1.总览

分词方法	特点	被提出的时间	典型模型
BPE	采用合并规则，可以适应未知词	2016年	GPT-2、RoBERTa
WordPiece	采用逐步拆分的方法，可以适应未知词	2016年	BERT
Unigram LM	采用无序语言模型，训练速度快	2018年	XLM
SentencePiece	采用汉字、字符和子词三种分词方式，支持多语言	2018年	T5、ALBERT

2.背景与基础

在使用GPT BERT模型输入词语常常会先进行tokenize，tokenize的目标是把输入的文本流，**切分成一个个子串，每个子串相对有完整的语义**，便于学习embedding表达和后续模型的使用。

tokenize有三种粒度：**word/subword/char**

- **word/词**，词，是最自然的语言单元。对于英文等自然语言来说，存在着天然的分隔符，如空格或一些标点符号等，对词的切分相对容易。但是对于一些东亚文字包括中文来说，就需要某种分词算法才行。顺便说一下，Tokenizers库中，基于规则切分部分，**采用了spaCy和Moses两个库**。如果基于词来做词汇表，由于长尾现象的存在，**这个词汇表可能会超大**。像Transformer XL库就用到了一个**26.7万**个单词的词汇表。这需要极大的embedding matrix才能存得下。embedding matrix是用于查找取用token的embedding vector的。这对于内存或者显存都是极大的挑战。常规的词汇表，**一般大小不超过5万**。
- **char/字符**，即最基本的字符，如英语中的'a','b','c'或中文中的'你'，'我'，'他'等。而一般来讲，字符的数量是**少量有限**的。这样做的问题是，由于字符数量太小，我们在为每个字符学习嵌入向量的时候，每个向量就容纳了太多的语义在内，学习起来非常困难。
- **subword/子词级**，它介于字符和单词之间。比如说'Transformers'可能会被分成'Transform'和'ers'两个部分。这个方案**平衡了词汇量和语义独立性**，是相对较优的方案。它的处理原则是，**常用词应该保持原状，生僻词应该拆分成子词以共享token压缩空间**

3.常用的tokenize算法

最常用的三种tokenize算法：BPE（Byte-Pair Encoding），WordPiece和SentencePiece

3.1 BPE（Byte-Pair Encoding）

BPE，即字节对编码。其核心思想在于将**最常出现的子词对合并，直到词汇表达到预定的大小时停止。**

BPE是一种基于数据压缩算法的分词方法。它通过不断地合并出现频率最高的字符或者字符组合，来构建一个词表。具体来说，BPE的运算过程如下：

1. **将所有单词按照字符分解为字母序列。例如：“hello”会被分解为["h","e","l","l","o"]。**
2. **统计每个字母序列出现的频率，将频率最高的序列合并为一个新序列。**
3. **重复第二步，直到达到预定的词表大小或者无法再合并。**

> **Note**

> 无法再合并就是指在当前数据中，**没有足够的共现频率支持继续合并**，也就是说：
>
> - 所有可合并的组合都已经合并过了
> - 或者剩下的组合频率太低，不足以触发合并
> - 或者只剩下单个字符，没有相邻字符对了

词表大小通常先增加后减小，每次合并后词表可能出现3种变化：

- `+1`，表明加入合并后的新字词，同时原来的2个子词还保留（2个字词不是完全同时连续出现）
- `+0`，表明加入合并后的新字词，同时原来的2个子词中一个保留，一个被消解（一个字词完全随着另一个字词的出现而紧跟着出现）
- `-1`，表明加入合并后的新字词，同时原来的2个子词都被消解（2个字词同时连续出现）

类型	含义	举例
`+1`	新增了一个符号，原两个子词仍然存在	AB 替代 ab，但 a 和 b 还出现在别处
`0`	新增了一个符号，其中一个子词被完全消解	AB 替代 ab，其中 b 只出现在 a 后面
`-1`	新增了一个符号，两个子词都被消解	AB 替代 ab，a 和 b 都只出现在一起

举例如下：

假设我们有以下单词：

```
low
lower
newest
widest
newest
widest
widest
widest
nice
```

首先将每个单词按照字符切分：

```
['l o w </w>',
'l o w e r </w>',
'n e w e s t </w>',
'w i d e s t </w>',
'n e w e s t </w>',
'w i d e s t </w>',
'w i d e s t </w>',
'w i d e s t </w>',
'n i c e </w>']
```

统计每两个相邻字符序列出现的频率：

```
{"es": 6, "st": 6, "t</w>": 6, "wi": 4, "id": 4, "de": 4, "we": 3, "lo": 2, "ow":
2, "ne": 2, "ew": 2, "w</w>": 1, "er": 1, "r</w>": 1, "ni": 1, "ic": 1, "ce": 1,
"e</w>": 1}
```

将出现频率最高的字符序列"**es**"进行合并，得到新的词表：

```
['l o w </w>',
'l o w e r </w>',
'n e w es t </w>',
'w i d es t </w>',
'n e w es t </w>',
'w i d es t </w>',
'w i d es t </w>',
'w i d es t </w>',
'n i c e </w>']
```

重复上述步骤，将出现频率最高的字符序列"st"进行合并,直到达到预定的词表大小或者无法再合并。

```
['lo w </w>', 'lo w e r </w>', 'n e w est</w>', 'widest</w>', 'n e w est</w>',
'widest</w>', 'widest</w>', 'widest</w>', 'n i c e </w>']
```

分词逻辑（Greedy Longest Match）：对于每一个单词，我们从左到右扫描，尝试用词表中 **最长的 subword** 匹配当前剩余部分。

- 如果匹配成功，则把这个 subword 加入结果，并跳过相应字符数。
- 如果没有匹配项，则继续尝试下一个较短的 subword。
- 如果所有 subword 都不匹配，就保留原字符或报错（取决于实现）

```
# 给定单词序列
[“the</w>”, “highest</w>”, “mountain</w>”]

# 已有的子词词表（按长度从长到短排序）
[“errrr</w>”, “tain</w>”, “moun”, “est</w>”, “high”, “the</w>”, “a</w>”]

# 迭代结果
"the</w>" -> ["the</w>"]
"highest</w>" -> ["high", "est</w>"]
"mountain</w>" -> ["moun", "tain</w>"]
```

代码

```
from collections import Counter
corpus='''low
lower
newest
widest
newest
widest
widest
widest
nice'''
import regex as re
# corpus=corpus.split('\n')
VOVAB_LENGTH=10
# corpus_char_counter=Counter(''.join((corpus)))
# print(dict(corpus_char_counter))
```

```
def get_status(corpus):
    # 统计相邻元素 XY出现的频率
    #  找出最大者
    merge_chars=[]
    for item in corpus:
        char_list=item.split(' ')
        for i in range(len(char_list)-1):

            merge_chars.append(''.join(char_list[i:i+2]))

    chars_count=Counter(merge_chars)
    most_common=chars_count.most_common(1)
    return most_common[0][0]
def merge_chars(corpus,chars_most_common):
    # 和并上一步得到的出现频率最大元素
    for idx,item in enumerate(corpus):
        _=re.sub('\s*'.join(chars_most_common),chars_most_common,item)
        corpus[idx]=_
    return corpus
def init(words):
    for idx,word in enumerate((words)):
        words[idx]=' '.join(list(word))+' </w>'
    return words
words=corpus.split('\n')
corpus=init((words))


while len(set(' '.join(corpus).split(' ')))>VOVAB_LENGTH:
    print(corpus)
    most_common=get_status(corpus)
    print(most_common)

    corpus=merge_chars(corpus,most_common)
    print(corpus)
```

3.2 WordPiece

WordPiece，从名字好理解，它是一种子词粒度的 tokenize 算法 subword tokenization algorithm，很多著名的 Transformers 模型，比如 BERT/DistilBERT/Electra 都使用了它。

WordPiece 算法可以看作是 BPE 的变种。不同的是，WordPiece 基于概率生成新的 subword 而不是下一最高频字节对。WordPiece 算法也是每次从词表中选出两个子词合并成新的子词。 **BPE 选择频数最高的相邻子词合并，而 WordPiece 选择使得语言模型概率最大的相邻子词加入词表。** 即它每次合并的两个字符串 A 和 B，应该具有最大的 $\frac{P(AB)}{P(A)P(B)}$值。合并 AB 之后，所有原来切成 A+B 两个 tokens 的就只保留 AB 一个 token，整个训练集上最大似然变化量与 $\frac{P(AB)}{P(A)P(B)}$成正比。

$$\log P(S)=\sum_{i=1}^{n}\log P(t_i)$$

$$S=[t_1,t_2,t_3,\ldots,t_n]$$

> **Note**
>
> **WordPiece 合并 A 和 B 的依据是它们的联合概率是否显著高于各自独立出现的概率乘积 。**
> 也就是说，WordPiece 判断的是：$score(A,B)=\frac{P(AB)}{P(A)P(B)}$

> - 如果这个比值很大 ⇒ 表示 A 和 B 经常一起出现，适合合并成一个 token。
> - 如果这个比值接近 1 或者很小 ⇒ A 和 B 没有强关联性，不值得合并。
>
> 比如说 $P(ed)$的概率比$P(e)+P(d)$单独出现的概率更大，可能比他们具有最大的互信息值，也就是两子词在语言模型上具有较强的关联性。

那 WordPiece 和 BPE 的区别：

- **BPE：apple 当词表有 appl 和 e 的时候，apple 优先编码为 appl 和 e**（即使原始预料中 app 和 le 的可能性更大）
- **WordPiece：根据原始语料，app 和 le 的概率更大**

类型	合并依据	示例
BPE	相邻子词共现频率最高	apple → appl + e
WordPiece	联合概率更高	apple → app + le（如果语言模型认为这样更合理）

3.3 Unigram

与 BPE 或者 WordPiece 不同，Unigram 的算法思想是从一个巨大的词汇表出发，再逐渐删除（trim down）其中的词汇，直到 size 满足预定义。

初始的词汇表**可以采用所有预分词器分出来的词，再加上所有高频的子串。**每次从词汇表中删除词汇的原则是**使预定义的损失最小。**训练时，计算 loss 的公式为：

$$\text{Loss} = -\sum_{i=1}^{N} \log\left(\sum_{x \in S(x_i)} p(x)\right)$$

- x_i：第 i 个训练样本（一个词或句子）
- $S(x_i)$：这个词的所有可能的 subword 切分方式
- $p(x)$：每种切分方式的概率之积（或直接由模型给出）

loss 越小 ⇒ 当前词表越能准确表达语料。

1.假设训练文档中的所有词分别为 $x_1; x_2, \ldots, x_N$，而**每个词 tokenize 的方法**是一个集合 $S(x_i)$。

2.当一个词汇表确定时，每个词 tokenize 的方法集合 $S(x_i)$ 就是确定的，而每种方法对应着一个概率 $P(x)$。

3.如果从词汇表中删除部分词，则某些词的 tokenize 的种类集合就会变少，$\log(\cdot)$ 中的求和项就会减少，从而增加整体 loss。

4.Unigram 算法每次会从词汇表中挑出**使得 loss 增长最小的 10%~20% 的词汇**来删除。

5.一般 Unigram 算法会与 SentencePiece 算法连用。

3.4 SentencePiece

SentencePiece，顾名思义，它是**把一个句子看作一个整体，再拆成片段**，而没有保留天然的词语的概念。一般地，它**把空格space也当作一种特殊字符来处理，再用BPE或者Unigram算法来构造词汇表。**

比如，XLNetTokenizer就**采用了_来代替空格**，解码的时候会再用空格替换回来。

目前，Tokenizers库中，所有使用了SentencePiece的都是与Unigram算法联合使用的，比如ALBERT、XLNet、Marian和T5.

> **Note**
>
> **SentencePiece 是一种“无视空格和词语边界”的通用分词器，它可以基于 BPE 或 Unigram 算法，对任意语言的句子进行统一的 subword 切分。**