Task 0

Load the preprocessed data records.

In [1]:

```
import json
import nltk
import numpy as np
from tqdm import tqdm
from itertools import product
import math
import pickle
from collections import defaultdict
from sklearn.metrics import f1_score
```

In [2]:

```
np.random.seed(0)
wiki_data = []
with open("enwiki_20220201.json","r") as f:
    for each_line in f:
        record = json.loads(each_line)
        wiki_data.append(record)
print(len(wiki_data))
```

```
10000
```

Task 1

Data exploring and preprocessing.
Use some simple methods to explore the data and obtain some information about the nature of the data, so as to facilitate the later language processing.
First, let's take a look at the basic structure of the text.

In [4]:

```
print(wiki_data[0].keys())
```

```
dict_keys(['title', 'label', 'text'])
```

You can see that each text data consists of **title**, **label** and **text**. Let's further observe these three elements.

In [10]:

```
print('The title of the first data: \n', wiki_data[0]['title'])
print('The label of the first data: \n', wiki_data[0]['label'])
print('The text of the first data: \n', wiki_data[0]['text'][:199],'\nBecause the text is too l
ong, only a small part is shown here.')
```

```
The title of the first data:
 Citizen_Kane
The label of the first data:
 Film
The text of the first data:
 Citizen Kane is a 1941 American drama film produced by, directed by, and starring
Orson Welles. He also co-wrote the screenplay with Herman J. Mankiewicz. The pictu
re was Welles' first feature film.
Because the text is too long, only a small part is shown here.
```

1) Print out how many documents are in each class

In [126]:

```
def counter(data):
    class_counter = {}
    sent_counter = {}
    token_counter = {}
    tokenizer = nltk.RegexpTokenizer(r'\w+')  # 设定一个分词器
    processed_data = []
    for text_dic in tqdm(data):
        processed_text = []
        text_dic['processed_data'] = []
        # 统计文档的类别，并初始化没有统计过的类别
        label = text_dic['label']
        if label not in class_counter:
            class_counter[label] = 1
            sent_counter[label] = 0
            token_counter[label] = 0
        else:
            class_counter[label] += 1
        for sent in nltk.sent_tokenize(text_dic['text']):
            # 统计每个类别的句子数
            sent_counter[label] += 1
            tokens = nltk.word_tokenize(sent)
            # 统计每个类别的token数
            token_counter[label] += len(tokens)
            # 处理每条句子，使之只保留英文单词和数字，且所有英文单词小写
            processed_text.append(tokenizer.tokenize(sent.lower()))
        processed_data.append(processed_text)
    for key in class_counter.keys():
        sent_counter[key] = round(sent_counter[key]/class_counter[key],2)
        token_counter[key] = round(token_counter[key]/class_counter[key],2)
    return class_counter, sent_counter, token_counter, processed_data
```

In [26]:

```
class_counter, sent_counter, token_counter, processed_data = counter(wiki_data)
```

100%|██| 1000
0/10000 [14:45<00:00, 11.30it/s]

In [27]:

```
# 打印出每个类有多少个文档
class_counter
```

Out[27]:

```
{'Film': 3048,
 'Book': 975,
 'Politician': 3824,
 'Writer': 837,
 'Food': 137,
 'Actor': 80,
 'Animal': 93,
 'Software': 266,
 'Artist': 520,
 'Disease': 220}
```

In [28]:

```
# 打印出每个类的平均句子数
sent_counter
```

Out[28]:

```
{'Film': 178.62,
 'Book': 205.26,
 'Politician': 225.29,
 'Writer': 217.89,
 'Food': 155.43,
 'Actor': 70.95,
 'Animal': 66.81,
 'Software': 202.62,
 'Artist': 185.04,
 'Disease': 349.6}
```

In [29]:

```
# 打印出每个类的平均token数
token_counter
```

Out[29]:

```
{'Film': 4440.56,
 'Book': 5296.75,
 'Politician': 5708.72,
 'Writer': 5806.98,
 'Food': 3477.58,
 'Actor': 1719.95,
 'Animal': 1432.14,
 'Software': 4813.02,
 'Artist': 4802.04,
 'Disease': 8012.64}
```

In [31]:

```
# 删除变量，节省内存
del class_counter
del sent_counter
del token_counter
```

In [30]:

```
# 打印出处理后的文本，检查一下
processed_data[0][0][:10]
```

Out[30]:

```
['citizen',
 'kane',
 'is',
 'a',
 '1941',
 'american',
 'drama',
 'film',
 'produced',
 'by']
```

文本中确实只剩英文单词和数字，且所有英文字母小写。

In [38]:

```
# 删除变量，节省内存
del processed_data
```

In [3]:

```
# 存储处理后的数据，以后就不用再处理了,因为数据量太大了存不下，分为两个文件保存
# path = 'processed_data1'
# with open(path, 'wb') as f:
#     pickle.dump(processed_data[:5000], f)
# f.close()
# path = 'processed_data2'
# with open(path, 'wb') as f:
#     pickle.dump(processed_data[5000:], f)
# f.close()
# 加载数据
path = 'processed_data1'
f1 = open(path, 'rb')
data = pickle.load(f1)
data[0][0][:10]
path = 'processed_data2'
f1 = open(path, 'rb')
data.extend(pickle.load(f1))
```

Task 2

首先要做的是对数据集进行划分，主要分为两部分：

1. 打乱数据集，避免其出现聚集现象导致模型效果降低
2. 划分数据集，使用9:1的比例划分为训练集和测试集

In [4]:

```
def split_data(data, seed, train_porp):
    '''
    输入待划分数据，随机种子以及训练集比例
    根据题目，我们暂且将训练集的比例固定为10%(最后的)
    '''
    np.random.seed(seed)
    # 打乱数据集
    np.random.shuffle(data)
    # 划分数据集
    num_train = math.floor(len(data)*train_porp)
    num_test = math.floor(len(data)*0.9)
    return data[:num_train], data[num_test:]
```

In [5]:

```
# test
train_data, test_data = split_data(data,1,0.9)
print(len(train_data),len(test_data))
```

```
9000 1000
```

首先要对数据集进行进一步的处理，主要包括两部分：

1. 对每一句话的首尾分别加上;表示语句的开始和结束，另外一个作用是保证一个句子里的每一个单词在分词后的词组中出现的次数是一致的。要注意如果ngram的n大于2的话，要加上n-1个首尾标识符。
2. 对所有的token进行初步的统计，如果出现的频数过小，比如一次，可以直接将其设置为,这样做最大的好处是对数据进行了降维，另外一个也可以防止过拟合，如果只出现过一次，我们可以认为它是一个非常稀少的词，不考虑再次出现的情况，有利于增强模型的泛化能力。

In [6]:

```
def add_s(data, n):
    processed_text = []
    start = ['<s>']*max(1,n-1)
    end = '</s>'  # 这里只拼一个是因为后面要把所有句子合并在一起再划分窗口，如果end的数量超过一个，会出现好几种切割出来的元组只有start和end的情况
    for text in tqdm(data):
        processed_text.extend(start)
        processed_text.extend(text)
        processed_text.append(end)
    return processed_text
```

In [7]:

```
def drop_unique(text):
    vocab = nltk.FreqDist(text)
    return [token if vocab[token] > 1 else '<UNK>' for token in text]
```

In [8]:

```
def predata(text_data, n):
    '''
    Input:
    text_data:一个列表，里面每个元素为预处理的分词之后句子列表
    n:ngram的词元数
    '''
    processed_text = add_s(text_data, n)
    return drop_unique(processed_text)
```

In [9]:

```
# 先将列表展开一层
flatten_data = [y for x in train_data for y in x]
pretrain_3 = predata(flatten_data,3)
```

100%|██████████| 1840809/1840809 [00:02<00:00, 843932.72it/s]

In [10]:

```
flatten_data = [y for x in test_data for y in x]
pretest_3 = predata(flatten_data,3)
```

100%|██████████| 207840/207840 [00:00<00:00, 577857.03it/s]

In [12]:

```
# 保存已经预处理好的数据
# path = 'trigram_train1'
# with open(path, 'wb') as f:
#     pickle.dump(pretrain_3, f)
# f.close()
path = 'trigram_test'
with open(path, 'wb') as f:
    pickle.dump(pretest_3, f)
f.close()
```

Language Model

手动实现laplace平滑和KN平滑的Ngram模型。

In [99]:

```
class LM():
    '''
    定义语言模型这个类，可以实现ngram及laplace、kneser-Ney平滑方法可选
    '''
    def __init__(self, train_data, n, laplace=False, KN = False):
        '''
        Input:
        train_data：处理过后的训练集，为一个列表，每个元素为一个token
        n：ngram的窗口大小
        laplace：取值为False，代表不使用这个平滑；否则其取值为k,代表实现加k平滑
        KN：取值为False，代表不使用这个平滑；否则其取值为d,代表折扣系数
        '''
        self.token = train_data
        self.n = n
        self.vocab = nltk.FreqDist(train_data)
        self.vocabsize = len(self.vocab)
        # 注意这里的话，如果是laplace模型，那么就只生成了ngram的数据集，但是如果是KNModel，可以生成所有小于等于n的数据集
        self.masks = list(reversed(list(product((0,1), repeat=n))))  # 后面为了匹配测试集与训练集的元组需要的，取反是为了实现最大匹配原则
        self.laplace = laplace
        self.KN = KN
        self.model = self.laplaceModel() if laplace else self.KNModel()

    def LpSmooth(self):
        '''
        实现n>1的laplace平滑
        '''
        # 统计ngram的词典
        ngram = nltk.ngrams(self.token, self.n)
        nVocab = nltk.FreqDist(ngram)
        # 统计ngram前面的N-1的词典
        pre_gram = nltk.ngrams(self.token, self.n-1)
        pre_Vocab = nltk.FreqDist(pre_gram)
        # 计算laplace平滑后的概率
        def Lpprob(gram, count):
            pre = gram[:-1]
            pre_count = pre_Vocab[pre]
            return (count + self.laplace)/(pre_count+self.laplace*self.vocabsize)
        Lpdict = {gram:Lpprob(gram, count) for gram, count in nVocab.items()}
        return Lpdict

    def laplaceModel(self):
        '''
        laplace平滑模型
        '''
        # 先考虑unigram的特殊情况
        if self.n == 1:
            den = len(self.token) # 分母
            Lpdict = {(gram,): count/den for gram, count in self.vocab.items()} # 为了保证输出时的key是tuple
            return Lpdict
        return self.LpSmooth()

    def KNModel(self):
        '''
        KN平滑模型
        '''
        kgram_counts = [nltk.FreqDist(nltk.ngrams(self.token, self.n))]
```

```
        for i in range(1, self.n-1):
            kgram_counts.append(nltk.FreqDist(nltk.ngrams(self.token, self.n-i)))
        den = len(self.token)
        kgram_counts.append({(gram,): count/den for gram, count in self.vocab.items()})
        probs = self.cal_probs(kgram_counts)
        return probs  # 这里返回的概率是所有小于等于n的KN平滑后概率，因此n=1,2,3只用做一次

    def cal_probs(self, orders):
        '''
        计算所有等级（n）的平滑概率
        '''
        backoffs = []
        for order in orders[:-1]:
            backoff = self.cal_order_backoff_probs(order)
            backoffs.append(backoff)
#         orders[-1] = self.cal_unigram_probs(order[-1])  # 当n等于1时要单独计算
        backoffs.append(defaultdict(int))  # n等于1时的backoff为0
        self.interpolate(orders, backoffs)  # 将上面的概率插值求出来
        return orders

#     def cal_unigram_probs(self, unigrams):
#         den = sum(value for value in unigrams.values())  # 计算分母
#         unigrams = {key: math.log(value/den) for key, value in unigrams.items()}
#         return unigrams

    def cal_order_backoff_probs(self, order):
        '''
        分别计算KN平滑的前后两部分
        '''
        prefix_sums = defaultdict(int)
        backoffs = defaultdict(int)
        for key in order.keys():
            prefix = key[:-1]
            count = order[key]
            prefix_sums[prefix] += count
            order[key] -= self.KN  # 减去折扣值
            backoffs[prefix] += self.KN
        for key in order.keys():
            prefix = key[:-1]
            order[key] = order[key]/prefix_sums[prefix]  #计算discounted ngram
        for prefix in backoffs.keys():
            backoffs[prefix] = backoffs[prefix]/prefix_sums[prefix]
        return backoffs  # 其实是返回order和backoff两部分，但由于order是直接输入的地址，因此无需传出

    def interpolate(self, orders, backoffs):
        '''
        计算所有KN平滑过后的概率
        '''
        for pre_order, order, backoff in zip(reversed(orders),reversed(orders[:-1]),
                                             reversed(backoffs[:-1])):
            for kgram in order.keys():
                prefix, suffix = kgram[:-1], kgram[1:]
                order[kgram] += pre_order[suffix]*backoff[prefix] # 动态更新ngram的KN平滑概率

    def perplexity(self, test_data, n_of_KN = None):
        '''
        用已经生成的模型计算给定测试数据的困惑度，要求test_data也是处理过的
        '''
        test_gram = nltk.ngrams(test_data, self.n) if self.laplace else nltk.ngrams(test_data, n_of_KN)
```

```
        N = len(test_data)
        # 找到测试集对应训练集中的元组，因为可能出现元组只有一部分出现在训练集中
        def matchvocab(gram):
            masks = self.masks if self.laplace else list(reversed(list(product((0,1), repeat=n_
of_KN))))
            gram = (gram, ) if type(gram) is str else gram # 为了匹配之前那种unigram的key也是元
组的情况
            for mask in masks:
                possible_in  = tuple((token if sign == 1 else '<UNK>' for token, sign in zip(g
ram, mask)))
                if n_of_KN:
                    if possible_in in self.model[self.n-n_of_KN]:
                        return possible_in
                else:
                    if possible_in in self.model:
                        return possible_in
        transform_gram = (matchvocab(gram) for gram in test_gram)
        if n_of_KN:
            probabilities = [self.model[self.n-n_of_KN][gram] for gram in transform_gram]
        else:
            probabilities = [self.model[gram] for gram in transform_gram]
        return math.exp((-1/N)*sum(map(math.log, probabilities)))

    def choose_next(self, prev, drop, i=0, n_of_KN=None):
        '''
        给定前面的词，生成下一个词，由于我的模型采取的是取概率最大的词，因此要
        输入一个drop list，以免生成的词前面出现过，否则就会一直重复。
        '''
        drop = ['<UNK>']+drop
        # 先考虑KN平滑模型的情况
        if n_of_KN:
            candidate = ((ngram[-1], prob) for ngram, prob in self.model[self.n-n_of_KN].items
() if ngram[:-1]==prev)
            candidate = filter(lambda candidate:candidate[0] not in drop, candidate)
            sorted_vocab = sorted(candidate, key=lambda cand:cand[1], reverse=True)
#             print(len(sorted_vocab))
#             print(sorted_vocab)
            if len(sorted_vocab):
                try:
                    return sorted_vocab[i] if sorted_vocab[i][0] !='</s>' else sorted_vocab[i+
1]
                except:
                    return ('</s>',1)
            else:
                return ('</s>',1)
        # laplace模型
        #依旧是先处理n=1的情况
        if self.n == 1:
            sorted_vocab = sorted(self.model.items(), key= lambda token:token[1], reverse=True)
            filtered_vocab = list(filter(lambda candidate:candidate[0][0] not in drop, sorted_
vocab))
            return (filtered_vocab[0][0][0],filtered_vocab[0][1]) if len(filtered_vocab) else
('</s>',1)
        # 处理n>1的情况
        candidate = ((ngram[-1], prob) for ngram, prob in self.model.items() if ngram[:-1]==pre
v)
        candidate = filter(lambda candidate:candidate[0] not in drop, candidate)
        sorted_vocab = sorted(candidate, key=lambda cand:cand[1], reverse=True)
        if len(sorted_vocab):
            return sorted_vocab[i] if sorted_vocab[i][0] !='</s>' else sorted_vocab[i+1]
        else:
```

```
            return ('</s>',1)

    def generate_sentence(self, num_sent, sent_start, max_len = 12, n_of_KN = None):
        '''
        Input:
        num_sent：要生成的句子数量
        sent_start：给定的句子开头（一个单词）
        max_len：要生成的句子的最大长度
        '''
        what_n = self.n if self.laplace else n_of_KN
        for i in range(num_sent):
            sent, prob = ['<s>']*max(1,what_n-1)+[sent_start[i]], 1
            while sent[-1] != '</s>':
                prev = tuple(sent[-(what_n-1):]) if what_n != 1 else ()
                if n_of_KN: # KN模型
                    next_token, new_prob = self.choose_next(prev, sent, i, n_of_KN)
                else:  # laplace模型
                    next_token, new_prob = self.choose_next(prev, sent, i)
                sent.append(next_token)
                prob *= new_prob
                if len(sent) >= max_len:
                    sent.append('</s>')
            print('The',i,'th sentence:\n',' '.join(sent),'\nProbability is',prob)
```

Laplace Smoothing

Unigram

先处理Unigram模型，训练模型，并在测试集上计算测试集困惑度。
给定五个单词（也可以给定多个单词或者其他数量的开头组别）开头，生成五条句子（**句子的最大长度默认设置为12**，后面保持一致）。

In [105]:

```
pretrain_3.extend(['<UNK>','<UNK>','<UNK>'])
laplace_1 = LM(pretrain_3, 1, laplace=1)
print('The perplexity of Unigram using Laplace Smoothing is',laplace_1.perplexity(pretest_3))
```

```
The perplexity of Unigram using Laplace Smoothing is 1183.6832139398616
```

In [107]:

```
start = ['he','like','the','rainy','day']
print('Genrate sentence given',start)
laplace_1.generate_sentence(5,start)
```

```
Genrate sentence given ['he', 'like', 'the', 'rainy', 'day']
The 0 th sentence:
 <s> he the </s>
Probability is 0.0023393517362952317
The 1 th sentence:
 <s> like the </s>
Probability is 0.0023393517362952317
The 2 th sentence:
 <s> the </s>
Probability is 0.03994934901695986
The 3 th sentence:
 <s> rainy the </s>
Probability is 0.0023393517362952317
The 4 th sentence:
 <s> day the </s>
Probability is 0.0023393517362952317
```

可以看到，上面的句子根本不成句子，还有一个重要的特点，每条句子都是由单词*the*结尾。这很好理解,Unigram不考虑词和词之间的关联性，统计的词频字典的排列完全是依赖于单词的出现概率。而我的模型的生成句子的机制为选择**句子之前没有出现过的**，**概率最高**的词作为下一个单词。众所周知，如果不对数据进行一些删除停用词之类的处理，那么出现频率最高的词大多都是一些无异议的助词、副词等，单词*the*就符合这样的条件。因此，我们可以推测，*the*这个单词的出现频率相当之高，END标识符的频率则仅次*the*。
接下来，让我们查看模型的词典验证猜想：

In [109]:

```
# 查看一下模型的词典排序前五的单词
sorted(laplace_1.model.items(), key= lambda token:token[1], reverse=True)[:5]
```

Out[109]:

```
[(('<s>',), 0.07989869803391972),
 (('the',), 0.05855794384289209),
 (('</s>',), 0.03994934901695986),
 (('of',), 0.028814567777359162),
 (('and',), 0.025821999133618743)]
```

可以看到，词频排序第一的是start标识符，由于每个句子开头都会有该标识符，因此生成句子时不会重复考虑该词。而*the*果然排在仅次于start标识符的第二位，且end标识符也在*the*后面，上面的猜想得到验证。

In [118]:

```
# 删除模型，节省内存
del laplace_1
```

Bigrams

In [23]:

```
laplace_2 = LM(pretrain_3, 2, laplace = 1)
print('The perplexity of Bigrams using Laplace Smoothing is',laplace_2.perplexity(pretest_3))
```

```
The perplexity of Bigrams using Laplace Smoothing is 1198.6933669904665
```

接下来生成句子，收到之前Unigram的启发，我将模型的生成句子功能稍作修改，每次不是选择概率最大的词组，而是给定了一个可选参数i（选择概率第i大的词组）,可以调节这个i使得多句话之间尽量避开出现重复的可能性，也增加了生成句子结果的丰富性，当然，这个i会比较小，能够确保生成的句子依旧是联合概率接近最大的。

In [46]:

```
# 生成句子
start = ['wrote','picture','feature','he','a']
print('Genrate sentence given',start)
laplace_2.generate_sentence(5,start)
```

```
Genrate sentence given ['wrote', 'picture', 'feature', 'he', 'a']
The 0 th sentence:
 <s> wrote that the film s first time yūko and a new </s>
Probability is 4.0851438666176433e-23
The 1 th sentence:
 <s> picture of his father was the first film s work on </s>
Probability is 1.450105434062245e-19
The 2 th sentence:
 <s> feature of a few years in which is an average grade </s>
Probability is 1.0688741125943289e-21
The 3 th sentence:
 <s> he also been described it s work in which had not </s>
Probability is 2.979635861755352e-21
The 4 th sentence:
 <s> a result was an attempt on november 2017 in its second </s>
Probability is 1.4837724352278043e-25
```

可以看到句子生成正常了许多，但其内在还是比较无逻辑。
和Unigram对比，我们也来看看排序比较前的unigram组合：

In [48]:

```
# 查看一下模型的词典排序前十的单词
sorted(laplace_2.model.items(), key= lambda token:token[1], reverse=True)[:10]
```

Out[48]:

```
[(('</s>', '<s>'), 0.9047834143512261),
 (('<s>', '<s>'), 0.4750061865555504),
 (('of', 'the'), 0.24517574550985746),
 (('at', 'the'), 0.2058128664004922),
 (('in', 'the'), 0.1950417376517553),
 (('for', 'the'), 0.1561703800318033),
 (('on', 'the'), 0.1487876333528144),
 (('from', 'the'), 0.1375576973545409),
 (('as', 'a'), 0.1345384198990141),
 (('with', 'the'), 0.12908515638403562)]
```

可以看到出现频率比较高的还是一些无意义的副词，不过好在我们的生成句子不会生成重复的词，也不会生成start和end标识符，因此扩充了生成句子的多样性。

Trigram

In [54]:

```
laplace_3 = LM(pretrain_3, 3, laplace = 1)
print('The perplexity of Trigrams using Laplace Smoothing is',laplace_3.perplexity(pretest_3))
```

```
The perplexity of Trigrams using Laplace Smoothing is 5064.635667231597
```

In [57]:

```
# 生成句子
start = ['feature','wrote']
print('Genrate sentence given',start)
laplace_3.generate_sentence(2,start)
start = ['picture','he','a']
print('Genrate sentence given',start)
laplace_3.generate_sentence(3,start)
```

```
Genrate sentence given ['feature', 'wrote']
The 0 th sentence:
 <s> <s> feature animation and the united states on november 6 2012 </s>
Probability is 1.9973856131544054e-28
The 1 th sentence:
 <s> <s> wrote in his book on a new constitution was adopted </s>
Probability is 8.587839036450261e-32
Genrate sentence given ['picture', 'he', 'a']
The 0 th sentence:
 <s> <s> picture of the film s release date was pushed back </s>
Probability is 1.1350961028740091e-26
The 1 th sentence:
 <s> <s> he also said the book s title and was released </s>
Probability is 3.2580665152331657e-28
The 2 th sentence:
 <s> <s> a new government and that it would not seek to </s>
Probability is 2.5961921770238573e-27
```

可以看到生成的句子变得通顺许多！
照例，查看一下模型词典概率较高的词组：

In [58]:

```
# 查看一下模型的词典排序前十的单词
sorted(laplace_3.model.items(), key= lambda token:token[1], reverse=True)[:10]
```

Out[58]:

```
[(('</s>', '<s>', '<s>'), 0.9047838590651693),
 (('<s>', '<s>', 'the'), 0.11044467272539604),
 (('the', 'united', 'states'), 0.09821698398307646),
 (('one', 'of', 'the'), 0.08719148356118002),
 (('<s>', '<s>', 'in'), 0.08073658289629546),
 (('as', 'well', 'as'), 0.06930922358545853),
 (('film', '</s>', '<s>'), 0.06653110197709214),
 (('<UNK>', '</s>', '<s>'), 0.06356196433491727),
 (('<s>', 'he', 'was'), 0.06354765257219022),
 (('<s>', 'in', 'the'), 0.06316536352487981)]
```

可以看到会出现比较多的start标识符和end标识符，这是正常的，因为输入的句子和句子连接，导致end标识符后面几乎一定是start标识符。
最后，来总结一下困惑度：

- The perplexity of Unigram using Laplace Smoothing is 1183.6832139398616
- The perplexity of Bigrams using Laplace Smoothing is 1198.6933669904665
- The perplexity of Trigrams using Laplace Smoothing is 5064.635667231597

可以看到困惑度在trigram的模型下反而有一个异常的上升，我对此查阅资料，提出了以下几个猜想：

1. 困惑度在样本量比较小的情况下波动会比较大，并不是严格下降的。我查阅到其他人做困惑度和LDA主题数关系的时候发现，困惑度的波动值非常大，但总体的趋势是下降的。
2. 困惑度越小代表这个句子的概率越大。因此也有可能是因为使用的数据集还不够大，并未统计出较为完善的词库，导致训练集和测试集之间的句子风格差异还是比较大，因此预测比较差。
3. 在数据处理时，我将一部分只出现了一次处理为了UNK，这导致一些原本不一样的词组被同类化，因此其条件概率就降低了，句子的联合概率降低，导致困惑度上升。 无论是哪一种原因导致，都需后续进行优化处理。

KN Smoothing

与laplace平滑不同的是，由于KN平滑在计算中是迭代计算的。为了优化我的模型的计算效率，我在计算时保留所有用于迭代的概率。也就是说，只要计算了trigram的KN平滑，就能同时得到unigram和bigram的结果

In [109]:

```
# 训练模型，直接训trigram
KN_3 = LM(pretrain_3,3,KN=0.1)
```

In [110]:

```
# 计算困惑度
print('The perplexity of Unigrams using KN Smoothing is',KN_3.perplexity(pretest_3,1))
print('The perplexity of Bigrams using KN Smoothing is',KN_3.perplexity(pretest_3,2))
print('The perplexity of Trigrams using KN Smoothing is',KN_3.perplexity(pretest_3,3))
```

```
The perplexity of Unigrams using KN Smoothing is 1189.0227997987856
The perplexity of Bigrams using KN Smoothing is 141.54655339809918
The perplexity of Trigrams using KN Smoothing is 31.628652819181333
```

可以看到这个困惑度相当标准，从unigram到bigram再到trigram呈递减的形式，这是因为n越大，KN平滑的模型效果就越好，因此在真实数据（或者说正确数据）上的概率越大，因此困惑度也越小。
另一方面，会发现KN平滑整体的困惑度比laplace平滑小的多，这说明KN平滑是比laplace平滑更加好的方法。

In [114]:

```
# 每个模型生成5个句子
start = ['feature','wrote']
print('Genrate sentence given',start)
print('Unigram model:')
KN_3.generate_sentence(2,start,n_of_KN=1)
print('Bigram model:')
KN_3.generate_sentence(2,start,n_of_KN=2)
print('Trigram model:')
KN_3.generate_sentence(2,start,n_of_KN=3)
start = ['picture','he','a']
print('Genrate sentence given',start)
print('Unigram model:')
KN_3.generate_sentence(3,start,n_of_KN=1)
print('Bigram model:')
KN_3.generate_sentence(3,start,n_of_KN=2)
print('Trigram model:')
KN_3.generate_sentence(3,start,n_of_KN=3)
```

```
Genrate sentence given ['feature', 'wrote']
Unigram model:
The 0 th sentence:
 <s> feature the of and in to a was s that he </s>
Probability is 3.08060513522822e-18
The 1 th sentence:
 <s> wrote of and in to a was s that he as </s>
Probability is 4.220595325284688e-19
Bigram model:
The 0 th sentence:
 <s> feature film s first time in the united states and a </s>
Probability is 3.230001888633974e-12
The 1 th sentence:
 <s> wrote the first film was not be used in his father </s>
Probability is 7.534654687096079e-15
Trigram model:
The 0 th sentence:
 <s> <s> feature animation and the united states on november 6 2012 </s>
Probability is 1.9005014884189034e-09
The 1 th sentence:
 <s> <s> wrote in his book on a new constitution was adopted </s>
Probability is 1.0987431096048057e-12
Genrate sentence given ['picture', 'he', 'a']
Unigram model:
The 0 th sentence:
 <s> picture the of and in to a was s that he </s>
Probability is 3.08060513522822e-18
The 1 th sentence:
 <s> he of and in to a was s that as for </s>
Probability is 4.142636971375308e-19
The 2 th sentence:
 <s> a of and in to was s that he as for </s>
Probability is 1.7699682592020075e-19
Bigram model:
The 0 th sentence:
 <s> picture of the film s first time in a new york </s>
Probability is 1.099135743562444e-12
The 1 th sentence:
 <s> he had a member and his father was not be used </s>
Probability is 4.837977059539752e-16
The 2 th sentence:
 <s> a few years later that it s work of her husband </s>
Probability is 5.150606136851034e-16
Trigram model:
The 0 th sentence:
 <s> <s> picture of the film s release date was pushed back </s>
Probability is 3.5023327167272474e-08
The 1 th sentence:
 <s> <s> he also served on a new constitution was adopted as </s>
Probability is 5.135559129925151e-12
The 2 th sentence:
 <s> <s> a new government and that it would not seek the </s>
Probability is 8.401232804133572e-13
```

可以看到生成的句子依旧符合之前的规律,gram的滑动窗口越大，生成的句子越有逻辑。其中Unigram不再像之前那样只生成一个或两个单词，是因为前面对模型的生成句子功能进行了一些修改，使其如果选到end标识符则跳过选择下一个频率最大的候选词。因此它的输出长度变长了一些。
照例，我们看一下模型训练出来的词典概率较大的词组：

In [113]:

```
# 查看一下模型的词典排序前十的单词
print('Unigram Model:')
print(sorted(KN_3.model[2].items(), key= lambda token:token[1], reverse=True)[:10])
print('Bigram Model:')
print(sorted(KN_3.model[1].items(), key= lambda token:token[1], reverse=True)[:10])
print('Trigram Model:')
print(sorted(KN_3.model[0].items(), key= lambda token:token[1], reverse=True)[:10])
```

```
Unigram Model:
[(('<s>',), 0.07989219120579981), (('the',), 0.05856060737906329), (('</s>',), 0.0
39946095602899905), (('of',), 0.02882285065211335), (('and',), 0.02582449939701394
3), (('in',), 0.023258439689003544), (('to',), 0.022127029785779476), (('a',), 0.0
18937303642181602), (('was',), 0.010778774491548017), (('s',), 0.00966088387118306
5)]
Bigram Model:
[(('</s>', '<s>'), 0.9999999500160902), (('doesn', 't'), 0.9999614251294366), (('w
asn', 't'), 0.9999263467283496), (('isn', 't'), 0.9999171934634385), (('couldn',
't'), 0.9998989408598283), (('wouldn', 't'), 0.9998744378270983), (('didn', 't'),
0.9996773555355648), (('hadn', 't'), 0.9996723033126893), (('weren', 't'), 0.99966
11949504075), (('shouldn', 't'), 0.9996600425522797)]
Trigram Model:
[(('film', '</s>', '<s>'), 0.9999999999996378), (('<UNK>', '</s>', '<s>'), 0.99999
99999999996199), (('him', '</s>', '<s>'), 0.999999999999544), (('it', '</s>', '<s>'),
0.9999999999995405), (('life', '</s>', '<s>'), 0.9999999999994994), (('time', '</s
>', '<s>'), 0.9999999999993631), (('election', '</s>', '<s>'), 0.999999999999229
2), (('years', '</s>', '<s>'), 0.9999999999992268), (('them', '</s>', '<s>'), 0.99
99999999992054), (('war', '</s>', '<s>'), 0.9999999999992006)]
```

Unigram排名较前的都是一些副词，是合理的。
可以看到Bigram排名较前的多是一些否定副词的前后缀，这主要是由于我们在处理数据时，要求去除所有除字母和数字的特殊符号，导致否定形式的前后被拆分为两部分，这其实是不太准确的，后续改进可以从数据处理角度再改进。
Trigram就更奇特了，可以看到前面的几个组合概率大多接近1，且最后两个词均为（end标识符，start标识符）。仔细一想，这也很容易理解，因为trigram要三个词组成，而我们输入的数据都是句子接句子，那么end标识符的下一个词一定是start标识符，无论第一个词是什么。不过这个现象并不会影响我们的句子生成和困惑度计算，反而，要有这个性质，我们的困惑度计算才是准确的（因为不希望句子和句子之间的衔接影响概率计算）。

In [115]:

```
# 第二题到此为止，删除训练好的模型，节省内存
del laplace_2
del laplace_3
del KN_3
```

Task 3

构建朴素贝叶斯分类器，对文本进行分类

In [189]:

```
# 创建NB模型
class NaiveBayes():
    def __init__(self, class_counter, train_data, laplace = 1):
        '''
        Input:
        class_counter:每个类别的文档数
        train_data:训练的文档数据 一个大字典，key为类别，value为列表，里面是token
        train_labels:训练数据对应的类别
        laplace:laplace平滑的参数
        '''
        self.class_counter = class_counter
        self.vocab = set([y for x in train_data.values() for y in x ])
        self.vocabsize = len(self.vocab)+1 # 记得加上UNK
        self.token_class_dict = defaultdict(dict)
        self.num_token_class = defaultdict(int)
        self.data = train_data
        self.laplace = laplace

    def create_model(self):
        '''
        训练NB模型
        '''
        for label, token in self.data.items():
            self.token_class_dict[label] = nltk.FreqDist(token)
            self.num_token_class[label] = len(token)

    def predict_text(self, text):
        '''
        给定一个文本text，预测其类别
        Input:
        text:处理过后的text，为一个大列表,里面是token
        '''
        predict_prob = {}
        for label, num_text in self.class_counter.items():
            predict_prob[label] = num_text  # 这里不除以文档数是因为对比较结果无影响，且也可以预
防概率过小不精确
            for token in text:
                temp = self.token_class_dict[label][token] if token in self.token_class_dict[la
bel] else 0
                predict_prob[label] *= (temp + self.laplace)/(self.num_token_class[label]+self.l
aplace*self.vocabsize)
        pred_label = max(predict_prob.items(), key=lambda x:x[1])[0]
        return pred_label

    def predict_test(self, testdata, tokens):
        '''
        在测试集上预测
        Input:
        testdata:处理后的测试数据，为一个大列表，里面嵌套小列表，每个小列表为一个text
        为了防止有些文本的过长，对每个文档我只取最多10000个tokens
        '''
        return [self.predict_text(text[:tokens]) for text in testdata]

    def report(self, testdata, test_labels, tokens = 10000):
        '''
        计算并报告模型在测试集上的准确率和Micro-F1 分数和 Macro-F1 分数
        '''
        pred_labels = self.predict_test(testdata, tokens)
        ac = sum([1 if pred_labels[i]==test_labels[i] else 0 for i in range(len(test_labels
```

```
))])/len(test_labels)
        print('The accuracy of the test data is',ac)
        print('The Micro-F1 of the test data is',f1_score(test_labels,pred_labels,average='micr
o'))
        print('The Macro-F1 of the test data is',f1_score(test_labels,pred_labels,average='macr
o'))
```

In [160]:

```
path = 'processed_data1'
f1 = open(path, 'rb')
data = pickle.load(f1)
path = 'processed_data2'
f1 = open(path, 'rb')
data.extend(pickle.load(f1))
```

In [130]:

```
# 因为需要使用类别数据和原始对应的标签，因此这里再统计一次数据
class_counter = defaultdict(int)
data_labels = []
for record in tqdm(wiki_data):
    class_counter[record['label']] += 1
    data_labels.append(record['label'])
```

100%|██████████| 10000/10000 [00:00<00:00, 490952.34it/s]

In [162]:

```
# 划分带标签的数据集
data_with_label = [[data[i], data_labels[i]] for i in range(len(data))]
traindata_with_label, testdata_with_label = split_data(data_with_label,1,0.9)
```

In [213]:

```
# 将数据处理成需要的输入格式
train_dict_30 = defaultdict(list)
class_counter_30 = defaultdict(int)
train_dict_50 = defaultdict(list)
class_counter_50 = defaultdict(int)
train_dict_70 = defaultdict(list)
class_counter_70 = defaultdict(int)
train_dict_90 = defaultdict(list)
class_counter_90 = defaultdict(int)
for i in tqdm(range(9000)):
    if i < 9000*0.3:
        text, text_label = traindata_with_label[i]
        flatten_text = [y for x in text for y in x]
        train_dict_30[text_label].extend(flatten_text)
        train_dict_50[text_label].extend(flatten_text)
        train_dict_70[text_label].extend(flatten_text)
        train_dict_90[text_label].extend(flatten_text)
        class_counter_30[text_label] += 1
        class_counter_50[text_label] += 1
        class_counter_70[text_label] += 1
        class_counter_90[text_label] += 1
    elif i < 9000*0.5:
        text, text_label = traindata_with_label[i]
        flatten_text = [y for x in text for y in x]
        train_dict_50[text_label].extend(flatten_text)
        train_dict_70[text_label].extend(flatten_text)
        train_dict_90[text_label].extend(flatten_text)
        class_counter_50[text_label] += 1
        class_counter_70[text_label] += 1
        class_counter_90[text_label] += 1
    elif i < 9000*0.7:
        text, text_label = traindata_with_label[i]
        flatten_text = [y for x in text for y in x]
        train_dict_70[text_label].extend(flatten_text)
        train_dict_90[text_label].extend(flatten_text)
        class_counter_70[text_label] += 1
        class_counter_90[text_label] += 1
    else:
        text, text_label = traindata_with_label[i]
        flatten_text = [y for x in text for y in x]
        train_dict_90[text_label].extend(flatten_text)
        class_counter_90[text_label] += 1
# 处理测试数据
test_texts = []
test_labels = []
for i in tqdm(testdata_with_label):
    test_texts.append([y for x in i[0] for y in x])
    test_labels.append(i[1])
```

In [211]:

```
# 分别构建不同训练集的模型，并进行训练
NB_30 = NaiveBayes(class_counter_30, train_dict_30)
NB_30.create_model()
NB_50 = NaiveBayes(class_counter, train_dict_50)
NB_50.create_model()
NB_70 = NaiveBayes(class_counter, train_dict_70)
NB_70.create_model()
NB_90 = NaiveBayes(class_counter, train_dict_90)
NB_90.create_model()
```

In [212]:

```
# 测试模型
print('For model using 30% train data:')
NB_30.report(test_texts,test_labels,80)
print('For model using 50% train data:')
NB_50.report(test_texts,test_labels,80)
print('For model using 70% train data:')
NB_70.report(test_texts,test_labels,80)
print('For model using 90% train data:')
NB_90.report(test_texts,test_labels,80)
```

```
For model using 30% train data:
The accuracy of the test data is 0.887
The Micro-F1 of the test data is 0.887
The Macro-F1 of the test data is 0.6805092966288969
For model using 50% train data:
The accuracy of the test data is 0.885
The Micro-F1 of the test data is 0.885
The Macro-F1 of the test data is 0.6795896767380041
For model using 70% train data:
The accuracy of the test data is 0.893
The Micro-F1 of the test data is 0.893
The Macro-F1 of the test data is 0.6874751559150226
For model using 90% train data:
The accuracy of the test data is 0.895
The Micro-F1 of the test data is 0.895
The Macro-F1 of the test data is 0.7687731308554194
```

观察模型的测试结果，可以看到基本上是符合数据集越大，Micro-F1和Macro-F1的数值趋势是越来越大的。这也是符合逻辑的，因为数据集越大，我的模型就越符合真实的概率分布模型，因此对测试数据的分类也越准确。
第二个观察到的点是，Macro-F1总是比Micro-F1小一些。这是由于从定义上来说，Macro-F1平等地看待各个类别，它的值会受到稀有类别的影响，也就说它更容易被一些难以预测的类别拉低正确率。
第三个点是，最后的预测中，我采取的测试集输入数据是只取了每个文本的前80个token。这个是试出来的。在此之前，我所有的模型的准确率都为0.22，这个数字和完全无训练的分类模型一模一样！这是由于NB模型是一个概率相乘的模型，词数越多，相乘的小于1的数字就越多，最终的句子概率呈现**指数下降**的趋势，而又由于计算机的存储精度并不高，最终导致所有句子的预测概率存储都变为了0，分类也是完全随机的了。因此，适当地减少一些输入测试数据是合理的。经过测试（**在验证集上**），当token取0-85时，模型的效果随着数据量的增加而增加，当token取85-100时，模型的效果随着数据量的增加而锐减，大于100后，效果几近于无。因此，我最终选取的token数为80.
ADD NOTE:
后面我又想到，解决这个问题其实可以使用log变换，不仅可以放大概率的数值，乘法变成加法也不会再出现概率消减的情况。后来尝试了这种方法后，发现最终模型效果并没有进一步提升，因此这里不再改动。