北京航空航天大學

BEIHANG UNIVERSITY

自然语言处理实验二

中文分词

院 系 名 称	国际通用工程学院
任 课 老 师	丁嵘
学 生 姓 名	林威
学 生 学 号	19351024

2021 年 11 月 4 日

0. 理论介绍
1. 模型原理简介
    （一） 最大匹配算法
    （二）最少分词法（最短路径法）
2. 具体Python实现步骤
    （一）准备工作
    （二）最大匹配算法
    （三）最少分词法
3. 结果与改进
4. 词云绘制

0. 理论介绍

分词是自然语言处理的一个基本工作。对于中文来说，在中文的句子当中，字词之间是没有空格的，因而若想要让计算机自动识别句子的含义，第一步要做的就是将中文的词给分隔开。这个过程就是分词。

中文分词方法可以简单归类为

1. 基于词表的分词方法
2. 基于统计的分词方法
3. 基于序列标记的分词方法

在本次实践当中，我们将运用**最大匹配算法**与**最少分词法**对中文语料进行分词。

1. 模型原理简介

（一） 最大匹配算法

最大匹配算法是一种基于词典的分词方法。它按照一定的策略将待分词文本的字符串与词典进行匹配，若在词典中找到该字符串，则匹配成功，认为该字符串是一个中文词。根据扫描方向的不同，最大匹配算法可以被分为正向最大匹配算法和逆向最大匹配算法。而将这两者结合则是双向最大匹配算法。

- 正向最大匹配算法

  该算法的基本流程如下

  1. 找到词典中最长的词条的长度max_len，令cur_len=max_len.
  2. 取待分词文本中的前cur_len个字，查找分词词典。
  3. 若分词词典中有这cur_len个字组成的词，则匹配成功。匹配字段被切分为一个词。进而对接下来的文本继续进行分词。
  4. 若分词词典中找不到这cur_len个字组成的词，则匹配失败。令cur_len=cur_len-1，继续步骤2，直到cur_len=1时,则将单个字分割。
  5. 重复该分词过程，直到全部文本被分割成功。
- 逆向最大匹配算法

  该算法与正向匹配算法相似，只不过扫描文本的方向是逆向，即从右至左。相应的，每次匹配不成功也是去掉最左边的一个字。实验表明逆向最大匹配算法要优于正向最大匹配算法。
- 双向最大匹配算法

  双向最大匹配算法是将正向最大匹配算法得到的分词结果与逆向最大匹配算法进行比较，从而选择最优分词结果的方法。

这里我采用一些启发式规则来选择最优分词结果：

1. 如果正向与逆向算法分词结果的词数不用，则选择词数少的结果。
2. 如果词数相同：

   a) 分词结果不同，选择单个字作为词的数量较少的。

   b）分词结果相同，随意选择一个。

（二）最少分词法（最短路径法）

最少分词法是一种基于词典的分词算法，每个句子生成一个有向无环图，节点代表字的间隔，边代表可能的分词。其具体过程如下：

1. 设待分词文本中有n个字。建立一个节点数为n+1的有向无环图$G=\{V_0,V_1,\dots V_n\}$，并在每个相邻节点V_k,V_{k+1}之间建立有向边$E_{k,k+1}$，边对应的词（字）为c_k。
2. 扫描字典，看字典中的词有没有出现在待分词文本中。如有，设这个词为$w=c_ic_{i+1}\dots c_j$，则在节点V_{i-1},V_j之间建立有向边$E_{i-1,j}$，边所对应的词为w。
3. 重复步骤2，直到没有新的边可以加入。
4. 用最短路径算法（这里采用Dijkstra Algorithm）找到最短路径。最短路径的边所对应的词的序列即为分词结果。

2. 具体Python实现步骤

（一）准备工作

先将所给的数据集词典和待分词文件做一个初步的处理。将字典文件储存在python的dictionary里面，key为词，value为True。

```
def get_dic(path):
    f=open(path,'r',encoding='utf-8')
    l=[]
    dic={}
    for line in f.readlines():
        l.append(line[:-1])
        dic[line[:-1]]=True
    f.close()
    return dic
```

将待分词文件转化为list。

```
def get_corpus(path):
    f=open(path,'r',encoding='utf-8')
    l=[]
    for line in f.readlines():
        if len(line)>1:
            l.append(line[:-1])
    f.close()
    return l
```

再写一个函数来判断一个词是否在词典当中

```
def is_word(try_word,dic):
    re=dic.get(try_word,False)
    return re
```

（二）最大匹配算法

根据上文中的描述，实现正向最大匹配算法

```
def FMM(sen,max_len,dic):
    res=[]
    while len(sen)>0:
        cur_len=max_len
        while cur_len>1:
            try_word=sen[:cur_len]
            if utils.is_word(try_word,dic):
                res.append(try_word)
                sen=sen[cur_len:]
                break
            else:
                cur_len-=1
        if cur_len==1:
            try_word=sen[:cur_len]
            res.append(try_word)
            sen=sen[cur_len:]
    return res
```

类似的，实现逆向最大匹配算法

```
def BMM(sen,max_len,dic):
    res=[]
    while len(sen)>0:
        cur_len=max_len
        while cur_len>1:
            try_word=sen[-1*cur_len:]
            if utils.is_word(try_word,dic):
                res.append(try_word)
                sen=sen[:-1*cur_len]
                break
            else:
                cur_len-=1
        if cur_len==1:
            try_word=sen[-1*cur_len:]
            res.append(try_word)
            sen=sen[:-1*cur_len]
    res.reverse()
    return res
```

将两个算法得到的结果进行对比，并根据选取最优结果

```
def merge(BMM_result,FMM_result):
    final_result=[]
    for i in range(len(BMM_result)):
        b=BMM_result[i]
        f=FMM_result[i]
        if len(b)<len(f):
            final_result.append(b)
        elif len(f)<len(b):
            final_result.append(f)
        else:
            b_single=[len(i) for i in b].count(1)
```

```
            f_single=[len(i) for i in f].count(1)
            if b_single<f_single:
                final_result.append(b)
            else:
                final_result.append(f)
    return final_result
```

以下是双向最大匹配算法的主体部分

```
def BM(corpus,dic,max_len):
    BMM_result=[]
    FMM_result=[]
    for line in tqdm(corpus):
        FMM_result.append(FMM(line,max_len,dic))
        BMM_result.append(BMM(line,max_len,dic))
    final_result=merge(BMM_result,FMM_result)
    return final_result
```

（三）最少分词法

这里我用了networkx来建立有向图，并调用network.shortest_path来计算最短路径，具体代码如下

```
def SPath(dic,corpus):
    result=[]
    for line in tqdm(corpus):
        G=nx.DiGraph()
        tuple2zi={}
        edges=[]
        for i,zi in enumerate(line):
            tuple2zi[(i,i+1)]=zi
            edges.append((i,i+1))
        G.add_edges_from(edges)

        new_edges=[]
        for word in dic:
            pos=line.find(word)
            while pos!=-1:
                s=pos
                e=pos+len(word)
                new_edges.append((s,e))
                pos=line.find(word,pos+1)
        G.add_edges_from(new_edges)
        sp=nx.shortest_path(G,0,max(G.nodes))

        re=[]
        for i in range(len(sp)-1):
            word=''
            for j in range(sp[i],sp[i+1]):
                word+=tuple2zi[(j,j+1)]
            re.append(word)
        result.append(re)
    return result
```

3. 结果与改进

将上述算法的分词结果与真实结果进行对比，可以计算精确率P，召回率R与F1值。设分词结果为集合A，真实结果为集合B,那么分词正确的结果则为$A \cap B$。精确率则为$P = \frac{|A \cap B|}{A}$,召回率为$R = \frac{|A \cap B|}{B}$，$F1 = \frac{2*P*R}{P+R}$。

具体结果如下：

	精确率P	召回率R	F1值
最大匹配算法	0.9090	0.8452	0.8759
最少分词法	0.9074	0.8437	0.8744

考虑到日期或者数字（算法会把数字分为单个数字）这种具有固定形式的词组可能并不会出现在词典中，我们可以手动地把它们设为词。

对上面地is_word函数进行改进

```
def is_number(word):
    ch2ara=
{'零':0,'一':1,'二':2,'三':3,'四':4,'五':5,'六':6,'七':7,'八':8,'九':9,'十':10}
    for ch in ch2ara:
        word=word.replace(ch,str(ch2ara[ch]))
    return word.isdigit()

def is_word(try_word,dic):
    re=dic.get(try_word,False)
    if try_word[-1] in ['年',"月","日",'点','时','分','秒','万','十万','百
万','千万','亿']:
        if is_number(try_word[:-1]):
            re=True
    if is_number(try_word):
        re=True
    return re
```

改进后，最大匹配算法的结果有显著提升

	精确率P	召回率R	F1值
最大匹配算法（改进后）	0.9253	0.8975	0.9112

4. 词云绘制

这里我选用了金庸的小说《倚天屠龙记》的作为语料库，用最大匹配算法进行分词。并将分词结果中的停用词去除掉，调用wordcloud库绘制词云。具体效果如下：

说道
武功
不知
姑娘
弟子
心中
心想
兄弟
金花
婆婆
性命
昆仑
张三
想到
问道
敌人
众人
大侠
天下
女子
兵刃
冷笑
出手
和尚
此刻
说话
眼见
跟着
地道
身子
之下
江湖
双手
左手
右手
神功
大师
一眼
脸上
之间
苦头
少女
英雄
光明
全身
宝刀