中文分词作业

魏少杭
学号：20373594
学院：人工智能研究院

2022 年 10 月 13 日

目录

1 思路与过程

1.1 最大匹配算法实现

首先分别通过正向最大匹配和逆向最大匹配，获得各自算法的结果，结果用**分词后的词语表**（list 类型，且存在词语的重复情况）和**分词后的区间**表示。

1.1.1 正向最大匹配（FMM）实现细节

FMM 算法

```
#BM.py
def FMM(entry_list):
    """
    Through FMM, get the partition array(type:list) for words in the range
        given.
    param: entry_list:待分词的句子列表
    return: 分词结果词语表FMM_res(list) 、分词区间FMM_intervals(list)
    """
    # 初始化FMM中存词典的词
    FMM_res = []
    # 为了后续count区间，所以需要对所有的FMM预测区间进行存储
    FMM_intervals = []
    sentence_begin = 0
    line_num = 0
    for entry in entry_list:
        # 每一行进行处理，但为了和标准答案能够有对比，在计算区间intervals的
            时候下标仍然是整个文档中的下标
        line_num += 1
        print('FMM: ',line_num)      # 打印输出当前处理到第几行了
        sentence = entry.strip()     # 去除尾部的\n
        N = len(sentence)
        i = 0
        while i < N:    #注意：当用for i in range(N)的时候，每次是从一个迭代
            器里面查找的
            for j in range(min(N-1, i+21), i-1, -1):
                if j-i+1 > 22:  # 由于词典最长的词长度只有22，这个continue可
                    以减少不必要的时间浪费
                    continue
                if judge(sentence, i, j):   # 判断是否切分
                    FMM_res.append(sentence[i:j+1])
                    FMM_intervals.append((sentence_begin + i, sentence_begin
                        + j))
                    i = j
                    break
            i += 1
```

```
        sentence_begin += N
    return (FMM_res, FMM_intervals)
```

传入待分词的句子列表,然后对每一个句子从前往后匹配词语。输出匹配结果 FMM_res 和分词区间 FMM_intervals。

判断是否需要切分一个词的时候，进行了两个 if 条件判断，第一个 if 语句是由于词典最长词长度为 22，故 continue 可以减少不必要的时间浪费。

第二个 if 语句调用了 judge() 函数，i 和 j 分别为当前待判断的区间的左右端点，sentence 为当前的一个句子。为了能够与标准答案有对比，在计算区间 intervals 的时候下标仍然使用整个文档 txt 的下标，所以每次 intervals 列表中插入的时候都是用绝对下标 sentence_begin+i 和 sentence_begin+j 来表示。judge() 函数将在后面进行具体说明。

1.1.2 逆向最大匹配（BMM）实现细节

逆向最大匹配算法实现基本与 FMM 一致，其区别主要在于每一个句子的遍历切分的方向不同。每一个句子分词结果的产生顺序是由右向左。

以下是 BMM 算法中与 FMM 算法不同的地方：

BMM 算法

```
# BM.py
        i = N - 1
        while i >= 0:
            for j in range(max(0, i-21), i+1):
                if i-j+1 > 22:
                    continue
                # if isWord(sentence[j:i+1]) or i == j:
                if judge(sentence, j, i):   # 判断是否拆分
                    BMM_res.append(sentence[j:i+1])
                    BMM_intervals.append((sentence_begin + j, sentence_begin
                        + i))
                    i = j
                    break
            i -= 1
        sentence_begin += N
```

1.1.3 双向最大匹配（BM）算法

双向最大匹配就是将分别调用一次 FMM 和 BMM，对返回的分词结果进行比较并返回更优的结果。

首先比较 FMM 和 BMM 算法分词结果数目，选择较小者作为最大匹配的结果。若两个算法结果数目相同，则对分词的词结果去重，即将相同的结果取一次返回。

其次，若去重后的结果不同，则取去重后分词数较少者作为最大匹配的结果。若去重之后的结果相同，则返回 BMM 算法结果。

因为研究表明 *BMM* 算法的结果从经验上讲准确率更高。

下面是实现：

BM 算法实现

```
# BM.py
def BM(txt):
    """
    输入为一个文章段落,首先对文章进行预处理操作TODO:
    传入txt文件变量即可
    res表示BM的结果
    下面选取两个算法得到的结果中最好的结果
    flag: 如果为0,返回FMM,如果为1,返回BMM
    """
    entry_list = []
    for entry in txt:
        entry_list.append(entry.strip())
    FMM_res, FMM_intervals = FMM(entry_list)
    BMM_res, BMM_intervals = BMM(entry_list)
    if len(FMM_res) < len(BMM_res):
        flag = 0
    elif len(FMM_res) > len(BMM_res):
        flag = 1
    else:
        # 去重，取去重之后结果最小的那个作为结果
        if len(list(dict.fromkeys(FMM_res))) < len(list(dict.fromkeys(
            BMM_res))):
            flag = 0
        else:
            flag = 1
    if flag == 0:
        return (FMM_res, FMM_intervals)
    else:
        return (BMM_res, BMM_intervals)
```

1.1.4 模型评估

第一部分：获取分词的标准答案

主要在 get_std_intervals() 函数中实现，其输出为分词的标准答案中的区间结果。

实现如下：

get_std_intervals

```
# BM.py
```

```python
def get_std_intervals():
    with open('./分词对比文件/gold.txt', 'r', encoding='utf-8') as fr:
        std_string = ''
        for entry in fr:
            std_string = std_string + entry.strip('\n')
        len_passage = len(std_string)
        std_intervals = []
        left = 0
        j = -1
        i = 0
        std_words = []
        while i < len_passage:
            if i+2 < len_passage and std_string[i+1] == ' ' and std_string[i
                +2] == ' ':
                len_string = i - j
                std_intervals.append((left, left + len_string - 1))
                std_words.append(std_string[j+1: i+1])
                left = left + len_string
                i = i + 2
                j = i
            i += 1
        std_words = std_words
        return std_intervals
```

在标准答案的区间结果（list 类型）已经获取后，对模型进行计算评估。

在评估之前，必须要注意如何定义 Positive 类和 Negative 类。

Positive 即预测分词区间在标准分词结果区间中的个数；*Negative* 即预测分词区间不在标准分词结果区间中的个数；标准分词结果中的所有区间的总数即为 *TP+FN*；预测分词结果中的所有区间的总数即为 *TP+FP*；预测分词结果中的所有区间与标准分词结果中的所有区间的交集大小即为 *TP*；

第二部分：计算 TP

calc_TP

```python
# BM.py
def calc_TP(std_intervals, prd_intervals):
    TP = 0
    for interval in prd_intervals:
        if interval in std_intervals:
            TP += 1
    return TP
```

有了 TP 之后，就可以计算 P,R 和 F1 了。

第三部分，计算 P,R,F1

eval_PRF1

```
# BM.py
    TPaddFN = len(std_intervals)
    TPaddFP = len(prd_intervals)
    TP = calc_TP(std_intervals, prd_intervals)
    precision = TP / TPaddFP
    recall = TP / TPaddFN
    F1 = 2 * precision * recall / (precision + recall)
    return (precision, recall, F1)
```

计算结果在 *20373594-魏少杭/评估结果/eval_res_BM.txt* 中。即：

Presicion:90.1214%

Recall:92.7852%

F1:91.4339%

1.2 最少分词法/最短路径法

1.2.1 生成有向图

由于整个方法是将词模型转换为图模型，所以首先将一个句子转换为有向图，数据结构是一个矩阵 Graph，v_{ij} 为矩阵的元素，即从 i 到 j 结点是否存在路径，换句话说，即从 c_i 到 c_{j-1} 是否存在。初始时，设定每两个直接相邻结点之间存在路径，即 $v_{i,i+1} = 1$，矩阵其他元素均为 0。

其中判断是否有路径使用与最大匹配法一样的方法，在 *judge* 函数中实现。

get_graph

```
# leastSegment.py
def get_graph(sentence):
    '''
    input one sentence
    output a graph:
    '''
    # 初始化一个图，是一个点-点矩阵，每一个元素为1或0，表示是否有从一个点到另一个点的有向边
    M = len(sentence)   # M表示节点最大下标，恰好为sentence的长度值（因为从0开始算）
    graph = []
    for i in range(M + 1):
        graph.append([])
        for j in range(M + 1):
            graph[i].append(0)
        if i < M:
            graph[i][i + 1] = 1
```

```
    # 扫描整个句子，判断结点之间是否为有向边
    for i in range(M):
        for j in range(i+1, min(i + 23, M + 1)):      # 词最长为22
            if judge(sentence, i, j-1):
                graph[i][j] = 1             # 如果判断需要分词的话，就分出来

    return graph
```

1.2.2 最短路径-dijkstra 算法

分别用 succession（list）存最短路径上的前继结点，found（list）存已经找到最短路径的结点，not_found（list）存尚未找到最短路径的结点，shortest（ndarray）存整个图中已经探测到的最短路径长度。

初始化和主要算法部分如下：

dijkstra 算法主体

```
# leastSegment.py->dijkstra()
    # 初始化
    for i in range(M + 1):
        succession.append(i - 1)   # 初始化，用i-1来作为前继结点 succession
            [0] = -1
        if graph[0][i] == 1:
            shortest[i] = 1
    # 开始搜索
    while M not in found:
        # 如果从0结点到最后一个结点M的最短路径没有找到的话，不断找最短路径
        # 从没有找到最短路径的点里面找到当前探测的最短距离对应的第一个点
        # v_new_found = list(shortest).index(np.min(shortest[not_found]))
        temp_v_list = np.argwhere(shortest == np.min(shortest[not_found])).
            reshape(-1)
        for ver in temp_v_list:
            if ver in not_found:
                v_new_found = ver
                break
        # 需要新加入的点从not_found里删除，并加入到found中
        not_found.remove(v_new_found)
        found.append(v_new_found)
        # 找这个新加入的点的邻接节点并判断是否更新最短距离
        # 如果需要更新最短距离，则将这些结点的前继结点进行更新为v_new_found
        for v in not_found:
            if v_new_found == M:    # 找到了M结点的最短路径后就结束了
                break
            if (graph[v_new_found][v] == 1
                and 1 + shortest[v_new_found] < shortest[v]):
```

```
                shortest[v] = shortest[v_new_found] + 1
                succession[v] = v_new_found

    return get_path(succession, M)
```

其中还定义并调用了 get_path，通过传入前继结点集合，返回一条最短路径（结点序列）。

1.2.3 最少分词法整体算法

如下面代码所示，传入文本文件，输出预测的分词结果的列表、预测的分词区间：

least_segment

```
# leastSegment.py
def least_segment(txt):
    '''
    传入一个文本文件
    按行进行get_graph 并 dijkstra 来get_path
        然后用path来get_intervals作为这一行的intervals
    '''
    line_counter = 0
    prd_intervals = []
    prd_words = []
    for entry in txt:
        sentence = entry.strip()
        graph = get_graph(sentence)
        path = dijkstra(graph, len(sentence))
        line_intervals = get_intervals(path)
        prd_intervals.append(line_intervals)
        for (left, right) in line_intervals:
            prd_words.append(sentence[left: right+1])
        line_counter += 1

    return (prd_words, prd_intervals)
```

1.2.4 计算评估模型

与最大匹配算法相同的计算思路。

结果如下:

Presicion:87.8124%

Recall:91.8922%

F1:89.8060%

1.3 词云图

选取了鲁迅先生的《铸剑》全文，大约一万字。

1.3.1 最大匹配算法-词云图

word_cloud

```
# BM.py
def word_cloud():
    stop_words = [line.strip() for line in open('./停用词/stop_word.txt', 'r', encoding='utf-8')]
    # 获取想要得到的
    with open('./喜欢的文章段落/《铸剑》by鲁迅.txt', 'r', encoding='utf-8') as f:
        words, __ = BM(f) # 将str格式用BM算法来获取分词结果
        # 去除停用词，得到sentences
        sentences = ""
        for word in words:
            if word in stop_words:
                continue
            sentences += str(word)+' '
        # 生成词云图
        cloud_img = WordCloud(background_color='white',
                              font_path='./字体/plun04wkyso54jx5893el7t1zkz94ix1.otf',
                              width=2000,
                              height=2000,).generate(sentences)
        plt.imshow(cloud_img)
        plt.axis('off')
        plt.savefig('./词云/《铸剑》by鲁迅')
        plt.show()
```

该算法实现的词云图如图 1:

1.3.2 最少分词法-词云图

结果如图 2:

1.4 切分词的条件判断方法

对于一个给定的句子片段，判断为需要切分的词，需要满足以下几种条件之一:

1. 如果这个片段在词典中。
2. 如果这个片段长度仅为 1。

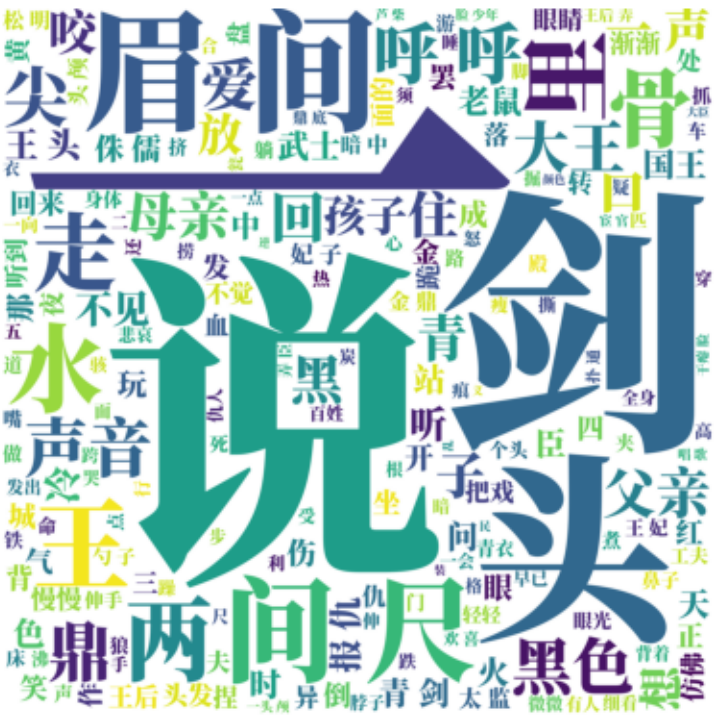

图 1: 《铸剑》词云图-BM 算法

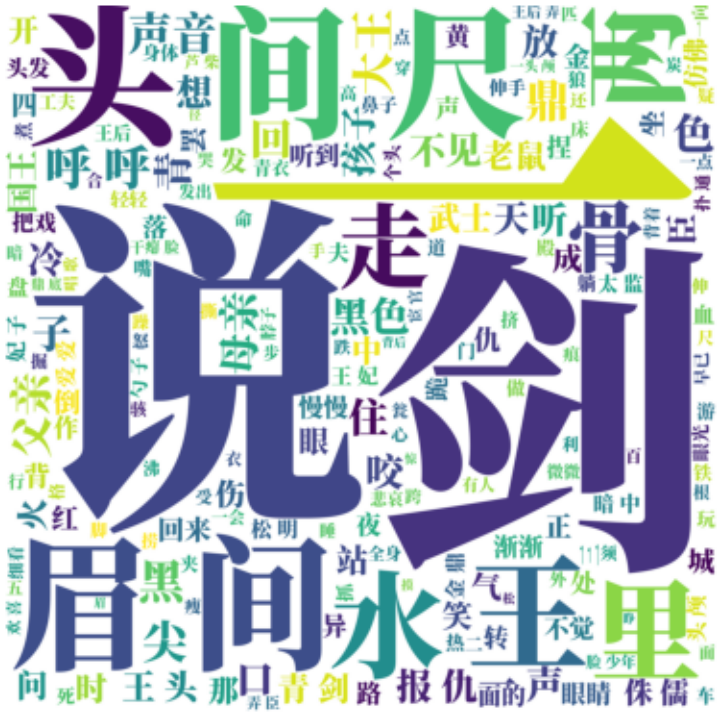

图 2: 《铸剑》词云图-Least Segment 算法

3. 如果这个片段为纯数字。在标准答案中，所有纯数字都分为一个词，且这些纯数字均为半角数字。而在词典中的所有纯数字均为全角数字，故仅查找词典不可能匹配成功。所以需要自己实现一个判断是否全为包含中文、阿拉伯数字的函数 *isAllNum()*。

4. 如果这个片段是日期。具体而言，末尾是‘年’，‘月’，‘日’，‘时’，‘分’等字的时候，就是一个日期。

按照以上思路，构建函数 judge()，传入需要判断的句子、待判断的左右端 i,j，输出布尔值 True|False。

判断是否分词 judge()

```
# leastSegment.py
def judge(sentence, i, j):
    # 此处的i<=j
    if (isWord(sentence[i:j+1])
        or i == j   # 如果只有单个字了
        or (isAllNum(sentence[i : j])              # 如果是 十九万八千 或 10238 整个串全是数字
            and (((sentence[j] in time_tags) or (sentence[j] == '年' and j - i == 4))
                 or isAllNum(sentence[j])))   # 如果是如 10月/23日/23时 或 如果是如 2001年或二○○一年
    ):
        return True
    return False
```

judge() 调用了判断片段是否全为数字（包括中文数字大写、中文进制数、百分号、全角 *0*、全角负号、中文小数标志‘点’字符、分数标志‘分’‘之’）的函数 isAllNum()。

isAllNum()

```
def isAllNum(word):
    for char in word:
        if char in Chinese_num:
            continue
        elif isNum(char):
            continue
        else:
            return False
    return True
```

isNum()

```
def isNum(char):
    '''判断是否是数字'''
    try:
        float(char)
        return True
```

```
    except ValueError:
        pass
    try:
        import unicodedata
        unicodedata.numeric(char)
        return True
    except (TypeError, ValueError):
        pass

    return False
```

2 总结和说明

2.1 两类算法的比较

双向最大匹配算法的三个评价标准优于最少分词算法。

但是两种算法作为基于词典的算法，在实现时并未经过消歧等操作，所以在实际预测结果中很多地方会出现不符合句子原意的分词方法。这需要额外的消歧手段，或者一些基于统计的算法也应该会有更好的效果。

而最少分词算法还存在多条路径情况下选择最优路径的原则不明确等情况。

2.2 算法上的局限性

在最少分词法中，我当前使用的是传统的单源最短路径的贪心算法-dijkstra 最短路径算法。由于这种贪婪算法实际上对于一个目标点计算出的最短路径只有一条，无法获取多条并依据一定的原则取舍一条最短路径作为输出结果，所以结果不一定可信。

由于时间紧迫，我无法实现广度优先搜索方法 BFS 实现最短路径搜索。实际上，如果一个句子较长时，可能出现许多条从第一个结点到最后一个结点的最短路径。如果使用 BFS 可以避免 dijkstra 算法仅计算一条最短路径的情况。BFS 可以实现一次性多条路径生成，但同样需要处理的问题是，如何制定多条最优路径的选择。经过查阅资料，可能选择标准有：去掉重复词后分词结果个数最少标准；分词结果集合中的分词长度的方差最小标准；分词结果中单字符结果最少标准等。

这两种基于词典的分词方法，效率十分低，时间复杂度极高。每个算法运行时间（cpu 跑）均在 20 分钟以上。

2.3 基于验证集和结果的改进

在验证集的对比文件中，发现了词典中不存在的词，如“多云转晴”在词典中不存在，这极大影响了模型评估的准确率等指标，因此，可以将这些在标准答案中反复出现的词加入到词典中。

但是对于较为大量的训练集或在验证集中出现次数较少的词语来说这种方式不切实际，这需要大量的人工去判断，操作较为复杂。

2.4 避免时间浪费的一些技巧

首先从词典中获取最长的词的长度，这样可以对后面每一次 i,j 所切分出的句子片段进行判断时，指定 i,j 的距离在词典最长词的长度之内，有效避免了长句子中的无效分词带来的时间浪费。