Shanghai Jiao Tong University

Smoothing Algorithm for n-gram Model

Author: *Zhiteng Li, Xiangge Huang, Huangji Wang*

Course: *Fall 2021, CS3602-2: Natural Language Processing*
Date: *October 12, 2021*

Contents

1. Program Basis

1.1. Background.

The language model construction is one of the most crucial parts in natural language processing, and now the most applied algorithm to solve this corresponding language term is to use the probability model to evaluate the natural language. According to the request for this project, we are expected to build our own n-gram language model on the basis of the word data and use relative smoothing way to optimize it.

Based on the analysis ahead, we use the corresponding method to achieve our 3-gram language model and evaluate the language part parameter perplexity.

1.2. Environment.

We use these environments and tools to finish this project.

- Python 3.8.7 and vscode in Windows 10
- Language Model: kenlm in Ubuntu 18.04
- Data Filter Tool: hash table

1.3. Division.

- Zhiteng Li:
  - Build n-gram model.
  - Design additive smooth, interpolation and Good-Turing discounting.
  - Code writing.
- Xiangge Huang:
  - Build 2-gram model.
  - Calculate the perplexity in the test set and train set in general way.
  - Paper writing.
- Huangji Wang:
  - Calculate the perplexity with this kenlm model.
  - Design additive smooth and test smooth interpolation.
  - Paper writing.

1.4. Model Description.

On the basis of the 2-gram model fundamental, we design one type of discounting algorithms and build our higher level n-gram language model and then use the train set to instantiate it. Then, after we get our n-gram and achieve the language word probability, we give the perplexity of the test set.

The basic n-gram language model can be described as:

$$P(w_k|w_{k-n+1},...,w_{k-1})$$

This formula means when the current word probability is only decided by its former $n-1$ words. The picture shows the 2-gram model as below.

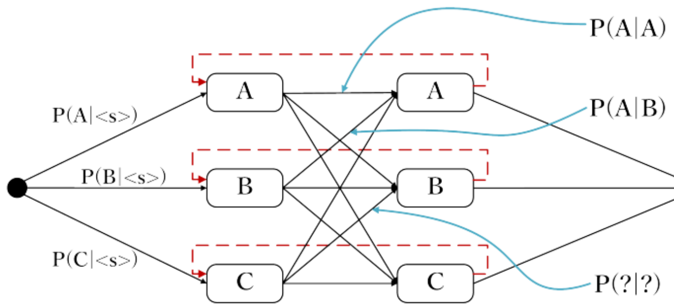


Figure 1: 2-gram model

Obviously, in this 2-gram model, the latter words A, B and C emergence can be linked to and depend on the one word before it. It can be expressed by the conditional probability between these two words. So now we can have our own 2-gram model on this theory and deepen it into n-gram model to pursue more precision in expressing our language model.

1.5. Perplexity.

Additionally, after the building of the language probability model, the most important work is to evaluate the language data through our probability model, and we take perplexity to illustrate and quantify its demerits. Generally, the definition of perplexity is

$$PPL = \prod_{k=1}^{K} P(w_k|w_{k-n-1}^{k-1})^{-\frac{1}{K}}$$

This formula is based on the n-gram model, and also we can use kenlm library to get the perplexity easily. But because of the lacking of arpa form in our model, so the terminal outcome is calculated by our own perplexity calculation function.

2. N-gram Language Model

2.1. Foundation: 2-gram.

We build our language model from the easier 2-layer part, which is every word except the start single is only related to its former one word. So we can show the model and perplexity respectively.

$$P(w_k|w_{k-1}) \qquad PPL = \prod_{k=1}^{K} P(w_k|w_{k-1})^{-\frac{1}{K}}$$

Use some words in the training data set as examples, the sentence

$$anarchism\ originated\ as\ a\ term\ of\ abuse$$

is the first vocabulary in this test set, and it will be transferred to these model

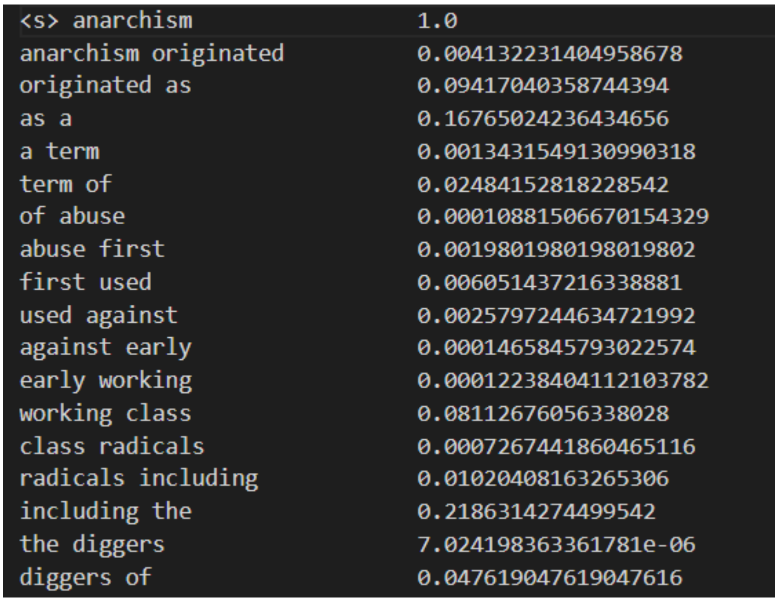
$$'<s>'\ \ 'anarchism'\ 'originated'\ 'as'\ 'a'\ 'term'\ 'of'\ 'abuse'$$

so we can construct its corresponding probabilities

$$P(anarchism|<s>) \quad P(originated|anarchism)$$

and others.Typically, the first part $P(<s>)$ is always thrown due to its value approaches 0 and it will lose the PPL precision.

Additionally, some probabilities used in 2-gram model are shown in Figure 2.



<s> anarchism	1.0
anarchism originated	0.004132231404958678
originated as	0.09417040358744394
as a	0.16765024236434656
a term	0.0013431549130990318
term of	0.02484152818228542
of abuse	0.00010881506670154329
abuse first	0.0019801980198019802
first used	0.006051437216338881
used against	0.0025797244634721992
against early	0.0001465845793022574
early working	0.00012238404112103782
working class	0.08112676056338028
class radicals	0.0007267441860465116
radicals including	0.01020408163265306
including the	0.2186314274499542
the diggers	7.024198363361781e-06
diggers of	0.047619047619047616

Figure 2: 2-gram probability

2.2. Deepening: n-gram.

After we build our 2-gram model, what we need to do is to deepen it and make it suitable for presenting our large data, so we build more deeper language model to solve this problem. And considering the data just consists of one long sentence, we believe the 3-gram model is enough to represent it, so we add one layer on the foundation of our 2-gram model. And Figure 3 exhibits some probabilities derived from this model.

```
<s> anarchism originated        1.0
anarchism originated as         1.0
originated as a                 0.40476190476190477
as a term                       0.0016281158769368964
a term of                       0.1232091690544126
term of abuse                   0.034482758620689655
of abuse first                  0.019230769230769232
abuse first used                1.0
first used against              0.007142857142857143
used against early              0.02127659574468085
against early working           1.0
early working class             1.0
working class radicals          0.006944444444444444
class radicals including        0.5
radicals including the          1.0
including the diggers           0.0005984440454817474
the diggers of                  0.16666666666666666
diggers of the                  1.0
```

Figure 3: 3-gram probability

However, because of our language model is so large that the sparsity will occur easily, and according to the requirement of implementing smoothing algorithm, we transfer our class model expression to hash map and build the relative probabilities data. And the raw probabilities in 3-gram model can be

$$P(w_3|w_1w_2) = \frac{C(w_1w_2w_3)}{C(w_1w_2)}$$

And the smoothing algorithm mentioned below will solve the sparsity including unseen and low frequency. Also, considering our control group kenlm 3-gram language model, due to its probabilities log expression, we provide the equivalent form of PPL to make our instructions clear and scientific.

The original form and log equivalent form of PPL for 3-gram model is

$$PPL = \prod_{k=1}^{K} P(w_k|w_{k-1}w_{k-2})^{-\frac{1}{K}}$$

$$\log PPL = -\frac{1}{K}\sum_{k=1}^{K} \log P(w_k|w_{k-1}w_{k-2})$$

3. Optimization: Smoothing

The irrationality of data frequency is the fundamental reason why we adopt smoothing method, which has been exhibited in the raw probabilities ahead, so we use these ways to smooth our probability model.

Actually, we take three methods to realize the discounting algorithm. The first is k-additive smoothing. The second is Jelinek-Mercer smoothing. And the last is Good-Turing discounting. All the smoothing algorithms are based on the n-gram model.

3.1. K-additive Smoothing.

For normal additive smoothing, it sets $k = 1$. If $0 < k \leq 1$, the test result must be nicer. Thus, we try to traverse the test set and judge whether each 3-word phrase lies in the n-gram model. If the phrase is not in the model, we give it a low weight 0 and then it can be done by k-additive smoothing.

$$P(w_i|w_{i-n+1}^{i-1}) = \frac{k + C(w_{i-n+1}^{i-1})}{k|V| + \Sigma_{w_i} C(w_{i-n+1}^{i})}, 0 < k \leq 1$$

Difficulties

Researchers have found that when the situation of $P(w_i) = 0$ occurs in k-additive smoothing, the result may be seriously bad. To deal with this problem, we firstly give each 3-word phrase one enough weight. However, if the 2-word phrase is not in the train set, our algorithm will leads to terrible error. Thus, we give each 2-word phrase that is not in the model 0 weight.

```
# pass in the initial value of para k
instance1 = Lidstone_smoothing(dev_data, k=0.5)

# smoothing
instance1.smoothing(hash2)

# calculate the PPL of test_set
res1 = instance1.calculate_PPL()
print(res1)
```

```
... 40.482017237459445
```

Figure 4: The PPL of K-additive smoothing.

3.2. Jelinek-Mercer Smoothing (Interpolation).

Actually, when we do k-additive smoothing, we rudely set two 3-word phrases that are not in the n-gram model the same weight and possibility. However, this result is obviously ambiguous. For example, ”ZZZ” and ”HAPPY” might not be totally equal in the same text. Thus, the Jelinek-Mercer Smoothing is to do interpolation in n-gram models. It can efficiently handle with this problem.

$$P^*(w_i|w_{i-1}) = \lambda P(w_i|w_{i-1}) + (1-\lambda)P(w_i)$$

In our 3-gram model, we hope to apply this method in our 3-gram model. Then, we apply the high order interpolation in our model. The key problem is to find good λ_n.

$$P^*(w_i|W_{i-n+1}^{i-1}) = \lambda_n P(w_i|W_{i-n+1}^{i-1}) + (1-\lambda_n)P(w_i|W_{i-n+2}^{i-1})$$

To train λ_n, we use one good method called "Gradient Descent". Gradient descent is widely used in unconstrained optimization problems. For convex differential f,

$$d_k \text{ is a descent direction} \Leftrightarrow d_k^T f(x_k) < 0$$

Algorithm 1 Descent Method

choose initial point $x_0 \in R^n$
repeat
  choose **descent direction** $d_k \in R^n$ and **step size** $t_k > 0$
  $x_{k+1} = x_k + t_k d_k$ s.t. $f(x_{k+1}) < f(x_k)$
until stopping criterion is satisfied

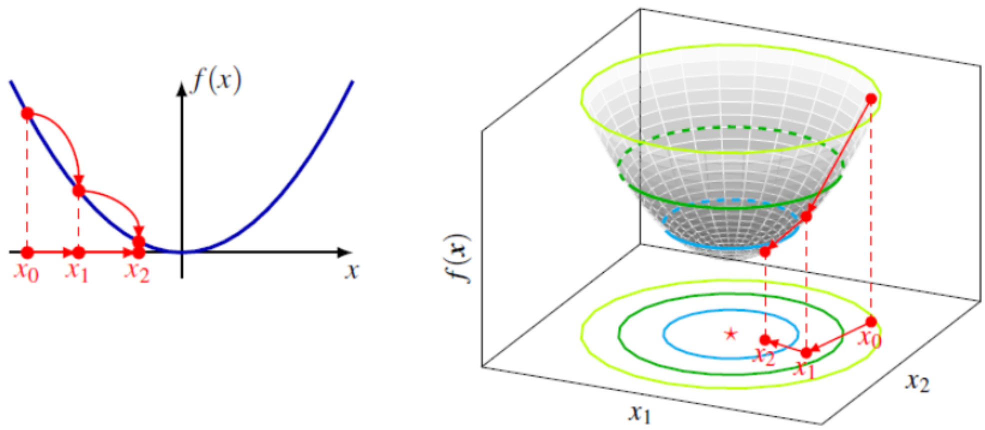


Figure 5: Principle of gradient descent

In our model, we calculate PPL by the formula:

$$PPL = (10^{-\Sigma \log(\lambda_2 P_3 + \lambda_1(1-\lambda_2)P_2 + (1-\lambda_1)(1-\lambda_2)P_1})^{\frac{1}{\text{length}}}$$

Luckily, it's obvious that the function of $f(\lambda_1, \lambda_2)$ is convex because the Hessian matrix of $f(\lambda_1, \lambda_2)$ is positive definite($0 \leq P_1, P_2, P_3 \leq 1$). It means we can use descent method directly and it will lead to perfect result.

Difficulties

Due to the reason that the function is discrete, if we want to build one complete function, the cost of gradient calculation must be terribly high. Thus, we apply **Expectation-maximization algorithm** in our gradient descent method to improve the efficiency. And this idea leads to nice result. We firstly fix one parameter and do gradient descent in another parameter. Next, we fix this unfixed parameter and let another parameter find the gradient descent. We repeat these methods to find the local minimum parameters. Due to the reason that $f(\lambda_1, \lambda_2)$ is convex, the local minimum is actually the global minimum.

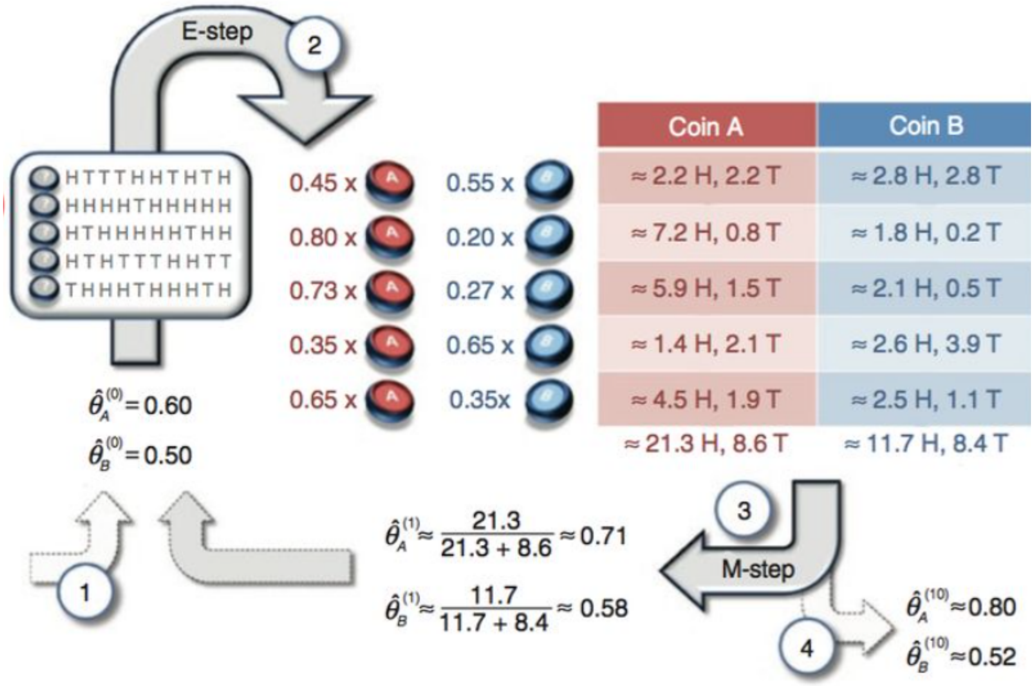


Figure 6: Example of EM algorithm

At last, the result is very nice due to the combination of gradient descent algorithm and EM algorithm.

```
# snd test the model on the test_set with paras lambda_1 and lambda_2

# pass in the initial values of lambda_1 and lambda_2
instance2 = Interpolation_smoothing(dev_data, test_data, lambda_1=0.5, lambda_2=0.5)

# train lambda_1 and lambda_2 on the dev_set
instance2.train_paras(hash1, hash2, hash3, dev_data, maxiter=1000, stepSize=0.05, tol=1e-5)

# calculate PPL on the test_set
res2 = instance2.calculate_PPL(hash1, hash2, hash3)
print("PPL of test_set: {}".format(res2))

...  ------------------------------------
lambda_1: 0.5818039058034233
lambda_2: 0.18286565390627954
410.50279117052855
```

Figure 7: The optimal value of λ_1 and λ_2 and the PPL of our interpolation algorithm.

3.3. Good Turing Discounting.

The main idea is to redistribute the words of $r+1$ times to the words of r times in the n-gram model. To finish this discounting algorithm, we should firstly modify N_r.

$$r^* = (r+1)\frac{N_{r+1}}{N_r} \text{ and } d(r) = \frac{(r+1)N_{r+1}}{rN_r} \text{ and } P(v : C(v) = r) = \frac{r^*}{N}$$

r	Symbols	N_r	rN_r	$P(x)$
0	C, H	2	0	0
1	A, B, E, I	4	4	4/13
2	D, F, G	3	6	6/13
3	J	1	3	3/13
Total			13	1

$\rightarrow$

r^*	Symbols	N_r	rN_r	$P(x)$
2	C, H	2	4	4/13
3/2	A, B, E, I	4	6	6/13
1	D, F, G	3	3	3/13
1	J	1	0	0
Total			13	1

Figure 8: The example of Good Turing Discounting.

Based on these two previous algorithms, we finish this algorithm very quickly. The result is putted below.

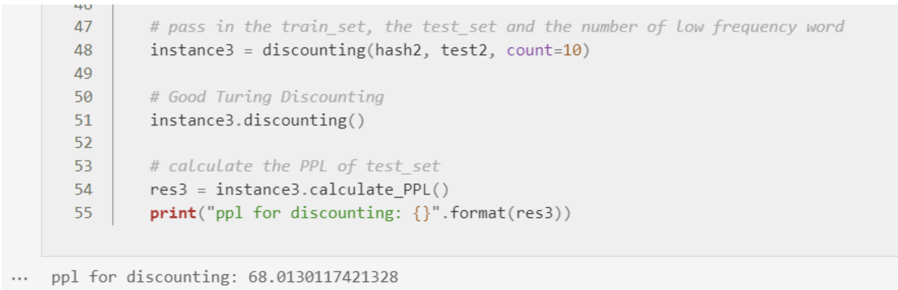


Figure 9: The PPL of Good Turing Discounting.

3.4. Katz's Back-off Model: Kenlm Model.

Based on the existing model: kenlm model, we build the **arpa** file of the train set and put the test set into the model. And the result is clearly good.

Figure 10: The PPL of kenlm model

3.5. Comparison.

Above all the result, we find that efficiency satisfies that (We don't consider K-additive smoothing because it isn't correct when it meets 0 possibility):

$$\text{Good Turing Discounting} > \text{Kenlm Model} > \text{Interpolation}$$

4. Discussion

The main goal of this lab is to understand n-gram model and language perplexity. By using different kinds of discounting algorithms, we get some efficient perplexities, and then choose the best of them according to the development data. For some details in this lab, we have queried class TA and got the corresponding response. Now, we will discuss the pros and cons of this model and explain the difficulties and solutions we have met and done.

4.1. Strengths.

1. Plenty Types of Smoothing and Nice Control Group

   We have finished 3 types of smoothing algorithms and run them well. To test our improved results, we use the existing kenlm model to calculate the back-off PPL and compare it with our three PPL results. It shows that some of our PPL results are good while others are not good enough.

2. Use **dev-set** to Train Parameters

   We make full use of three sets the teacher provided. The dev-set is well applied in our parameters training. λ_1 and λ_2 these two parameters have good convergence. Because the dev-set has no connection with the train-set, the parameters have excellent independence.

3. More Real Condition: 3-gram

   The information of 3-gram is much larger than the information of 2-gram. It means the prefixes of 3-gram is more complex than 2-gram. The more information we get, the realer result we have. If we want the real condition, then n in n-gram model must lead to infinity, which is impossible in real situation. Thus, we use 3-gram that is easy to finish and calculate. The PPL is less than the PPL in 2-gram and 1-gram.

4. Good Function Gradation

   We try to make all the codes simpler and cleaner to be understood. Thus, our functions in the codes has good hierarchy. Our codes are easy to debug and read.

4.2. Weaknesses.

1. Slow in Gradient Descent

   The compute complexity of gradient descent is incredibly high. Thus when we execute the code, it runs very slow and often costs 6-8 minutes. Anyway, the result is precise.

2. Lack of 0 Possibility in K-additive Smoothing

   We rudely set the items with 0 possibility. And in smoothing method we give it a k-weight. The result may be ambiguous due to the smoothing method. It means that the model is in bad conditions when it meets $P = 0\%$ item.

5. Conclusion

Briefly speaking, we have built our n-gram language model, implemented some smoothing algorithms and tested relative perplexities based on test data. The key point to this lab is to set a rational smoothing algorithm to let the language perplexities decrease so the whole model sparsity problem can be solved. So we use four traditional smoothing algorithm to finish this work, and the correlative outcomes have shown in part Smoothing. Additionally, we also have researched other smoothing algorithms like Absolutely discounting, while they are not presented in our major work.

In general, on the basis of the data provided, we construct our final 3-gram language model and then use smoothing way to decrease its low frequency. After that we add the train data to initialize our model and acquire its probabilities values. Also, through development data we choose the best smoothing algorithm in the combination. Finally, we reach the final goal returning the language perplexity on the test data and then analyze this whole project.

As the process of calculating the perplexities, we use kenlm library to aid and check the rationality of our results. Considering the library situation that it uses Backoff algorithm to build language model, we notice that our smoothing results are expected to be similar with it in numerical terms, and the before-smoothing results are supposed to be larger than it. And recalling our explanation of these smoothing perplexities, we can be sure that our smoothing algorithm is useful in the 3-gram language model.

A. Appendix for Codes

A.1. Kenlm Model in Ubuntu 18.08.

The code for build the arpa set:

bin/lmplz -o 3 –verbose header –text mytext/19-01-au.txt –arpa mylmmodel/test.arpa

```
import kenlm
model = kenlm.LanguageModel("test_train.bin")
f =  open("test_set.txt","r")
s = f.read()
print(model.perplexity(s))
```

Listing 1: **Kenlm.py**

A.2. Three Type of Smoothing methods.

We finish these methods in the **ipynb** format. The codes in **py** format are presented below.

```
# library
import math

# Lidstone Smoothing
class Lidstone_smoothing:
    def __init__(self, dev_data, k):

        self.dev_data = dev_data

        # log10 probability of each 3 words phase
        self. hash = {}

        # para, k=1 is Laplacian Smoothing
        self.k = k

        # len of dev_data
        self. len =  len(self.dev_data)

    def smoothing(self, hash2):
        for i in  range(2, self. len):
            # the previous two words
            prefix = self.dev_data[i-2] + ' ' + self.dev_data[i-1]
            # 3 words phase
            tmp = self.dev_data[i-2] + ' ' + self.dev_data[i-1] + ' '  + self.dev_data[i]
            # prefix not in hash2
            if (prefix not in hash2):
                pass
            elif (tmp not in hash2[prefix]):
                hash2[prefix][tmp] = 0

        for key2 in hash2:
```

```
            # number of str start in prefix "key2"
            v =  len(hash2[key2].keys())
            m =  sum(hash2[key2].values())
            for key3 in hash2[key2]:
                # Lidstone Smoothing
                self. hash[key3] = math.log10((hash2[key2][key3] + self.k) / (m + v*self.k))

    def calculate_PPL(self):
        # calculate the PPL
        ppl = 0

        # number of 3 words phase
        cnt = 0

        for i in  range(2, self. len):
            # 3 words phase
            tmp = self.dev_data[i-2] + ' ' + self.dev_data[i-1] + ' '  + self.dev_data[i]
            if (tmp not in self. hash):
                continue
            ppl += self. hash[tmp]
            cnt += 1

        ppl *= -1
        ppl /= cnt
        ppl = 10**ppl

        return ppl

# Jelinek-Mercer Smoothing
class Interpolation_smoothing:
    def __init__(self, dev_data, test_data, lambda_1, lambda_2):
        self.dev_data = dev_data
        self.test_data = test_data

        # para lambda, learn from dev_set
        self.lambda_1 = lambda_1
        self.lambda_2 = lambda_2

        # len of dev_data
        self. len =  len(self.dev_data)

    # maxiter, stepSize and tol represent maxDepth of iteration, update step size and stop flag respectively
    def train_paras(self, hash1, hash2, hash3, dev_data, maxiter, stepSize, tol):

        # gradient decent for lambda_2
        def gd2(lambda_1, lambda_2):
            res = 0

            # number of derivative
            cnt = 0

            for i in  range(2,  len(dev_data)):
                word3 = dev_data[i-2] + ' ' + dev_data[i-1] + ' '  + dev_data[i]
                word2 = dev_data[i-1] + ' '  + dev_data[i]
                word1 = dev_data[i]
```

```
                # probability of different length phase
                p3 = hash3[word3][1] if word3 in hash3 else 0
                p2 = hash2[word2][1] if word2 in hash2 else 0
                p1 = hash1[word1] if word1 in hash1 else 1

                # sum the derivative of each phase
                res += (p3-lambda_1*p2-(1-lambda_1)*p1)/(lambda_2*p3 + lambda_1*(1-lambda_2)*p2 + (1-lambda_1)*(1-lambda_2)*p1)
                cnt += 1

            # get the avg of PPL derivative
            res /= -cnt

            return res

        # gradient decent for lambda_1
        def gd1(lambda_1, lambda_2):
            res = 0

            # number of derivative
            cnt = 0

            for i in  range(2,  len(dev_data)):
                word3 = dev_data[i-2] + ' ' + dev_data[i-1] + ' '  + dev_data[i]
                word2 = dev_data[i-1] + ' '  + dev_data[i]
                word1 = dev_data[i]

                # probability of different length phase
                p3 = hash3[word3][1] if word3 in hash3 else 0
                p2 = hash2[word2][1] if word2 in hash2 else 0
                p1 = hash1[word1] if word1 in hash1 else 1

                # sum the derivative of each phase
                res += ((1-lambda_2)*p2-(1-lambda_2)*p1)/(lambda_2*p3 + lambda_1*(1-lambda_2)*p2 + (1-lambda_1)*(1-lambda_2)*p1)
                cnt += 1

            # get the avg of PPL derivative
            res /= -cnt

            return res


        # fst train para lambda_1 and lambda2 on the dev_set

        # EM algorithm - fst fix one var and update another, then fix another var and update the previous one
        while (gd2(self.lambda_1, self.lambda_2)**2 >= tol or gd1(self.lambda_1, self.lambda_2)**2 >= tol):
            # update lambda_2
            cnt = 0
            gd = gd2(self.lambda_1, self.lambda_2)
            while (cnt <= maxiter and gd**2 >= tol):
                # update lambda_2 based on the gradient
                self.lambda_2 -= gd*stepSize

```

```python
                gd = gd2(self.lambda_1, self.lambda_2)
                cnt += 1

            # update lambda_1
            cnt = 0
            gd = gd1(self.lambda_1, self.lambda_2)
            while (cnt <= maxiter and gd**2 >= tol):
                # update lambda_1 based on the gradient
                self.lambda_1 -= gd*stepSize

                gd = gd1(self.lambda_1, self.lambda_2)
                cnt += 1


        print("---------------------------------------")
        print("lambda_1: {}". format(self.lambda_1))
        print("lambda_2: {}". format(self.lambda_2))


    def calculate_PPL(self, hash1, hash2, hash3):
        # calculate the PPL
        ppl = 0

        for i in  range(2,self. len):
            word3 = self.test_data[i-2] + ' ' + self.test_data[i-1] + ' '  + self.test_data[i]
            word2 = self.test_data[i-1] + ' '  + self.test_data[i]
            word1 = self.test_data[i]

            # probability of different length phase
            p3 = hash3[word3][1] if word3 in hash3 else 0
            p2 = hash2[word2][1] if word2 in hash2 else 0
            p1 = hash1[word1] if word1 in hash1 else 1

            # interpolation
            # lambda_1 and lambda_2 are learned from dev_set
            ppl += math.log10(self.lambda_2*p3 + self.lambda_1*(1-self.lambda_2)*p2 + (1-self.lambda_1)*(1-
                self.lambda_2)*p1)

        ppl *= -1
        ppl /= (self. len-2)
        ppl = 10**ppl

        return ppl

# Good Turing Discounting
class discounting:
    def __init__(self, hash2, test2, count):
        # train_set
        self.hash2 = hash2

        # test_set
        self.test2 = test2

        # define the number of low frequency word
        self.numlow = count
```

```
        # record probability of 3 word phase
        self.hash3 = {}

    def discounting(self):
        for key2 in hash2:
            if (key2 not in self.test2):
                continue

            # find the maximum number of occurrences in train_set
            values =  list(self.hash2[key2].values())
            m =  max(values)

            # vocabulary size
            v =  sum(self.hash2[key2].values())

            # probability array
            p = [0 for _ in  range(m+1)]

            # record the number of words of the same occurrences
            cnt = [0 for _ in  range(m+1)]

            # record the number of words of the same occurrences
            for word3 in self.test2[key2]:
                if (word3 not in self.hash2[key2]):
                    cnt[0] += 1
                else:
                    cnt[self.hash2[key2][word3]] += 1

            # Good Turing Discounting for lower frequency word
            n =  min(m, self.numlow)
            for i in  range(n):
                if (cnt[i] == 0):
                    p[i] = 0
                else:
                    j = i+1
                    while (j < n-1 and cnt[j]==0):
                        j += 1
                    p[i] = (j)*cnt[j]/cnt[i] / v

            # the higher item retains the original probability
            if (n < m+1):
                for i in  range(n, m+1):
                    p[i] = cnt[i] / v

            # normalize to ensure the sum of probability is 1
            ss =  sum(p)
            if (ss == 0):
                continue
            for i in  range(m+1):
                p[i] /= ss

            # record probability of 3 word phase
            for word3 in self.test2[key2]:
                if (word3 not in self.hash2[key2]):
                    self.hash3[word3] = p[0]
                else:
```

```
                    self.hash3[word3] = p[self.hash2[key2][word3]]

    def calculate_PPL(self):
        # calculate the PPL
        ppl = 0

        cnt = 0

        for key2 in self.test2:
            for word3 in self.test2[key2]:
                if (word3 not in self.hash3 or self.hash3[word3]==0):
                    continue
                ppl += math.log10(self.hash3[word3])
                cnt += 1

        ppl /= -cnt
        ppl = 10**ppl

        return ppl

# align
length = 30
if __name__ == '__main__':

    train_data = []

    # f = open("hw1_dataset\\train_set.txt", "r")
    with  open(r"hw1_dataset\train_set.txt", "r") as f:
        # read by line
        for line in f:
            train_data.append(line)

    # split by whitespace
    train_data = train_data[0].split(" ")

    # begin and end symbol
    train_data.insert(0, '<s>')
    train_data.append('</s>')

    dev_data = []

    with  open(r"hw1_dataset\dev_set.txt",'r') as f:
        # read by line
        for line in f:
            dev_data.append(line)

    # split by whitespace
    dev_data = dev_data[0].split(" ")

    # insert begin and end symbols
    dev_data.insert(0, '<s>')
    dev_data.append('</s>')

    test_data = []

    with  open(r"hw1_dataset\test_set.txt",'r') as f:
```

```python
        # read by line
        for line in f:
            test_data.append(line)

    # split by whitespace
    test_data = test_data[0].split(" ")

    # insert begin and end symbols
    test_data.insert(0, '<s>')
    test_data.append('</s>')

    # test the Lidstone Smoothing algorithm

    # record 3 words phase with its prefix
    hash2 = {}

    # 2-gram
    for i in  range(1, len(train_data)):
        # concatenate two words
        tmp = train_data[i-1] + ' ' + train_data[i]

        # record the previous word
        if (tmp not in hash2):
            hash2[train_data[i-1] + ' ' + train_data[i]] = {}

    # 3-gram
    for i in  range(2, len(train_data)):
        # the previous two words
        prefix = train_data[i-2] + ' ' + train_data[i-1]
        tmp = train_data[i-2] + ' ' + train_data[i-1] + ' ' + train_data[i]
        if (tmp not in hash2[prefix]):
            hash2[prefix][tmp] = 1
        else:
            hash2[prefix][tmp] += 1

    # pass in the initial value of para k
    instance1 = Lidstone_smoothing(dev_data, k=0.5)

    # smoothing
    instance1.smoothing(hash2)

    # calculate the PPL of test_set
    res1 = instance1.calculate_PPL()
    print(res1)


    # fst train paras lambda_1 and lambda_2 on dev_set

    # 1-gram
    hash1 = {}

    # 2-gram
    hash2 = {}

    # 3-gram
    hash3 = {}
```

```
    # 1-gram
    for word in train_data:
        if (word not in hash1):
            hash1[word] = 1
        else:
            hash1[word] += 1

    # 2-gram
    for i in  range(1, len(train_data)):
        # concatenate two words
        tmp = train_data[i-1] + ' ' + train_data[i]

        # record the previous word
        if (tmp not in hash2):
            hash2[train_data[i-1] + ' ' + train_data[i]] = [train_data[i-1], 1]
        else:
            hash2[train_data[i-1] + ' ' + train_data[i]][1] += 1

    # 3-gram
    for i in  range(2, len(train_data)):
        # concatenate three words
        tmp =  train_data[i-2] + ' ' + train_data[i-1] + ' ' + train_data[i]

        if (tmp not in hash3):
            hash3[tmp] = [train_data[i-2] + ' ' + train_data[i-1], 1]
        else:
            hash3[tmp][1] += 1

    # proportion
    for key in hash3:
        hash3[key][1] /= hash2[hash3[key][0]][1]

    for key in hash2:
        hash2[key][1] /= hash1[hash2[key][0]]

    for key in hash1:
        hash1[key] /=  len(train_data)

    # snd test the model on the test_set with paras lambda_1 and lambda_2

    # pass in the initial values of lambda_1 and lambda_2
    instance2 = Interpolation_smoothing(dev_data, test_data, lambda_1=0.5, lambda_2=0.5)

    # train lambda_1 and lambda_2 on the dev_set
    instance2.train_paras(hash1, hash2, hash3, dev_data, maxiter=1000, stepSize=0.05, tol=1e-5)

    # calculate PPL on the test_set
    res2 = instance2.calculate_PPL(hash1, hash2, hash3)
    print("PPL of test_set: {}". format(res2))


    # test the Good Turing Discounting algorithm

    # record 3 words phase with its prefix
    hash2 = {}
```

```python

    # 2-gram
    for i in  range( len(train_data)-1):
        # concatenate two words
        tmp = train_data[i]

        # record the previous word
        if (tmp not in hash2):
            hash2[train_data[i]] = {}

    # 3-gram
    for i in  range(1, len(train_data)):
        # the previous two words
        prefix = train_data[i-1]
        tmp = train_data[i-1] + ' ' + train_data[i]
        if (tmp not in hash2[prefix]):
            hash2[prefix][tmp] = 1
        else:
            hash2[prefix][tmp] += 1

    # record 3 words phase with its prefix
    test2 = {}

    # 2-gram
    for i in  range( len(test_data)-1):
        # concatenate two words
        tmp = test_data[i]

        # record the previous word
        if (tmp not in test2):
            test2[test_data[i]] = {}

    # 3-gram
    for i in  range(1, len(test_data)):
        # the previous two words
        prefix = test_data[i-1]
        tmp = test_data[i-1] + ' ' + test_data[i]
        if (tmp not in test2[prefix]):
            test2[prefix][tmp] = 1
        else:
            test2[prefix][tmp] += 1

    # pass in the train_set, the test_set and the number of low frequency word
    instance3 = discounting(hash2, test2, count=10)

    # Good Turing Discounting
    instance3.discounting()

    # calculate the PPL of test_set
    res3 = instance3.calculate_PPL()
    print("ppl for discounting: {}". format(res3))
```

Listing 2: **n-gram-model.py**