

BLEU分数简介

下面是paddlenlp中的评价指标，我们这里仅仅实现BLEU

Metric	简介	API
Perplexity	困惑度，常用来衡量语言模型优劣，也可用于机器翻译、文本生成等任务。	<code>paddlenlp.metrics.Perplexity</code>
BLEU(bilingual evaluation understudy)	机器翻译常用评价指标	<code>paddlenlp.metrics.BLEU</code>
Rouge(Recall-Oriented Understudy for Gisting Evaluation)	评估自动文摘以及机器翻译的指标	<code>paddlenlp.metrics.RougeL</code> , <code>paddlenlp.metrics.RougeN</code>
AccuracyAndF1	准确率及F1-score，可用于GLUE中的MRPC和QQP任务	<code>paddlenlp.metrics.AccuracyAndF1</code>
PearsonAndSpearman	皮尔森相关性系数和斯皮尔曼相关系数。可用于GLUE中的STS-B任务	<code>paddlenlp.metrics.PearsonAndSpearman</code>
Mcc(Matthews correlation coefficient)	马修斯相关系数，用以测量二分类的分类性能的指标。可用于GLUE中的CoLA任务	<code>paddlenlp.metrics.Mcc</code>
ChunkEvaluator	计算了块检测的精确率、召回率和F1-score。常用于序列标记任务，如命名实体识别(NER)	<code>paddlenlp.metrics.ChunkEvaluator</code>
Squad	用于SQuAD和DuReader-robust的评价指标	<code>paddlenlp.metrics.compute_predictions</code> , <code>paddlenlp.metrics.squad_evaluate</code>

参考：

https://github.com/nltk/nltk/blob/develop/nltk/translate/bleu_score.py#L483

https://www.nltk.org/_modules/nltk/translate/bleu_score.html

https://www.nltk.org/api/nltk.translate.bleu_score.html#module-nltk.translate.bleu_score

<https://github.com/PaddlePaddle/PaddleNLP/blob/develop/paddlenlp/metrics/bleu.py>

另外，最好读者有了一定bleu的概念后再来读本文，可能会更有感觉

```
In [1]: import math
import nltk
from collections import defaultdict,Counter
```

一些预备概念

预备知识1：Counter

下面的操作仅求交集和特殊并集有用，其他的放在这里仅供复习

对列表计数，统计词频

```
In [2]: gram1 = [(1, 2, 3), (4, 5, 6), (7, 8, 9), (1, 2, 3), (4, 5, 6)]
gram2 = [(1, 2, 3), (7, 8, 9), (1, 2, 3), (4, 5, 6), (7, 8, 9), (1, 2, 3), (4, 5, 6), (4, 5, 6), (7, 8, 9)]
c1 = Counter(gram1)
c2 = Counter(gram2)
c1, c2

Out[2]: (Counter({(1, 2, 3): 2, (4, 5, 6): 2, (7, 8, 9): 1}),
Counter({(1, 2, 3): 3, (7, 8, 9): 3, (4, 5, 6): 3}))
```

求并集、交集

```
In [3]: c1+c2, c1&c2

Out[3]: (Counter({(1, 2, 3): 5, (4, 5, 6): 5, (7, 8, 9): 4}),
Counter({(1, 2, 3): 2, (4, 5, 6): 2, (7, 8, 9): 1}))
```

求特殊并集

上面的并集是取出所有key，value是两个集合的value之和，就是统计两个集合的元素
由于特殊需求（暂且不提什么需求），要求并集的时候，元素个数只取较大的那个

```
In [4]: dict1 = Counter({'a': 1, 'b': 2, 'c': 3})
dict2 = Counter({'a': 4, 'b': 5, 'c': 2, 'd': 6})

uniset = dict1+dict2
result = {key: max(dict1.get(key,0), dict2.get(key,0)) for key in uniset.keys()}
uniset, result

Out[4]: (Counter({'a': 5, 'b': 7, 'c': 5, 'd': 6}), {'a': 4, 'b': 5, 'c': 3, 'd': 6})
```

```
In [5]: dict1 = Counter({'a': 1, 'b': 2, 'c': 3})
dict2 = Counter({'a': 4, 'b': 5, 'c': 2, 'd': 6})
dict3 = Counter({'a': 4, 'b': 5, 'c': 2, 'e': 6})
uniset = dict1+dict2+dict3
result = {key: max(dict1.get(key,0), dict2.get(key,0), dict3.get(key,0)) for key in uniset.keys()}
uniset, result

Out[5]: (Counter({'a': 9, 'b': 12, 'c': 7, 'd': 6, 'e': 6}),
{'a': 4, 'b': 5, 'c': 3, 'd': 6, 'e': 6})
```

动态更新词典(新字典键值对覆盖旧字典)

对于列表，统计再更新，对于字典，直接更新

```
In [6]: c1.update(["张三", (2,2,2),9, (1, 2, 3)])
c2.update({"李四":10,"a":2})
c1, c2

Out[6]: (Counter({(1, 2, 3): 3,
(4, 5, 6): 2,
(7, 8, 9): 1,
'张三': 1,
(2, 2, 2): 1,
9: 1}),
Counter({(1, 2, 3): 3, (7, 8, 9): 3, (4, 5, 6): 3, '李四': 10, 'a': 2}))
```

按值排序

```
In [7]: sorted(c1.items(),key=lambda x:x[1],reverse=True)



Out[7]: [(1, 2, 3), 3),
((4, 5, 6), 2),
((7, 8, 9), 1),
('张三', 1),
((2, 2, 2), 1),
(9, 1)]
```

集合不同元素个数、集合元素总个数

```
In [8]: len(c1), sum(c1.values())
```

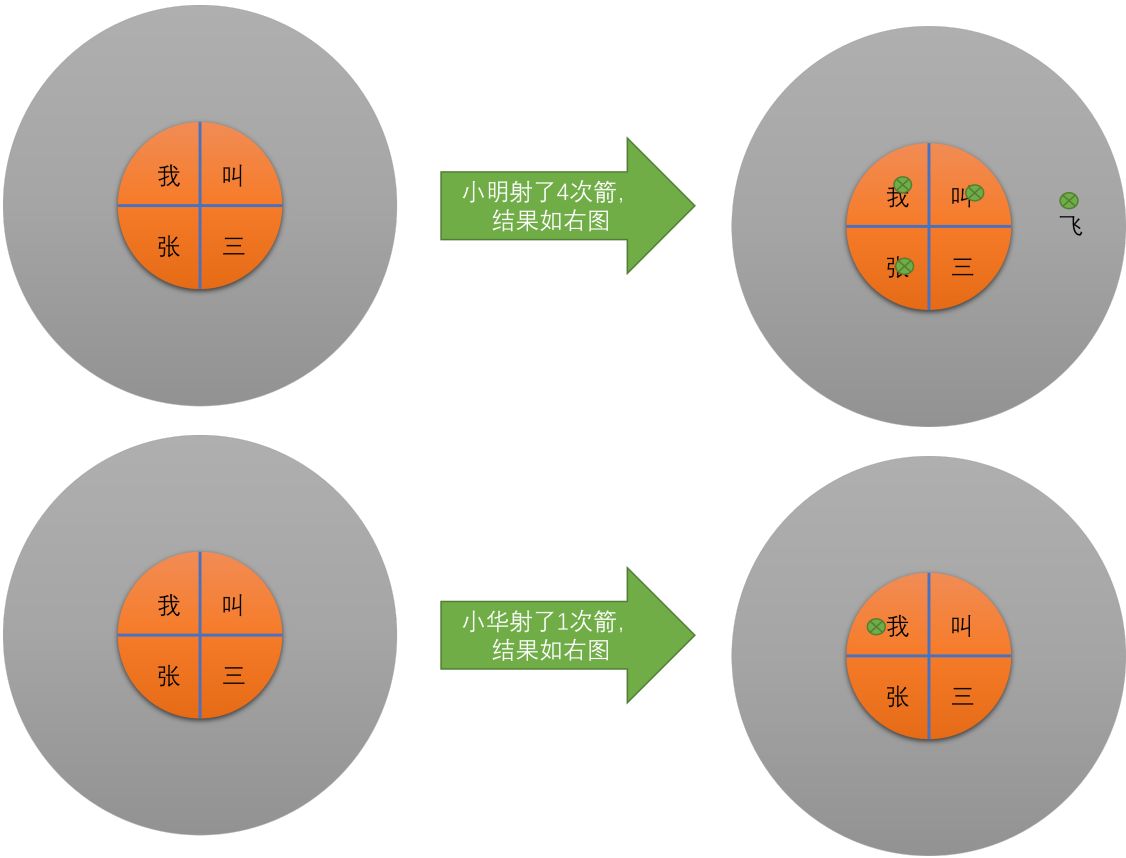
Out[8]: (6, 9)

预备知识2：准确率与召回率


	真实有病	真实没病
预测有病	1人 	0人
预测没病	4人	95人 


有100人来看病，医生给出的诊断如上图：
真有病的有5个人，医生诊断只认为里面1个人有病，其他4人诊断为没病
真没病的有95人，医生诊断认为都没病
可以看出来医生诊断的准确率为：96%
虽然准确率相当高，但是有病的大部分没诊断出来，只要医生的诊断倾向于没病，那么准确率就非常高，但是这样问题很大，有病的诊断不出来，这种问题可以用另一个评价指标召回率来衡量，就是有病的5人当中你到底诊断对了多少，这里的召回率为：1/5=20%

预备知识3：射击比赛



那么小明和小华谁的成绩好呢？
评价指标1，命中率：小明4发中3发，命中率75%；小华1发中1发，命中率100%；
评价指标2，覆盖率：红心靶4个，小明中了3个，覆盖率75%；小华中了1个，覆盖率25%。
覆盖率就是召回率，有病的就是正中心的靶子，4个有病的，小明医生预测对了3个，小华医生预测对了1个。

小明	红心部分	红心以外
射击命中	我，叫，张 	飞
没命中	三	

小华	红心部分	红心以外
射击命中	我 	null
没命中	叫，张，三	

翻译评估指标

直接借鉴上述评估指标

原文：“I am Zhang San”
参考翻译：“我叫张三”
小明翻译：“我叫张飞”
小华翻译：“我”

如果把参考翻译（标准答案）以及两人的翻译都切分成一个一个词来看：
参考翻译：“我”,“叫”,“张”,“三”
小明翻译：“我”,“叫”,“张”,“飞”
小华翻译：“我”
小明给出的翻译准确翻对了3个词，1个词翻错，1个词没翻到；
小华给出的翻译准确翻对了1个词，没有翻错的，3个词没翻到。
小明翻译的准确率：3/4=75%，覆盖率：3/4=75%

小华翻译的准确率：1/1=100%，覆盖率：1/4=25%
很显然只看准确率，小华更胜一筹，但是没什么用，仅仅翻出1个字没什么意义，
只看覆盖率的话，虽然小明大部分都翻出来了，好像很但关键的部分翻错了，
如果去北京翻成去北极，停车翻成开车，很显然问题也很大。

如果这样改进呢：
成绩=准确率x覆盖率
小明成绩=0.75x0.75=0.5625
小华成绩= 1x0.25=0.25
也就是说，我们考虑用准确率衡量两个人的成绩，
但是要同时用覆盖率给准确率打个折扣，如果你翻译出来一个较长的句子，准确率不是很高，但覆盖率很高，
分数还是有可能不错的，如果你翻译出来较短的句子，
即便零零散散翻对了，百分百准确，但是大部分没翻译，
你的准确率成绩还是要大打折扣的

新的问题1：
原文：“I am Zhang San”
参考翻译：“我”,叫”,张”,三”
小华翻译：“我”,我”,我”,我”
小华翻了4个“我”，从翻译角度来看，正确的翻译当中只有1个，
所以多余的3个其实都是按翻错处理的，所以此时小华的准确率
应该是1/4=25%，而不是按射击命中率100%来算。
此时的靶心被命中一次后，就不能被再次选中了。
因此我们的准确率公式不是正常的准确率公式了，而是修正后的准确率公式

新的问题2：
原文：“I am Zhang San”
参考翻译：“我”,叫”,张”,三”
小明翻译：“飞”,张”,我”,叫”
可以看出，即便打乱翻译的单词顺序，翻的乱七八糟，不影响小明的成绩
那么这种翻译流畅度的问题该如何解决呢？

Translation Accuracy-Recall Score (TARS)：综合前面所有考虑的打分机制

对于问题2，是因为我们只把句子切分成一个词一个词来看，这样就忽略了词之间的
顺序关系，准确率就无法体现句子的流畅性，所以我们可以分别切分成2、3、4个词来看

原文：“I am Zhang San”
参考翻译：“我叫张三”
小明翻译：“三张我叫”

参考翻译切分为1个词：“我”,叫”,张”,三”
小明翻译切分为1个词：“三”,张”,我”,叫”
修正后的准确率：100%，覆盖率：100%

参考翻译切分为2个词：“我叫”,“叫张”,“张三”
小明翻译切分为2个词：“飞张”,“张我”,“我叫”
修正后的准确率：1/3，覆盖率：1/3

参考翻译切分为3个词：“我叫张”,“叫张三”
小明翻译切分为3个词：“飞张我”,“张我叫”
修正后的准确率：0，覆盖率：0

参考翻译切分为4个词：“我叫张三”
小明翻译切分为4个词：“飞张我叫”
修正后的准确率：0，覆盖率：0

上面分别切分成1、2、3、4个词来看，相当于分别打了4次分数，现在我们要算综合分数，
于是就简单求和平均分来作为最后的分数，于是小明成绩=(1x1+1/3x1/3+0+0)/4=5/18

回顾与思考：一个高分的翻译应该长什么样呢？
①. 将标准答案分别切成1、2、3、4长度后，我的翻译里面也都有，
这就保证了大部分内容我翻了出来，并且很流畅，这点由覆盖率保证，
覆盖率保证了翻译内容的完整性。
②. 我翻译的东西，在标准答案之外的东西，要尽可能少，这就保证了
我翻译的准确率的问题，要想翻得准，北京翻译成北极的情况以及翻译
结巴连续翻相同词的情况都会降低我的分数

BLEU分数：更关注翻译的匹配程度

由于当前的机器翻译模型大多是自回归的语言模型，基本就是一个字一个字对着翻的，
训练的时候，就是拿标准答案一个一个比对，所以最后翻出来的结果一般完整性不会
丢失多少，就算丢失了，也可以通过错误率这个指标近似判断，比如准确率1%，很显然
翻了一大堆词，但是只翻对了一点点内容，通过这里也能判断翻译的内容完整性差的很。
BLEU分数就是看翻译的句子和标准答案的匹配程度如何，也就是准确率(修正版的准确率)
那么该如何计算呢？请看下面的例子

原文：“I am Zhang San”
参考翻译：“我叫张三”
小明翻译：“三张我叫”

参考翻译切分为1个词：“我”,叫”,张”,三”
小明翻译切分为1个词：“三”,张”,我”,叫”
修正后的准确率：100%

参考翻译切分为2个词：“我叫”,“叫张”,“张三”
小明翻译切分为2个词：“飞张”,“张我”,“我叫”
修正后的准确率：1/3

参考翻译切分为3个词：“我叫张”,“叫张三”
小明翻译切分为3个词：“飞张我”,“张我叫”
修正后的准确率：0

参考翻译切分为4个词：“我叫张三”
小明翻译切分为4个词：“飞张我叫”
修正后的准确率：0

小明4次打分的平均成绩=(1+1/3+0+0)/4=1/3

原文：“I am Zhang San”
参考翻译：“我叫张三”
小华翻译：“我”

参考翻译切分为1个词：“我”,叫",张",三”
小明翻译切分为1个词：“我” 修正后的准确率：100%

参考翻译切分为2个词：“我叫”,“叫张",“张三”
小明翻译切分为2个词：“”
修正后的准确率：0

参考翻译切分为3个词：“我叫张",“叫张三”
小明翻译切分为3个词：“”
修正后的准确率：0

参考翻译切分为4个词：“我叫张三”
小明翻译切分为4个词：“”
修正后的准确率：0

小华4次打分的平均成绩=(1+0+0+0)/4=1/4

新的问题：
从上面可以看出，虽然小华只翻了一个字，但是第一次打分高的离谱，
导致即使后面3次打分都得0分，最后平均下来还剩不少分，而后面切分
成多个词的打分实际上更能反映翻译的匹配程度，比如4个词为一个单位进行
匹配的话，要想匹配正确要比单个字匹配成功的难度大得多。
所以对于短的句子，其打分不应该太高。

解决方案一：
4次打分配分配的权重不一样，给长词匹配的分数高一点，短词匹配的分数低一点。
但是这种统一的分配权重，就忽略了翻译句子与标准答案之间的相对长度，
有些情况下，即便按不同权重分配，还是会失效（短句子打分还是很高）
这个怎么理解呢？请看下面的例子

原文：“I am Zhang San”
参考翻译：“我叫张三我叫张三我叫张三”
小明翻译：“我叫张三”
小华翻译：“我叫张三我叫张三我叫张三”

参考翻译切分为1个词：“我”,叫",张",三”,“我”,叫",张",三”,“我”,叫",张",三”
小明翻译切分为1个词：“我”,叫",张",三”
翻译切分为1个词：“我”,叫",张",三”,“我”,叫",张",三”,“我”,叫",张",三”
小明准确率：100%
小华准确率：100%

后面就不切了，计算结果都是一样的，也就是4次打分，两者完全一样，
即使按不同权重给每次打分，最后总分还是一样的

到这里有人就会发出疑问，你这样只打4次分数，前4次短句子长度实际上都“够”，
最后结果当然一样了，你后面再切分成5个词，6个词，不就解决了吗？
是的，没错，但是这个打多少次分，是很难决定的一件事，为什么这么说呢，
比如句子的长度范围从1到50，如果你一个单位的单词太长，比如30，也就是进行
30次打分，因为越长的单位后面越难匹配，我们必然给后面的分数更高的权重，
这样前面1、2、3、4的打分配分配的权重必然会稀释的很低很低，就会导致前几次
的打分不起作用，而看1、2、3、4个词的匹配程度对我们来说非常重要，因为大部分
句子翻译质量就看前4次左右的打分，后面太长了又能精确匹配的少之又少，导致
翻译的成绩呈现两极分化的状况，成绩特别好的寥寥无几，成绩差的一大堆。

如何解决？
我们不从每次打分的权重上下手，我们从最终成绩上下手，把最后4次打分的平均成绩
乘以一个惩罚项BP(Brevity Penalty)，就是短句惩罚项，根据你翻出来的句子长度和
标准答案的长度计算而来的，如果你的长度够，就不惩罚，如果不够，就会按不够的程度惩罚。
这个不够的程度就是bp惩罚项，怎么算的我们放在后面一块介绍

BLEU计算公式

$$bleuN = BP \times \exp\left(\sum_{i=1}^N w_i \log p_i\right)$$

符号说明：
 p_i ：就是第i次的打分，modified_precision，修正后的准确率
 w_i ：权重，默认就是1/4，平均的权重
 $bleu3$ ：进行3次打分的最终成绩
 bp ：短句惩罚项
log和exp：是因为直接加权求和后数值经常很小，进行适当缩放

p_i 计算的重新说明：
以下切词放到集合当中，相同的词按照不同的元素处理，比如：
“哈哈哈哈哈哈”按三个词切成(“哈哈哈哈”，“哈哈哈哈”，“哈哈”)}
这三个“哈哈”是不同的，不要合并成一个，如果另外一个集合是
(“哈哈”，“哈哈”)，那么它们的交集就是(“哈哈”，“哈哈”)

p_i 计算：
把翻译的句子和参考的句子分别按i个词切，分别放到两个集合A和B当中，求交集元素个数，
就是翻译匹配的数量，然后用匹配的数量除以翻译切的集合A中元素的数量，求出翻的准确率
如果还没有具体的概念，下面复现代码的时候会有例子，看例子就明白细节了

BLEU代码的复现

第一步：切词

这步在nltk里面没有，需要你自己切，下面是参考了paddlenlp中的切词

```
In [9]: def get_ngram(sent, n_size, label=None):  
        """
```

```
:param sent: ["我","叫","张","三"]
:param n_size: 2
:param label:
:return: [("我","叫"),("叫","张"),("张","三")]
"""

ngram_list = []
for left in range(len(sent) - n_size+1):
    ngram_list.append(sent[left : left + n_size ])
ngram_list=[tuple(gram) for gram in ngram_list]
return ngram_list
```

```
In [10]: # test
get_ngram(["我","叫","张","三"],2)
```

```
Out[10]: [('我', '叫'), ('叫', '张'), ('张', '三')]
```

第二步：对切好词的两个集合求交集，然后算准确率

这步在nltk里面叫modified_precision(), 在paddlenlp里面叫get_match_size()
所做的事情基本是一样的，就是计算 p_i 的分子和分母，分子就是匹配数，分母就是
预测给出的翻译(通常被叫做候选)切的集合的元素个数，这样分子除以分母就是准确率

```
In [11]: # 这个类后面会用到，用来存两个数并求商用的
from fractions import Fraction
frac = Fraction(3, 4)

print(frac.numerator)      # 输出: 3
print(frac.denominator)    # 输出: 4
print(float(frac))         # 输出: 0.75
```

```
3
4
0.75
```

另外，参考答案一般不止一个，比如我叫张三也可以是我是张三，这样就是两个句子，切出来的
集合就是两个，取哪个呢，就是取特殊并集，出现的元素按照两个集合里较大的那个算，然后拿
这个取了特殊并集后的集合当作标准答案，让预测翻译来匹配这个集合，然后算准确率

```
In [12]: def modified_precision(cand_ngram, refs_ngram):
        """
        :param cand_ngram: [("我","叫"),("叫","张"),("张","三")]
        :param refs_ngram: [[("我","叫"),("叫","张"),("张","三")], [("我","是"),("是","张"),("张","三")]]
        :return: Fraction(3, 3)
        """

        # 对切好词的列表进行统计
        cand_counts = Counter(cand_ngram)

        # 求出参考答案的ngram特殊并集
        refers=[Counter(ref) for ref in refs_ngram]
        max_counts = Counter()
        for key in cand_counts.keys():
            max_value=0
            for ref in refers:
                max_value=max(max_value,ref[key])
            max_counts[key]=max_value

        # 求交集
        matched_counts = {ngram: min(count, max_counts[ngram]) for ngram, count in cand_counts.items()}
        # 计算分子
        numerator = sum(matched_counts.values())
        # 计算分母，注意分母不能为0
        denominator = max(1, sum(cand_counts.values()))

        return Fraction(numerator, denominator, _normalize=False)
```

```
In [13]: # test1
cand_ngram=[("我","叫"),("叫","张"),("张","三")]
refs_ngram=[[("我","叫"),("叫","张"),("张","三")], [("我","是"),("是","张"),("张","三")]]
p_2=modified_precision(cand_ngram,refs_ngram)
p_2,p_2.numerator,p_2.denominator,float(p_2)
```

```
Out[13]: (Fraction(3, 3), 3, 3, 1.0)
```

```
In [14]: # test2
cand_ngram=[]
refs_ngram=[("我","叫"),("叫","张"),("张","三")]
p_2=modified_precision(cand_ngram,refs_ngram)
p_2,p_2.numerator,p_2.denominator,float(p_2)
```

```
Out[14]: (Fraction(0, 1), 0, 1, 0.0)
```

第三步：计算惩罚项

由于惩罚项是根据候选句以及参考答案句的长度计算而来，但是参考答案的句子有多句话，
那么该拿哪句话的长度呢？根据nltk中的公式，拿的是候选句长度最接近的那个句子，如果
两个句子一长一短，并且与候选句长度差值一样，也就是同样接近，这时选短的

```
In [15]: def brevity_penalty(cand_sent, refs_sent):
        # 计算候选句和参考句的长度
        cand_len=len(cand_sent)
        ref_lens = (len(reference) for reference in refs_sent)
        closest_ref_len = min(ref_lens, key=lambda ref_len: (abs(ref_len - cand_len), ref_len))

        # 候选句长度大于参考句长度时，bp=1，不对综合打分惩罚
        if cand_len > closest_ref_len:
            return 1

        # 候选句长度为0时，说明一个字都没翻译，直接令bp=0，bleu打分直接为0
        elif cand_len == 0:
            return 0

        # 候选句长度大于0小于等于参考句时，按如下公式计算惩罚项bp
        else:
            return math.exp(1 - closest_ref_len / cand_len)
```

第四步：分别计算p1,p2,p3,p4，然后计算bleu4-score，并和nltk库函数比较

```
In [16]: def sentence_bleu(cand_sent, refs_sent,weights=(0.25, 0.25, 0.25, 0.25)):
        """
        :param cand_sent: ["我","叫","张","飞"]
        :param refs_sent: [["我","叫","张","三"],["我","是","张","三"]]
        """

        # 计算4次的准确率
        p=[0,0,0,0]
        for n in range(1,5):
            cand_ngram=get_ngram(cand_sent,n)
            refs_ngram=[]
            for ref in refs_sent:
                refs_ngram.append(get_ngram(ref,n))
            p[n-1]=modified_precision(cand_ngram,refs_ngram)
```

```
# 计算短句惩罚项
bp=brevity_penalty(cand_sent, refs_sent)

# 计算最终打分
score = (w_i * math.log(p_i) for w_i, p_i in zip(weights, p) if p_i > 0)
score = bp * math.exp(math.fsum(score))

return score

In [17]: hypothesis1 = ['It', 'is', 'a', 'guide', 'to', 'action', 'which', 'ensures',
                    'that', 'the', 'military', 'always', 'obeys', 'the',
                    'commands', 'of', 'the', 'party']
reference1 = ['It', 'is', 'a', 'guide', 'to', 'action', 'that', 'ensures',
            'that', 'the', 'military', 'will', 'forever', 'heed',
            'Party', 'commands']
sentence_bleu(hypothesis1, [reference1])

Out[17]: 0.41180376356915777

In [18]: nltk.translate.bleu_score.sentence_bleu([reference1], hypothesis1)

Out[18]: 0.41180376356915777
```

分析短句子（缺少某个ngram）为什么nltk的bleu分数为0

```
references = [["我", "叫", "张", "三"], ["我", "是", "张", "三"]]
hypothesis = ["我", "叫", "张", "飞"]
weights = [0.25, 0.25, 0.25, 0.25]

In [20]: sentence_bleu(hypothesis, references)

Out[20]: 0.7071067811865476

In [21]: nltk.translate.bleu_score.sentence_bleu(references, hypothesis)

Out[21]: 8.636168555094496e-78

In [22]: chencherry = nltk.translate.bleu_score.SmoothingFunction()
nltk.translate.bleu_score.sentence_bleu(references, hypothesis, smoothing_function=chencherry.method0)

Out[22]: 8.636168555094496e-78
```

```
# Returns 0 if there's no matching n-grams
# We only need to check for p_numerators[1] == 0, since if there's
# no unigrams, there won't be any higher order ngrams.
if p_numerators[1] == 0:
    return 0 if len(weights) == 1 else [0] * len(weights)
```

如果没有1-gram, 直接返回0分

```
# If there's no smoothing, set use method0 from SmoothingFunction class.
if not smoothing_function:
    smoothing_function = SmoothingFunction().method0
# Smoothen the modified precision.
# Note: smoothing_function() may convert values into floats;
# it tries to retain the Fraction object as much as the
# smoothing method allows.
p_n = smoothing_function(
    p_n, references=references, hypothesis=hypothesis, hyp_len=hyp_lengths
)

bleu_scores = []
for weight in weights:
    # Uniformly re-weighting based on maximum hypothesis lengths if largest
    # order of n-grams < 4 and weights is set at default.
    if auto_reweigh:
        if hyp_lengths < 4 and weight == (0.25, 0.25, 0.25, 0.25):
            weight = (1 / hyp_lengths,) * hyp_lengths

    s = (w_i * math.log(p_i) for w_i, p_i in zip(weight, p_n) if p_i > 0)
    s = bp * math.exp(math.fsum(s))
    bleu_scores.append(s)
return bleu_scores[0] if len(weights) == 1 else bleu_scores
```

默认使用method0对4次打分p进行修改, 使得p=0的变成一个特别小的数

这样特别小的数大于0, 这里就能取log, 为负无穷, 然后exp后返回0分

```
def method0(self, p_n, *args, **kwargs):
    """
    No smoothing.
    """
    p_n_new = []
    for i, p_i in enumerate(p_n):
        if p_i.numerator != 0:
            p_n_new.append(p_i)
        else:
            _msg = str(
                "\n\nThe hypothesis contains 0 counts of {}-gram overlaps.\n"
                "Therefore the BLEU score evaluates to 0, independently of\n"
                "how many N-gram overlaps of lower order it contains.\n"
                "Consider using lower n-gram order or use "
                "SmoothingFunction()"
            ).format(i + 1)
            warnings.warn(_msg)
            # When numerator==0 where denominator==0 or !=0, the result
            # for the precision score should be equal to 0 or undefined.
            # Due to BLEU geometric mean computation in logarithm space,
            # we we need to take the return sys.float_info.min such that
            # math.log(sys.float_info.min) returns a 0 precision score.
            p_n_new.append(sys.float_info.min)
    return p_n_new
```

举个例子, 假如没有4-gram, 此时第4个p为0, 就会转到else执行, 使得newp添加一个特别小的数, 这个newp就是最后的4次准确率, 然后由于最后一个特别小, log后为负无穷, 导致其他打分跟它加了后还是负无穷, 最后取exp得到bleu分数为0


关于BLEU中的平滑：避免零除问题与过度惩罚

零除问题

比如翻译：i love apple, 标准答案：i like apple, 这就导致2-gram的准确率为0, 即使两句话意思差不多, 但是这样精确匹配会导致匹配数目为0, 可以简单将分子分母同时加1解决。从概率分布的角度来看, 某些低频词的匹配概率很低, 导致即使翻对了, 匹配数也为0, 这样准确率就会失真, 所以需要让概率分布更加平滑, 适当增加低频词的概率

过度惩罚问题

比如翻译：爽, 标准答案：太高兴了, 太爽了, 可以看出有些简短的翻译语义实际和长句子相似, 如果仅仅从句子单词匹配考虑, 会对短句惩罚过度, 从而得分非常低, 这时候对未匹配的n-gram 添加一个平滑项或进行概率分布的平滑处理就可以使得短句子也能有不错的分数, 对长句子分数也并没有多少影响, 具体细节请参考nltk的源代码



好的, 我会仔细计算确保准确性。

假设我们有以下参考翻译和生成的翻译：

参考翻译：I like to eat apples.
生成的翻译：I enjoy eating oranges.

我们将计算3-gram的准确率。

首先, 我们需要从参考翻译和生成的翻译中提取3-gram。在这个示例中, 参考翻译和生成的翻译中的3-gram如下所示：

参考翻译的3-gram: [“I like to”, “like to eat”, “to eat apples”]
生成的翻译的3-gram: [“I enjoy eating”, “enjoy eating oranges”]

根据公式, 我们需要计算匹配的3-gram数量和生成的3-gram的总数量。

匹配的3-gram数量: 0, 因为在参考翻译的3-gram中没有与生成的翻译的3-gram完全匹配的词组。
生成的3-gram总数量: 2, 因为生成的翻译中有2个3-gram。

所以, 准确率 = 匹配的3-gram数量 / 生成的3-gram总数量 = 0 / 2 = 0

因此, 在这种情况下, 3-gram的准确率为0, 表示生成的翻译中的3-gram没有与参考翻译中的3-gram完全匹配。

这个例子展示了低频n-gram的问题, 因为在参考翻译中没有出现的3-gram被认为是错误的, 导致准确率为0。然而, 实际上, 生成的翻译中的某些3-gram可能是合理的翻译片段, 但因为它们在参考翻译中没有出现, 所以被忽略了。这样的情况会导致BLEU评分的准确性不足。

```
In [23]: print(nltk.translate.bleu_score.sentence_bleu([reference1], hypothesis1))

print(nltk.translate.bleu_score.sentence_bleu([reference1], hypothesis1, smoothing_function=chencherry.method0))

print(nltk.translate.bleu_score.sentence_bleu([reference1], hypothesis1, smoothing_function=chencherry.method1))

print(nltk.translate.bleu_score.sentence_bleu([reference1], hypothesis1, smoothing_function=chencherry.method2))

print(nltk.translate.bleu_score.sentence_bleu([reference1], hypothesis1, smoothing_function=chencherry.method3))

print(nltk.translate.bleu_score.sentence_bleu([reference1], hypothesis1, smoothing_function=chencherry.method4))

print(nltk.translate.bleu_score.sentence_bleu([reference1], hypothesis1, smoothing_function=chencherry.method5))

print(nltk.translate.bleu_score.sentence_bleu([reference1], hypothesis1, smoothing_function=chencherry.method6))

print(nltk.translate.bleu_score.sentence_bleu([reference1], hypothesis1, smoothing_function=chencherry.method7))

0.41180376356915777
0.41180376356915777
0.41180376356915777
0.4452945001507636
0.41180376356915777
0.41180376356915777
0.4905328138015114
0.41358958106633686
0.4905328138015114
```

关于段落或其他语境级别的BLEU

假如我们翻译5句话, 分别对应5个参考 (每个参考里面包含对那句话的多个标准答案) 现在定义这5句话的翻译质量, 给这5句话打分

方案一：分别计算每句话的打分, 最后求平均

一般这种情况适用于句子之间非常独立, 没有上下文, 不考虑上下文的连贯性
计算方法: 每句话逐个求bleu分数, 最后求平均, 看平均分

方案二：基于上下文, 综合打分

比如第一次打分, 假如这5个句子代表5份考题, 分别由小A, 小B, 小C, 小D, 小E翻译, 然后评委进行第一次准确率计算, 会统计他们考卷上分别答对了多少, 最后算准确率的时候, 并不是一张一张考卷单独计算准确率, 而是把答对的数量求和作为分子, 然后把答题的总数目也求和作为分母, 然后求出准确率 这种打分方案就是基于上下文的打分, 是对这5个人的共同作战能力进行计算, 而不是求每个人能力的平均 所以当你的语料库需要考虑上下文的连贯性时, 才使用这种打分方案, 并且nltk和paddlenp里面默认的就是这种打分方案, 你看看你的一个batch里面的句子是否构成上下文关系

两种打分在库里面的实现机制

在nltk里面, 实现了corpus_bleu和sentence_bleu, 两者分开的, 想用哪个就用哪个, 一般不会出错; 实现细节: corpus_bleu会遍历上面5个人, 然后把他们答对的ngram累加, 最后求准确率; 对于sentence_bleu, 直接调用corpus_bleu, 让一个人来参加考试即可 在paddlenp里面, 通过update将答对的ngram数量累加到metric的属性当中, 实现的是语境bleu, 至于句子bleu, 需要把句子拆成一条一条算, update后, 计算下一条句子时, 立马reset

将模型输出转换为BLEU的输入

```
In [24]: import torch
import torch.nn as nn
import torch.nn.functional as F
```

```
In [25]: def default_trans_func(output, label, seq_mask, vocab):
    seq_mask = np.expand_dims(seq_mask, axis=2).repeat(output.shape[2], axis=2)
    output = output * seq_mask
    idx = np.argmax(output, axis=2)
    cand, ref_list = [], []
    for i in range(idx.shape[0]):
        token_list = []
        for j in range(idx.shape[1]):
            if seq_mask[i][j][0] == 0:
                break
            token_list.append(vocab[idx[i][j]])
        cand.append(token_list)

    label = np.squeeze(label, axis=2)
    for i in range(label.shape[0]):
        token_list = []
        for j in range(label.shape[1]):
            if seq_mask[i][j][0] == 0:
                break
            token_list.append(vocab[label[i][j]])

        ref_list.append([token_list])
    return cand, ref_list
```

```
In [26]: import numpy as np

# 假设以下是函数所需的输入数据
output = np.array([[0.1, 0.2, 0.7], [0.3, 0.5, 0.2], [0.2, 0.4, 0.4]],
                  [[0.4, 0.1, 0.5], [0.2, 0.3, 0.5], [0.6, 0.2, 0.2]])

label = np.array([[0], [1], [2]], [[1], [0], [2]])

seq_mask = np.array([[1, 1, 1], [1, 1, 0]])

vocab = {0: 'apple', 1: 'banana', 2: 'orange'}

# 调用函数
cand, ref_list = default_trans_func(output, label, seq_mask, vocab)

# 打印结果
print("Candidate:", cand)
print("Reference:", ref_list)
```

Candidate: [['orange', 'banana', 'banana'], ['orange', 'orange']]
Reference: [['apple', 'banana', 'orange'], ['banana', 'apple']]

其他地方你可能看到的不一样的计算BLEU的方法

李沐的计算公式

$$bleuN = BP \times \exp\left(\sum_{i=1}^N w_i \log p_i\right)$$

符号说明：

p_i : 就是第i次的打分，modified_precision，修正后的准确率

w_i : 权重，这里不是1/n了，变成 $1/2^n$

转化为：

$$bleuN = BP \times \exp\left(\sum_{i=1}^N \frac{1}{2^i} \log p_i\right) = BP \times \prod_{i=1}^N p_i^{0.5^n}$$

实际上从这里可以看出，log和exp没了，其本质是求p的几何平均；也能看出为什么某个pi为0后，整个bleu为0

```
In [27]: import math
import collections

def bleu(pred_seq, label_seq, k):
    """李沐中的bleu计算方法"""
    pred_tokens, label_tokens = pred_seq.split(' '), label_seq.split(' ')
    len_pred, len_label = len(pred_tokens), len(label_tokens)
    score = math.exp(min(0, 1 - len_label / len_pred)) # 这是惩罚项

    for n in range(1, k + 1):
        num_matches, label_subs = 0, collections.defaultdict(int)
        for i in range(len_label - n + 1):
            label_subs[' '.join(label_tokens[i: i + n])] += 1
        for i in range(len_pred - n + 1):
            if label_subs[' '.join(pred_tokens[i: i + n])] > 0:
                num_matches += 1
                label_subs[' '.join(pred_tokens[i: i + n])] -= 1
        score *= math.pow(num_matches / (len_pred - n + 1), math.pow(0.5, n))

    return score
```

```
In [28]: # 使用示例
pred_sequence = "我叫张 飞"
label_sequence = "我叫 张 三"
k = 3 # 最大匹配到3-gram，如果设为4，算出来为0

bleu_score1 = bleu(pred_sequence, label_sequence, k)
bleu_score2 = sentence_bleu(["我", "叫", "张", "飞"], ["我", "叫", "张", "三"])

print("李沐:", bleu_score1)
print("自己手写的:", bleu_score2)
```

李沐: 0.7175944439422446
自己手写的: 0.7071067811865476

由此看出，即便实现的不是一个公式（主要区别在权重系数），分数是差不多的
注意李沐这里对于ngram不存在的情况没有处理手段，会报错