
        normalized_score = F.log_softmax(score, dim=-1)

        ### END OF YOUR CODE
        return normalized_score, hidden
```

[34]:
```
def train(model, train_iter, valid_iter, vocab_size, criterion, optimizer,␣
↪num_epochs):
    for n in range(num_epochs):
        train_loss = 0
        target_num = 0
        model.train()
        for batch in train_iter:

            text, targets = batch.text.to(device), batch.target.to(device)
            loss = None

```

```
            ### we don't consider "hidden" here. So according to the default␣
↪setting, "hidden" will be None
            ### YOU CODE HERE (~5 lines)

            optimizer.zero_grad()
            pred, _ = model(text)
            loss = criterion(pred.view(-1, vocab_size), targets.view(-1))
            loss.backward()
            optimizer.step()

            ### END OF YOUR CODE
            ##########################################
            train_loss += loss.item() * targets.size(0) * targets.size(1)
            target_num += targets.size(0) * targets.size(1)

        train_loss /= target_num

        # monitor the loss of all the predictions
        val_loss = 0
        target_num = 0
        model.eval()
        for batch in valid_iter:
            text, targets = batch.text.to(device), batch.target.to(device)

            prediction,_ = model(text)
            loss = criterion(prediction.view(-1, vocab_size), targets.view(-1))

            val_loss += loss.item() * targets.size(0) * targets.size(1)
            target_num += targets.size(0) * targets.size(1)
        val_loss /= target_num

        print('Epoch: {}, Training Loss: {:.4f}, Validation Loss: {:.4f}'.
↪format(n+1, train_loss, val_loss))
```

[35]:
```
def test(model, vocab_size, criterion, test_iter):
    '''
    params:
        model: LSTM model
        test_iter: test data
    return:
        ppl: perplexity
    '''
    ppl = None
    test_loss = 0
    target_num = 0
    with torch.no_grad():
        for batch in test_iter:
```

```
            text, targets = batch.text.to(device), batch.target.to(device)

            prediction,_ = model(text)
            loss = criterion(prediction.view(-1, vocab_size), targets.view(-1))

            test_loss += loss.item() * targets.size(0) * targets.size(1)
            target_num += targets.size(0) * targets.size(1)

        test_loss /= target_num

        ### Compute perplexity according to "test_loss"
        ### Hint: Consider how the loss is computed.
        ### YOUR CODE HERE(1 line)

        ppl = np.exp(test_loss)

        ### END OF YOUR CODE
        return ppl
```

[36]:
```
num_epochs=10
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
vocab_size = len(TEXT.vocab)
criterion = nn.NLLLoss(reduction='mean')

config = {
    'vocab_size':vocab_size,
    'emb_size':128,
    'hidden_size':128,
    'num_layer':1,
    'bidirectional': False
}

LM = LangModel(config)
LM = LM.to(device)
```

[37]:
```
optimizer = optim.Adam(LM.parameters(), lr=1e-3, betas=(0.7, 0.99))
```

[38]:
```
train(LM, train_iter, valid_iter, vocab_size, criterion, optimizer, num_epochs)
```

```
Epoch: 1, Training Loss: 6.0577, Validation Loss: 5.1698
Epoch: 2, Training Loss: 5.3880, Validation Loss: 4.9414
Epoch: 3, Training Loss: 5.1200, Validation Loss: 4.8541
Epoch: 4, Training Loss: 4.9522, Validation Loss: 4.8108
Epoch: 5, Training Loss: 4.8313, Validation Loss: 4.7831
Epoch: 6, Training Loss: 4.7345, Validation Loss: 4.7646
Epoch: 7, Training Loss: 4.6525, Validation Loss: 4.7527
Epoch: 8, Training Loss: 4.5823, Validation Loss: 4.7468
```

```
Epoch: 9, Training Loss: 4.5210, Validation Loss: 4.7446
Epoch: 10, Training Loss: 4.4664, Validation Loss: 4.7461
```

[51]:
```python
with open('./model/LSTM_model.pt', 'wb') as f:
  torch.save(LM.state_dict(), f)
  print("Model saved")
```

```
Model saved
```

[40]:
```python
test(LM, vocab_size, criterion, test_iter)
```

[40]:
```
99.29450889854887
```

[41]:
```python
config = {
    'vocab_size':vocab_size,
    'emb_size':128,
    'hidden_size':128,
    'num_layer':1,
    'bidirectional': True
}

biLSTM = LangModel(config)
biLSTM = biLSTM.to(device)

biLSTM_optimizer = optim.Adam(biLSTM.parameters(), lr=1e-4, betas=(0.7, 0.99))
```

[42]:
```python
train(biLSTM, train_iter, valid_iter, vocab_size, criterion, biLSTM_optimizer,
  num_epochs)
```

```
Epoch: 1, Training Loss: 6.3001, Validation Loss: 4.4504
Epoch: 2, Training Loss: 4.4463, Validation Loss: 3.6178
Epoch: 3, Training Loss: 3.7648, Validation Loss: 3.1029
Epoch: 4, Training Loss: 3.2507, Validation Loss: 2.6631
Epoch: 5, Training Loss: 2.7988, Validation Loss: 2.2787
Epoch: 6, Training Loss: 2.4088, Validation Loss: 1.9577
Epoch: 7, Training Loss: 2.0814, Validation Loss: 1.6955
Epoch: 8, Training Loss: 1.8105, Validation Loss: 1.4843
Epoch: 9, Training Loss: 1.5873, Validation Loss: 1.3136
Epoch: 10, Training Loss: 1.4005, Validation Loss: 1.1746
```

[43]:
```python
with open('./model/biLSTM_model.pt', 'wb') as f:
  torch.save(biLSTM.state_dict(), f)
  print("Model saved")
```

```
Model saved
```

[44]:
```python
test(biLSTM, vocab_size, criterion, test_iter)
```

[44]: 3.1091208393789365

1.2.2 Question 7 [code][written]

When we use trained language model to generate a sentence given a start token, we can choose `greedy search`.

As shown above, `greedy search` algorithm will pick the token which has the highest probability and feed it to the language model as input in the next time step. The model will generate `max_len` number of tokens at most.

- Implement `word_greedy_search`

[45]:
```
def word_greedy_search(model, start_token, max_len):
    '''
    param:
        model: nn.Module --- language model
        start_token: str --- e.g. 'he'
        max_len: int --- max number of tokens generated
    return:
        strings: list[str] --- list of tokens, e.g., ['he', 'was', 'a',␣
↪'member', 'of',...]
    '''
    model.eval()
    ID = TEXT.vocab.stoi[start_token]
    strings = [start_token]
    hidden = None

    ### You may find TEXT.vocab.itos useful.
    ### YOUR CODE HERE

    predicts = torch.ones(1, 1).long().to(device) * ID
    for _ in range(max_len):
        out, _ = model(predicts, hidden)
        predicts = torch.argmax(out[-1,:,:], dim=-1)
        strings.append(TEXT.vocab.itos[predicts.cpu().numpy()[0]])
        if strings[-1] == '<eos>':
          break
        predicts.unsqueeze_(0)

    ### END OF YOUR CODE
    print(strings)
```

To read the trained parameters

[46]:
```
#LM.load_state_dict(torch.load("model/LSTM_model.pt",map_location = torch.
↪device('cpu')))
```

```
#biLSTM.load_state_dict(torch.load("model/biLSTM_model.pt",map_location = torch.
↪device('cpu')))
```

[47]:
```
word_greedy_search(LM, 'he', 64)
```

```
['he', 'was', 'the', 'first', 'time', '.', '<eos>']
```

Review Question: Based on your understanding, can we use the Bidirectional LSTM for this language generation (decoding) task? Explain why? write your explanation: No we cannot. Bi-LSTM consists of two layers of LSTM, hence its hidden state would consists of both the trained parameters from two LSTM. However, this tasks only proceed in thew forward direction as we start from a start token, then generate the sentence after the token.

1.2.3 Question 8 [code][written]

- We will use the hidden vectors (the working memory) of LSTM as the contextual embeddings. Implement `contextual_embedding` function.
- Use the `contextual_embedding` function to get the contextual embeddings of the word "sink" in four sequences "wood does not sink in water", "a small water leak will sink the ship", "there are plates in the kitchen sink" and "the kitchen sink was full of dirty dishes". Then calculate the cosine similarity of "sink" from each pair of sequences. Assume that w_1 and w_2 are embeddings of "sink" in sequences "wood does not sink in water" and "a small water leak will sink the ship" respectively. The cosine similarity can be calculated as

$$similarity = cos(\theta) = \frac{w_1^T w_2}{||w_1||_2||w_2||_2} \tag{9}$$

Give the explanation of the results.

[48]:
```
def contextual_embedding(model, sentence):
    '''
    params:
        model: nn.Module --- language model
        sentence -- list[str]: list of tokens, e.g., ['I', 'am',...]
    return:
        embeddings -- numpy array of shape (length of sentence, word embedding␣
↪size)
    '''
    model.eval()
    hidden = None

    ### YOUR CODE HERE

    temp = sentence.split(" ")
    target = temp.index("sink")
    sentence = temp[:target] + ["sink"]

    ID = []
```

```
    for t in sentence:
      ID.append(TEXT.vocab.stoi[t])

    word = torch.LongTensor([ID]).to(device)
    _, embed = model(word)

    output = embed[-1].cpu().detach().numpy()
    embeddings = output[0]

    ### END OF YOUR CODE
    return embeddings
```

[49]:
```
def cosine_sim(w1, w2):
  w1 = np.mean(w1, 0)
  w2 = np.mean(w2, 0)
  return np.dot(w1, w2) / (np.linalg.norm(w1) * np.linalg.norm(w2))
```

[50]:
```
sink_seq1 = "wood does not sink in water"
sink_seq2 = "a small water leak will sink the ship"
sink_seq3 = "there are plates in the kitchen sink"
sink_seq4 = "the kitchen sink was full of dirty dishes"

### YOUR CODE HERE

embed_1 = contextual_embedding(LM, sink_seq1)
embed_2 = contextual_embedding(LM, sink_seq2)
embed_3 = contextual_embedding(LM, sink_seq3)
embed_4 = contextual_embedding(LM, sink_seq4)

print("1 and 2:\t", cosine_sim(embed_1, embed_2))
print("2 and 3:\t", cosine_sim(embed_2, embed_3))
print("3 and 4:\t", cosine_sim(embed_3, embed_4))
print("1 and 3:\t", cosine_sim(embed_1, embed_3))
print("2 and 4:\t", cosine_sim(embed_2, embed_4))
print("1 and 4:\t", cosine_sim(embed_1, embed_4))

### END OF YOUR CODE
```

```
1 and 2:         0.64669484
2 and 3:         0.6001737
3 and 4:         0.7029462
1 and 3:         0.63270664
2 and 4:         0.6446002
1 and 4:         0.57950085
```

write your explanation:

The cosine similarities between the pair 3 and 4 are the highest. The reason for this could be that the word `sink` in both sentences is noun. Whereas the similarity between 2 and 3 are the lowest, as the word `sink` in sequence 2 is a verb, indicating an action while in sequence 3, the word `sink` is a noun.

Review Question: Based on your understanding, can we use the Bidirectional LSTM for this contextual embedding task? Explain why? write your explanation:
Yes, we can, but certain changes are needed. The sequence is truncated to be stopped at the word `sink`(e.g. ['wood', 'does', 'not', 'sink'] for sequence 1) However, in the case of biLSTM, we need to make use of the sequence after the word `sink` to obtain the hidden state in the backward direction. By combining the forward and backward hidden state, generated by the sequence before and after the word `sink` respectively, we can get a better embedding.

1.2.4 Requirements:

- This is an individual report.
- Complete the code using Python.
- List students with whom you have discussed if there are any.
- Follow the honor code strictly.

1.2.5 Free GPU Resources

We suggest that you run neural language models on machines with GPU(s). Google provides the free online platform Colaboratory, a research tool for machine learning education and research. It's a Jupyter notebook environment that requires no setup to use as common packages have been pre-installed. Google users can have access to a Tesla T4 GPU (approximately 15G memory). Note that when you connect to a GPU-based VM runtime, you are given a maximum of 12 hours at a time on the VM.

It is convenient to upload local Jupyter Notebook files and data to Colab, please refer to the tutorial.

In addition, Microsoft also provides the online platform Azure Notebooks for research of data science and machine learning, there are free trials for new users with credits.