汉语分词系统

王家琪
1190301610
哈尔滨工业大学

摘要

本次实验任务是对汉语进行自动分词，流程包括：词典建立，分词算法，性能优化和评价等环节。本次试验考察了以下知识：最大正向匹配算法、最大反向匹配算法、查找优化等数据结构相关算法、基于统计的 n 元文法、未登录词识别、马尔可夫模型和隐马尔可夫模型等相关知识。

1 前言

中文分词指的是中文在基本文法上有其特殊性而存在的分词。分词就是将连续的字序列按照一定的规范重新组合成词序列的过程。在英文的行文中，单词之间以空格作为自然分界符，但是中文中词没有一个形式上的分界符，虽然英文也同样存在短语的划分问题，不过在分词这一任务上，中文比之英文要复杂得多、困难得多。

中文分词的难点表现为**歧义切分**、**分词规范**、**词义模糊**、**交集型切分问题**、**多义组合型歧义切分**、**未登录词识别**等等。由于人们认识水平的不同，对词和短语的边界很难去区分。例如：“对随地吐痰者给予处罚”，“随地吐痰者”本身是一个词还是一个短语，不同的人会有不同的标准，即使是同一个人也可能做出不同判断，除此之外，很多新生词汇，比如“奥力给”，“u1s1”，“笑 yue 了”等词的出现也对自动分词技术提出了挑战。

2 介绍

词是自然语言中能够独立运用的最小单位，是语言信息处理的基本单位。自动分词是正确的中文信息处理的基础。本实验所用到的基础原理是：贝叶斯条件概率公式，动态规划。

现有的分词算法可分为三类：**基于字符串匹配分词算法，基于经验和统计的分词方法，基于深度学习的分词方法**。（本实验未实现基于深度学习的分词方法）

基于字符串匹配的分词算法，又叫基于机械匹配的分词方法，基本思想是先构建一个充分大的词典，利用字符串匹配的方式在词典中自动查找分词。目前存在的算法有：正向最大匹配算法（FMM），逆向最大匹配算法（BMM），双向最大匹配算法，最短路径分词法。

基于经验和统计的分词方法，在机械匹配的基础上引入了概率统计模型的数学知识，基本思想是从大量的已经分词的文本中学习到词语切分的规律（训练），从而实现对未知文本的切分。随着大规模语料库的建立，基于统计的中文分词方法渐渐代替了传统的机械分词方法。主要应用模型有：N **元文法模型** (N-gram) , **隐马尔可夫模型，最大熵模型，条件随机场模型**等等。

本实验实习的功能如下：

- **词典构建**
- **正反向最大分词算法实现**
- **正反向最大分词算法性能优化**
- **基于动态规划的** 1-**元文法，**2-**元文法**

- **基于 HMM 的未登录词识别**

3 实验说明

对本实验各文件的功能进行详细说明，方便老师助教阅读检查。

3.1 项目目录

实验完整目录如下图所示，其中 io_file 文件夹里面是输入输出文件,lab1code 文件夹里面是所有的代码文件。

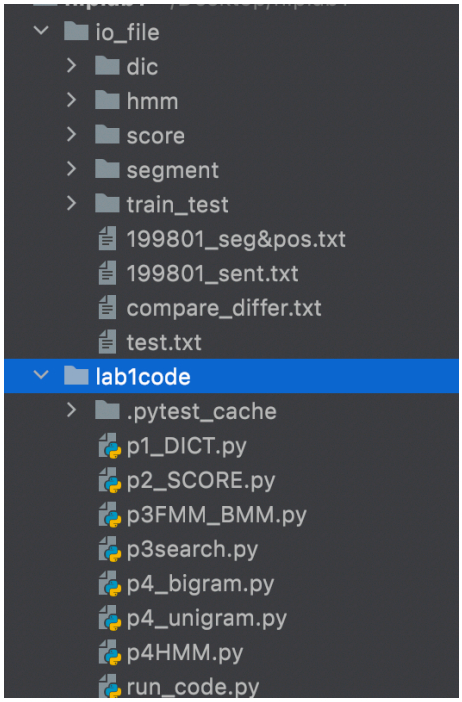

图 1: 项目目录说明

io_file 文件夹存放了所有的产生文件或依赖文件，二级目录中有 5 个文件夹，分别为 dic、hmm、score、train_test 和 segment 文件夹以及 4 个文本文件，分别为 199801_segpos.txt、19980_sent.txt、compare_differ.txt 和 test.txt 文本文件，说明如下:

- dic 文件夹存放了根据标准训练集产生的用于一元文法和二元文法的依赖词典
- hmm 文件夹存放了 hmm 模型的训练参数、整个系统的标准训练集、标准测试集以及和测试及对应的标准答案集
- score 文件夹存放第三部分生成的评测分词效果的文本，后来也存放所有分词评测效果的输出文本
- segment 文件夹存放了所有的分词结果文件，包括实验第 2 和 4 部分产生的 FMM 和 BMM 机械匹配分词结果，以及基于 MM 的一元文法分词结果、基于 MM 的二元文法分词结果、基于 HMM 的分词结果
- train_test 文件夹存放划分后的训练集和测试集，测试集人工分词的标准结构，还有根据标准训练集产生的用于机械匹配分词的词典
- 199801_segpos.txt 为标准的文本，所有的训练文本和测试文本对应的标准答案文本均来源于此文件
- 199801_sent.txt 为标准的文本，所有的测试文本均来源于此文件
- compare_differ.txt 为输出两个分词结果文件不同语句的文件
- test.txt 文件为去掉词性标注之后的测试集的人工分词文件

lab1code 文件夹存放了实验的所有代码。命名为 px...，px 表示实现的是实验指导书上要求的第 x 部分的具体功能。详细的说明如下:

- p1_DiCT.py 文件存放生成词典的代码
- p2_SCORE.py 文件存放性能测试的代码
- p3FMM_BMM.py 文件存放正反向最大匹配代码的最少实现

- p3serch.py 文件存放机械分词的优化代码，这里我用了 Trie 树和 hash 结构进行了优化
- p4_Unigram.py 文件实现了一元文法分词 + 未登录词识别
- p4_bigram.py 文件实现了二元文法分词 + 未登录词识别
- p4HMM.py 文件实现了基于 HMM 的自动分词分词 + 未登录词识别
- run_code.py 文件为测试代码，有已经写好的测试程序，可以测试实验的所有功能，稍后会进行详细说明

3.2 代码测试

为了方便检查运行实验，我已经将实验的几个模块的运行接口全部放入了一个新的 py 可执行文件，该文件的文件目录为 lab1_-code/run_code.py。教师或助教只需要取消注释该文件相关的模块测试行即可。

```
if __name__ == '__main__':
    # test_p1()
    # test_p2()
    # test_p3code()
    # test_p4_unigram()
    # test_p4_bigram()
    # test_p4_hmm()
```

图 2: 项目复查说明

3.3 训练测试文件

实验训练数据和测试数据分别为 199801_segpos.txt 和 199801_sent.txt 文本文件，编码方式为 gbk。

训练数据规模和测试数据规模总体和是固定的，为 23031 行分词文件。训练集大小可以在项目文件 lab1code/p1_DICT.py 模块中通过 createdict 函数进行设置，生成的文件均为 utf-8 编码。当 K=10 时 (默认)，表示训练集为 9/10，测试集为剩余的 1/10，生成的文件方放到 train_test 文件夹中。

训练集和测试集的划分相对比较随机。划分方式为取模划分，即若模 10 余 0，则加入测试集，否则加入作为训练集。虽然模运算的分母是固定的，但是这样划分是可以保证良好的随机性的。因为给定的 199801_segpos.txt 和 199801_sent.txt 的内容在相应的行数分配上并不具有规律性，因此这种简单的方式可以做到良好的随机性。另外，为了简化测试集，删除测试集中的空行。

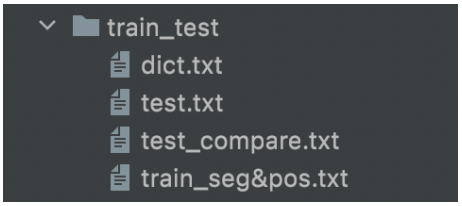

图 3: 训练测试集

4 实验代码分析和结果分析

4.1 词典构建

首先我们需要根据测试集来构建词典，以备机械分词算法使用，但是代码实现的过程中需要注意 2 点:

- 选择性。不是所有的词都需要加入词典的，一方面，词典的容量越大，后续查找速度越慢，不利于后续性能优化，另一方面，有一些词的实际意义并不太大，比如量词，每句话之前的日期这类词语，重复出现的概率太低，加入词典的意义不大。基于这些考虑，词典选择不收录量词。词数由 70217 减少到了 48706，词典容量减少 30%。
- 词组选择。有一些词组，本身也是有多个词构成的，为了统一标准，带 [] 的词组一律忽略不计
- 有序性。在创建词典时，需要对词进行排

序，这样可以进一步优化后续的 4.3 节的查找性能，

- 任务性。针对不同的任务，需要构建不同的词典，比如针对针对机械分词词典，词典中只要有词即可，但是针对后续的 N-元文法，需要构建概率词典，即统计词出现的频数，针对二元文法还需要统计词对出现的频数等等，不同的生成方法会在 4.4，4.5，4.6 节分别说明。

```
for line in lines:
    for word in line.split():
        if '/m' in word:  # 不考虑将将量词加入
            continue
        word = word[1 if word[0] == '[' else 0:word.index('/')]
        word_set.add(word)  # 将词加入词典
        max_len = len(word) if len(word) > max_len else max_len
word_list = list(word_set)
word_list.sort()  # 对此列表进行排序
dic_file.write('\n'.join(word_list))  # 一个词一行
```

图 4: 词典生成主要逻辑

4.2 正反向最大匹配分词——最少代码

正反向分词的最大匹配实现是基于一个基本思想: 找到一句文本中最大词长以下的尽可能长的词序列，即长词优先原则，匹配结果就是分词结果。

值得注意的是分词的依赖词典来源于 199801_segpos.txt 标准文件的 90%(通过模 10 计算得到)，分词的测试文件来源于 199801_sent.txt 文件的 10 由于并未将量词加入词典，因此机械匹配分词的不足需要额外的算法实现量词识别。具体操作如下:

出于最少代码量的要求，分词算法的复杂度很高。将整篇测试文本的 1/10 运行结束需要 25 分左右，运行方式为: 取消 lab1code/run_code.py 文件主函数中的 test_p2code() 的注释，运行该程序即可。分词结果保存在 io_file/segment 文件下的 seg_FMM.txt 和 seg_BMM.txt 中 (测试集为 1/10 量的标准分词文本)

- 最少代码。利用 python 里面的 list 数据结构 + 遍历查找即可。
- 量词处理。由于词典中并未加入量词，所以需要对 FMM/BMM 得到的进行进行后处理。即处理分词结果中连续的量词、字母、英文符号组合，并将他们视为一个词。

在利用 FMM/BMM 得到了分词结之后 test_p2code() 会利用 p2_SOCRE.py 计算性能，即计算分词的准确率、召回率、F 值。公式如下:

- $precision = \frac{\text{正确分词数}}{\text{标准分词数}}$
- $recall = \frac{\text{正确分词数}}{\text{分词总数}}$
- $F = \frac{precision \times recall}{precision + recall}$

通过逐行比较标准答案文本和分词文本得到分词效果的评价。主要是统计标准答案的总词数、分词的总词数以及正确分词数。

该部分代码实现了复用，实验后期的分词文本可以利用此代码进行效果评价得到的分数保存在 io_file/score 文件下，分别命名为 score_FMM.txt,score_BMM.txt。

FMM 结果如下:

- precision:0.8820533174571674
- recall:0.9320999161152018
- F:0.9320999161152018

BMM 结果如下:

- precision:0.8833657766187976
- recall:0.9340494609625004
- F:0.9340494609625004

正反向分词匹配结果深入分析如下:

- 分词性能差异，从不同规模和不同方式生成的的随机训练集和对应互补测试集看 FMM 和 BMM 的分词效果存在一定的性能差异的，且均是 BMM 效果较好

- 分词精度差异 (基于内容分析)，汉语独特的构词特点使得逆向最大匹配分词的性能更优。直观来看，对于汉语的一个词语来说，在词后加一个字仍能构成新词概率较高，而在一个词前加上一个字还可构成新词概率较低，这就使得逆向最大匹配得到的最长词为准确分词结果的可能性较高。例如，“当时间的脚步”，正向最大匹配会因为在寻找最长词的时候失去本意选择“当时”作为第一个词，划分错误，而逆向最大匹配可正常分词

- 汉语特点引起的机械分词差异，汉语的构词法和侧重词语偏句后的特点决定了一般 BMM 分词效果较好。无论是 FMM 还是 BMM 都是对已知词语的一个判断。但是在进行已知词语判断的过程中会出现“伪最长词”的情况，即匹配的一个失去原意的最长词会导致后面的词语因为匹配不到对应词语而出现单字或者说出现错误词。但汉语一句话一般侧重的词语在句子的后面 (有别于英语)，如“教育部门”落脚点在“部门”，而“教育”是形容该机构的一个属性，“教育部”是由“教育部门”构词形成的相应机构。汉语的构词法使得两个词语可以有相同的前缀且表意相近，从而使得正向匹配分词可能落入这样构词特点的“陷阱”

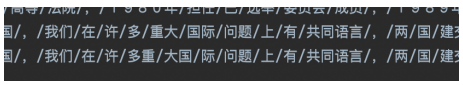

图 5: 上为 BMM 正确结果，下为 FMM 错误结果：重大/国际

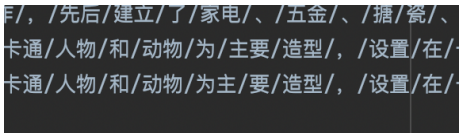

图 6: 上为 BMM 正确结果，下为 FMM 错误结果：为/主要

4.3 正反向最大匹配分词——速度优化

实验第 4 部分，代码实现在两个方面超过了实验的要求: 一，不仅对 FMM 进行了分词性能优化也对 BMM 进行了相应的性能优化。二分词时间的优化结果远超底线要求，时间优化了将近 2500 倍 (对测试集进行分词的时间由 1 个小时缩短到 0.3 ~ 0.5 秒钟左右);

时间优化策略: 减少搜索次数、匹配次数。

减少搜索次数，实验第 2 部分中，使用数据结构为基本 list，且查找算法几近为 0，只使用最简单的 if in 语句判断一个词是否在词典中。现在使用 Trie 结构保存所有的词的每个字; 同时手写 HASH 数据结构保存一个 Trie 树节点的所有子节点。

Trie 树节点的数据结构为，其中 char 代表的是当前结点所存储的字符；is_word 判断当前字符是否为某个词的最后一个字，是为 true，默认为 false；child_list 代表子节点的集合，now_words 代表子节点的数目：

```
def __init__(self,is_word=False,char='',init_size=700):
    self.is_word=is_word #表示这是不是一个词最后一个字
    self.char=char
    self.wordnum=0
    self.list_len=init_size
    self.list=[None]*init_size
```

图 7: trie 结点属性

Trie 树节点的方法如下图所示，包含的主要方法有：初始化结点，计算 hash 值，创建子节点，返回节点：

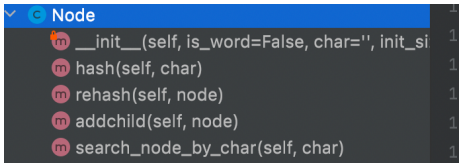

图 8: trie 结点方法

查找一个子节点的复杂度降低为 O(1)，在 Trie 树中查找一个最长词的时间复杂度为 O(Max_Len)，一般情况下词长较小，大约为 3，因此整体查找复杂度很低;

减少匹配次数，传统机械匹配分词天然缺

输入：训练文本，测试集。 输出：FMM 测试集分词结果。
start 构建在线前缀词典：生成 trie 树，返回 root。 读取测试集待分词行。 search trie 树 root ， if node not None, continue until node.is_word=True, 得到最大长度词 if node is None ，则该字单独分词 until 测试文件无剩余行，end

表 1: FMM 优化伪代码

输入：训练文本，测试集。 输出：BMM 测试集分词结果。
start 构建在线后缀词典：生成 trie 树，返回 root。 读取测试集待分词行，翻转。 search trie 树 root ， if node not None, continue until node.is_word=True, 得到最大长度词 if node is None ，则该字单独分词 until 测试文件无剩余行，end

表 2: BMM 优化伪代码

点是匹配次数过多。对于一个 26 最长词的分词系统，得到一个分词平均需匹配 23 次 (假设在一个待分词系统中平均词长为 3)，这极大地提高了时间复杂度。Trie 树有明显优势可解决过度匹配问题:Trie 树可保存以一个词开头的所有词，要匹配一个词，只需通过查找一棵 Trie 树的前缀节点即可，匹配次数为 3 次左右 (同样假设在一个待分词文本系统中平均词长为 3)

FMM 查找优化算法：算法伪代码如表 1 所示。优化后的代码维护一棵 Trie 树，树的每一个节点保存的是一个字，其子节点保存的是以父节点保存字为前缀的下一个字，同时父节点的子节点列表采用了 Hash 思想以及线性冲突探测技术 (加 1 保存和加 1 探测)。对于待分词行，若首字在 Trie 树中，则说明可能存在以该字为前缀的词，继续判断下一个字是否在该节点的子结点中; 若不在子结点中，则保存最长的一个终结词 (通过节点的属性 is_word 判断)，并将此行前面的分词去掉，继续执行上述分词过程，直到该行长度为 0，说明该行分词结束。在该过程中查找一个字是否存在于一个结点的子结点中使用的是 Hash 思想，即将字与字所在列表中的下标一一对应起来，实现 O(1) 查找，若发生冲突，则执行线性 +1 再探测。实验装载因子为 2/3。

BMM 查找优化算法：BMM 优化算法伪代码如表 2 所示，与 FMM 优化算法逻辑相同，只有 2 点需要注意的地方，1、trie 树为后缀树，对于一个词从后往前索引。2、将待分词语句翻转，即可用 FMM 的代码逻辑来优化搜索。

- 测试代码：取消 lab1code/run_code.py 文件主函数中的 test_p3code() 的注释，运行该程序即可。
- 实验结果：FMM:0.3857998847961426s, BMM:0.3751046657562256, 算法性能优化上千倍。

  优化分词速度关键，优化后分词系统的的结构特点是是整体 Trie 树储存节点，内部子节点使用 Hash 思想对 List 进行操作。分词速度优化的关键是减少查找词是否在词典中的搜索次数，减少最长匹配实现过程中的匹配次数。Trie 前缀和后缀树分别减少词前缀和后缀匹配的次数、Hash 思想减少词查找次数。但是现在系统速度仍有两个限制: 一是更好的 Hash 函数，二是更符合每个结点的 List 列表初始大小 (因为当列表存储的子节点满的时候需要重新对所有节点进行 Hash 处理，这部分操作对于时间消耗很大，因此要避免重哈希的次数)，限于实验时间较为短暂，我未能将这两部分

完全解决，但是仍给出未来解决方案: 对于好的散列函数，可以先对词频进行一个统计，然后采用哈夫曼编码的形式将每个字进行编码得到一个 0、1 序列并作为该字 hash 值，将其保存到本地文件中并于下次直接使用 (因为这部分时间不算在分词时间内，而是算在预处理时间内); 对于更符合每个结点的 List 列表的初始大小，这个仍然是要基于对每个字的后缀字进行统计，然后根据统计结果对每个字生成特定的子节点列表长度即可。

4.4 基于统计的一元文法实现

一元文法的假设是一个词与其前后的词是无关的、相互独立的。一元文法等价于最大频率分词，即把切分路径上每一个词的词频相乘得到该切分路径的概率，并把词频的负对数理解成“代价”，这种方法也可以理解为最少分词法的一种扩充。同时由于一元文法的分词正确率可达到 92%，且简便易行，效果一般好于基于词表的方法。

- 测试代码：取消 lab1code/run_code.py 文件主函数中的 test_p4_unigram () 的注释，运行该程序即可。
- 生成在线词典结构,读取离线词典中的每一个词及其对应的词频，保存到 Word_freg 数据结构中，并更新总词数。
- 一元文法算法思想：一元文法的公式为

  $p(w_1, w_2, \ldots\ldots w_n) = p(w_1)p(w_2)\ldots\ldots p(w_n)$

  一元文法做了一个强假设：每个词的出现与上下文无关，只与自己有关，即各个词之间是相互独立的。一元文法的优化目标是，找到一个分词的结果，使得每句话的联合概率值，即上式的概率值最大。

  具体的算法为：根据其是否词频大于 0 计算其 DAG，并根据填表算法动态规划求解该行文本的最大概率路径 (符合动态规划的条件：重复子问题和最优子结构)，最后从一行末尾进行路径还原得到最可能的分词结果。值得注意的是，算法需要采用平处理，因为计算概率时，为了防止概率相乘过小而产生下溢的问题，需要将概率取对数处理，这样就要确保排除 0 概率事件，即将 0 概率事件 +1 平滑处理，作为最终的概率对数结果

```
nslen(line)
routes={n:(0,0)}
log_total=log(wordnum)
for idx in range(n-1,-1,-1):
    route[idx]=max(( (log(word_freg.get(line[idx:x],0) or 1)-log_total+route[x][0],x) for x in dag[idx])
return route
```

图 9: 动态规划主要代码

- 实验结果：准确率很低，为 69% 左右 (因为每行文本前面的数字标识)。一元文法是最大概率分词，依赖于词典中是否存在某词语，若词典中不存在该词语，那么一元文法较难实现对于该词语的识别，即一元文法对于未登录词的解决能力是很差的。

```
precision:0.6842443619164605
recall:0.925893994469817
F:0.7869357846615946
```

图 10: unigram 分词结果

4.5 基于统计的二元文法实现

二元文法的公式为：

$p(w_0, \ldots\ldots w_n) = p(w_1|w_0)\ldots\ldots p(w_n|w_{n-1})$

二元文法 (一阶马尔可夫) 的假设相对弱化了一元文法的强假设，一元文法的假设是一个词出现的概率与其余词无关，而二元文法假设的是一个词语的出现仅依赖于前一个词语，使得该模型从原理上讲更加的合理化。利用二元文法可以利用更多上下文信息，从而进一步提高性能。

- 测试代码：取消 lab1code/run_code.py

文件主函数中的 test_p4_bigram () 的注释，运行该程序即可。

- 生成在线词典结构：读取离线词典中的每一个词及其对应的词频，保存到 Word_freg 数据结构中，并更新总词数。除此之外，由于二元文法还需要考虑词对出现的频率，需要额外定一个一个字典数据结构 word_dict ，保存 key= 词、value={key= 上一个词、value= 两个词对出现的频率}。另外为了保持概率值和为 1，需要在每句话前加 BOS 作为开始标志，每句话之后加 EOS/作为结束标志。主要函数逻辑如下图所示：

```
seg_line.append('EOS/ ')
seg_line.insert(0, 'BOS')
for idx,word in enumerate(seg_line):
    if word=='BOS':
        word_freg[word] +=1
        continue #第一个词不计入二元词典
    else:
        word=word[1 if word[0]=='[' else 0:word.index('/')]
        seg_line[idx]=word
        #加入词频表
        if word in word_freg: word_freg[word]+=1
        else: word_freg[word]=1
        #加入二元词频表
        if word not in word_dict.keys():
            word_dict[word]={}
        if seg_line[idx-1] not in word_dict[word]:
            word_dict[word][seg_line[idx-1]]=0
        word_dict[word][seg_line[idx-1]] +=1
```

图 11: bigram 词典生成主要代码

- 二元文法算法思想，首先读取离线词典建立在线词典结构数据结构，初始化 words_dic，用于保存所有词对应的前缀词及其对应的词频；根据建立的数据结构，不断处理测试文本的每一行: 首先将字符串’ BOS’ 和’ EOS’ 分别加入到行的开头和末尾，计算得到此行的 DAG，随即对改行进行最大概率分词，计算概率最大路径。值得注意的是计算概率最大路径的时候需要综合考虑一个词的在某处出现的概率最大仍然需要考虑该词的前词情况，因为二元文法的两个相邻词是条件相关的。
- 动态规划最大概率求解，二元文法在条件相关的最大概率路径求解从前向后依据前词及其对应的组合概率生成每一个字可能的最大概率分词结果，同时需要保存该词对应的前词及其概率结果。不断向后对每个字组成的词进行填表规划，最终生成最大概率结果。
- 平滑处理，二元文法在概率平滑处理上采用对数概率的处理方式，概率结果计算方式变为加法操作，对于可能出现条件概率出现 0 概率的情况，特意对全部词的条件概率求解，分母整体加上总词数，分子词频加 1。
- 实验结果：通过对比一元文法和二元文法的分词结果，我们发现一元文法和二元文法的准确率很低，可能的原因是因为每行文本前面的数字标识，导致分词数增加。二元文法是条件最大概率分词，依赖于词典中是否存在某些词对，若词典中不存在该词语或者词对，那么二元文法较难实现准确预测，即在相同的训练集的情况下，二元文法对于未登录词或者词对的解决能力是更差的。

```
precision:0.6481109548994667
recall:0.9023052785285985
F:0.7543702433829452
```

图 12: bigram 分词结果

4.6 HMM 模型

HMM(隐马尔可夫模型) 是一个隐状态序列统计模型，常用来进行未登录词的识别。HMM 算法对于未登录词的识别具有优势。HMM 算法通过对训练集文本的状态’ BMES’ 进行标注和统计得到 HMM 模型参数 $\lambda = (\pi, A, B)$ 三元组，并根据训练得到的参数利用 Viterbi(维特比) 算法，本质上还是动态规划的思想，对测试集的每一行寻找最大概率路径，即寻找隐状态序列的划分。

- 测试代码：取消 lab1code/run_code.py 文件主函数中的 test_p4_hmm () 的注释，运行该程序即可。

- 生成在线词典结构，HMM 模型使用一元文法的词典生成在线数据结构 word_set 集合，用于判断一个词是否在词典中。

- HMM 算法思想。HMM 模型的初始化参数为：初始状态概率分布 、状态转移概率分布 A 以及观测概率分布 B 确定，为了方便表达，把 A, B, π 用表示，即：$\lambda = (A, B, \pi)$

  状态集合 S：{B,M,E,S}，N=4

  π：初始状态概率分布，如 {B:-0.26268660809250016, E:-3.14e+100, M:-3.14e+100, S:-1.4652633398537678}

  A：状态转移概率分布，N*N 的矩阵如：{'S': {'S': 747969.0, 'E': 0.0, 'M': 0.0, 'B': 563988}, 'E': {'S': 737404.0, 'E': 0.0, 'M': 0.0, 'B': 651128.0}

  B：观测概率分布或者称为发射概率分布，隐藏状态到观测状态的混淆矩阵，N×M，如 {'S': {' 否': 10.0, ' 昔': 25.0, ' 直': 238.0, ' 六': 1004.0, ' 殖': 17.0, ' 仗': 36.0, ' 挪': 15.0, ' 朗': 151.0}}

  O：输出序列

  利用 Viterbi 算法输出该行测试文本最大可能的分词隐藏状态序列 O; 然后对标注结果按照 B、E 和 S 界定一个词的始末位置，从而对整个文本进行状态还原，得到分词结果

- 实验结果：受限于训练语料库规模较小、训练样本的词汇筛选方式无法达到最优，导致使用训练的参数得到的分词结果性能不是特别的理想，准确率为 75% 左右，召回率为 78% 左右，运行时不依赖输入标准词典文件，仅仅依靠训练好的模型参数便可实现相对较好的分词效果。

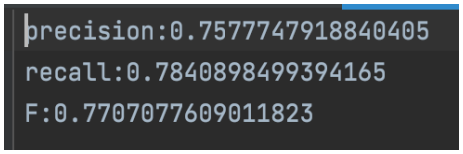

图 13: hmm 分词结果

除此之外，hmm 模型尤其是对于未登录词识别较为准确，我们可以看到分词结果比较好的将 n 元文法无法分离的数字等量词进行了识别。

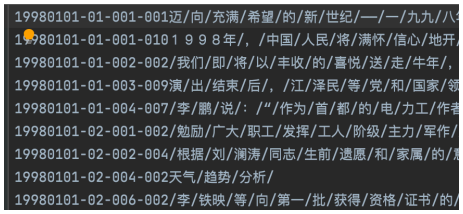

图 14: hmm 未登录词识别

4.7 系统优化：n 元文法 + 未登录词识别

n 元文法是基于词典进行分词的，准确性较高，但是对于未登录词识别较差；HMM 模型是基于统计参数进行分词的，对于未登录词有一定的识别能力，但是对于一些已有的词识别准确性较低，将两者综合一下，希望实现系统优化。

- 测试代码，取消 lab1code/run_code.py 文件主函数中的 test_p4_bigram ( ) test_p4_Unigram () 的注释，未登录词相关实现已经加入，运行该程序即可。

- 优化实现算法思想，可在 n 元文法进行搜索分词空间的最大概率路径时，利用 HMM 思想识别出未登录词，并将其作为分词空间的一个新的分词路径，适当增加其权重值; 最后对新的解空间的路径进行统一的最大化概率求解，得到含有未登录词的新的分词路径。

  除此之外，为了更好的提升性能，对训练集和测试集，在代码运行的时候均不考

虑前置的结构化时间，如：“19980101-01-002-004”，只在自动化分词之后再加入到分词结果中。

- 实验结果。在训练集为标准的测试文件的 9/10，测试集为其互补的 1/10 时，分词结果均相比于原本单一的思想有了较大提升，最高可达到 94%

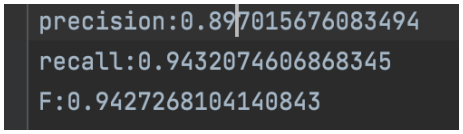

图 15: 1 元文法优化结果

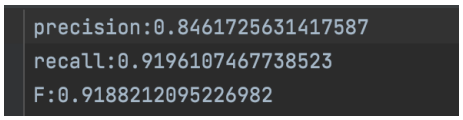

图 16: 2 元文法优化结果

5 实验结论与展望

5.1 实验总结

本实验通过中文分词这一固定任务，引发了一系列的讨论和探究。

- 从最基础的中文分词词典开始构建，构建词典的时候不是简单的对词语进行罗列，还需要进行实用性考量，真的不同的任务，选择最合适、最高效的词典构建方式。例如：不将结构化的量词、字母、符号等放到词典中，而采用基于规则的处理等方式进行划分。构建词典的任务繁复琐碎，但却是不同的分词算法需要优先考虑的，好的词典构建可以为之后的自动分词提供便利。

- 本实验要求实现的分词的算法为正反向最大匹配算法，最少代码实现和优化；基于统计的 1-元文法，2-元文法；识别未登录词的 HMM 算法。三种算法各有侧重，正反向最大匹配算法是分词的基础，也是最简单的算法，但是简单不意味着准确率低，当词典足够大的情况下，FMM 和 BMM 可以达到很好的效果。基于统计的 n 元文法算法，引入了概率论的知识，加入了动态规划的算法思想，更复杂，但是理论基础比机械分词要好。上述两种算法共同点是无法识别未登录词，而 HMM 分词模型可以很好的识别未登录词，但是由于是基于参数的自动分词模型，对于已登录词的准确性可能会下降。最后的系统优化环节为希望可以将 hmm 与 n 元文法模型融合起来，兼具他们的优点，实现更好的分词性能。

5.2 未来展望

- 虽然在本实验中,最终的算法在开放测试集中达到了 94.3% 左右的分词性能，但是仍然存在着优化的空间: 模型平滑处理、词典中词的进一步筛选。

- 对于二元文法的参数平滑处理处理的不是很好，二元文法的性能提升空间有限，可能和参数平滑不够充分有关，本代码采用的是 1-平滑方法，希望接下来可以尝试多种参数平衡方法。

- 词典中词的进一步筛选,目前词典的做法是将训练集中所有词都加入词典,因为这样可以获得 94.3% 左右的分词效果。未来可以通过相应的特点适当的甄选加入词典的词，并通过对相关分词语句进行附加的处理 (如连续的 ASCII 字符、连续的全角英文和数字字符一般可以作为一个分词结果) 实现更好的模型分词结果 (体现在减小过拟合的情况)

6 参考文献

[1] 宗成庆. 统计自然语言处理. 清华大学出版社：第二版

[2] 云时之间.NLP 入门之 N 元语法模型：https://zhuanlan.zhihu.com/p/29824784, 2017-09-30

[3] 果 7. 基于 N-gram 的双向最大匹配中文分词.csdn:2013-12-24

[4] CQUPT-Wan. 基于 HMM 的中文分词.csdn：2018-05-19

[5] 如何通俗地讲解 viterbi 算法? ：https://www.zhihu.com/question/20136144