汉语分词系统

哈尔滨工业大学
张景润_1172510217

摘要

本次实验目的是对汉语自动分词技术有一个全面的了解，包括从词典的建立、分词算法的实现、分词结果性能评价和优化等环节。本次实验用到了以下知识：基本编程能力，如文件处理和数据统计能力、相关的查找算法和数据结构实现能力，如双 trie 树和 Hash 列表、语料库的相关知识、正反向最大匹配分词算法、分词性能评价的常用指标，如准确率和召回率、N 元语言模型的相关知识、马尔可夫模型和隐马尔可夫模型的相关知识。

1 绪论

汉语中词是最小可独立活动的有意义的语言成分，汉语以字为单位，有别于西方语言，词与词之间没有空格之类的标志指示词的边界，因此需要汉语言分词系统。而同时分词问题为中文文本处理的基础性工作，分词好坏对后面的中文信息处理起关键作用

目前中文分词存在以下难点：**分词规范**、**词定义不明确**、**歧义切分**、**交集型切分问题**和**多义组合型切分歧义**等；

同时**未登录词识别问题**也对中文分词提出了更高的要求。对于大规模真实文本来说，已有的词表中没有收录的词，和已有训练语料中未曾出现过的词对分词的精度的影响远超歧义切分。

2 相关工作

实验相关的理论基础中文分词(Chinese Word Segmentation) 指的是将一个汉字序列切分成一个一个单独的词。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

现有的分词算法可分为三大类：**基于字符串匹配**、**基于理解**和**基于统计的分词方法**（本实验未实现基于理解的分词方法）；

基于字符串匹配的分词方法：又叫做机械分词方法，它是按照一定策略将待分析的汉字串与一个“充分大的”机器词典中的词条进行配，若在词典中找到某个字符串，则匹配成功（识别出一个词）：目前存在**正向最大匹配法**、**逆向最大匹配法**、**最少切分**和**双向最大匹配法**分词方法

基于统计的分词方法：给出大量已经分词的文本，利用统计机器学习模型学习词语切分的规律（称为**训练**），从而实现对未知文本的切分。例如最大概率分词方法和最大熵分词方法等。随着大规模语料库的建立，统计机器学习方法的研究和发展，基于统计的中文分词方法渐渐成为了主流方法。主要统计模型：**N 元文法模型（N-gram）**，**隐马尔可夫模型（Hidden Markov Model ，HMM）**，**最大熵模型（ME）**，**条件随机场模型（Conditional Random Fields，CRF）**等。

实验实现了所有功能，具体内容如下：

- **词典的构建**
- **正反向最大匹配分词的实现和效果分析**
- **基于机械匹配的分词系统的速度优化**
- **基于 MM 的一元、二元文法分词**
- **基于 HMM 的未登录词识别**

3 实验设置

3.1 项目目录说明

包文件下有 io_file 文件夹、lab_code 文件夹、实验报告和 README.md 文件如图 3.1 所示。

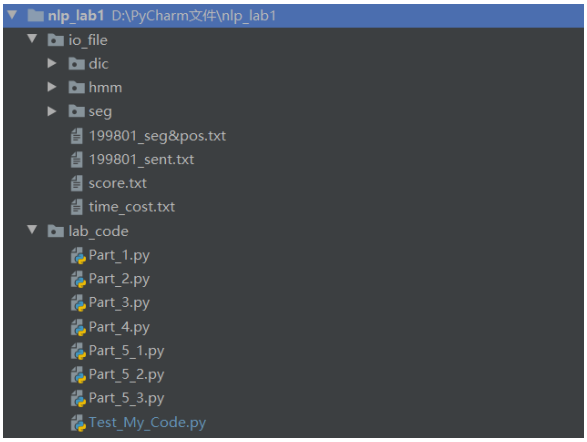

图 3.1：项目目录说明

io_file 文件夹该文件夹存放了所有的产生文件或依赖文件，二级目录中有 3 个文件夹，分别为 dic、hmm 和 seg 文件夹以及 4 个文本文件，分别为 199801_seg&pos.txt、19980_sent.txt、score.txt 和 time_cost.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机械匹配分词、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
- **seg** 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
- **199801_seg&pos.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
- **score.txt** 为第三部分生成的评测分词效果的文本，后来也作为所有分词评测效果的输出文本，输出格式已经规约好，条目清晰
- **time_cost** 文本为第四部分生成的运行时间对比文本，输出条目清晰，对比明显，主要用于对比优化前后的分词时间消耗

lab_code 文件夹存放了实验的所有代码，其中我在原本实验的基础上，实现了基于 Trie 树进行机械匹配分词、基于 HASH 结构的机械匹配分词，基本目录如图 3.2：

- **Part_X.py** 文件为对应的实验部分的代码，值得注意的是，代码之间的耦合关系不强，第五部分的代码耦合关系较强
- **Test_My_Code** 文件为为了方便检查才实现的一个类，具体内容在下一小节介绍。
- **MORE** 文件夹存放了实验第 4 节用其他数据结构实现的分词算法，如基于字典树和基于散列技术的分词算法

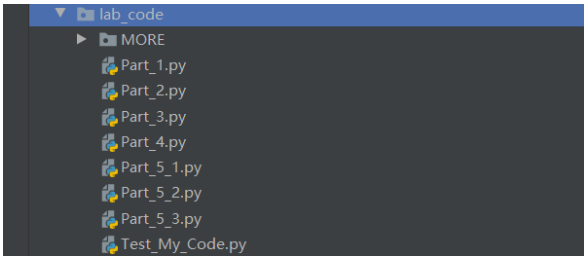

图 3.2：lab_code 文件夹

3.2 实验复查说明

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 **lab_code/Test_My_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。

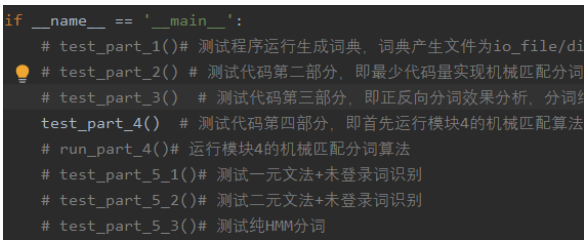

图 3.3：测试实验第四模块

3.3 训练测试相关

实验训练数据和测试数据分别来源于 **199801_seg&pos.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。其中训练集和测试集在内容上是不相同的，互补的（对于标准文本来说）。

训练集大小可以在项目文件 **lab_code/Public.py** 模块中通过设置 K 全局变量的大小进行设置。当 K=10 时（默认），表示训练集为 9/10，测试集为剩余的 1/10。

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_seg&pos.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。

4 实验内容及过程

目前我的实验工作已经全部完成，预计未来将实现参数的进一步训练。

- 实验第 1 部分工作是形成一个词典，目前形成的词典有三个，分别用于不同的实验阶段或者说不同的实验方法
- 实验第 2 部分是通过最少代码量实现机械匹配分词，为此选用了最简单的 Python 的 List 作为在线词典数据结构，并直接进行相应的查找和匹配操作，由于查找次数的时间复杂度为 **O(N)**、匹配次数极大的取决于最长词长 **Max_Len**，该部分时间复杂度很大，运行时间较长，但是我可以保证该份代码的准确性以及分词结果的可复现性
- 实验第 3 部分是对机械匹配分词效果进行分析，该部分代码是可被其他模块复用的，但是仍然实现了一个一次性对正反向最大分词进行测试的程序。基本公式：

$$precision = \frac{正确分词数}{标准分词总数} \quad (1)$$

$$recall = \frac{正确分词数}{分词总数} \quad (2)$$

$$f\_value = \frac{(k^2+1)*precision*recall}{k^2*precision+recall} \quad (3)$$

- 实验第 4 部分实现了基于散列技术和字典树的机械匹配算法，极大的减小了算法的时间复杂度，最终优化程度大约为 3000 倍（即时间由 5 个小时缩短到 7 秒钟左右），值得注意的是为了使数据测试规模更大，仅在优化测试时选择了将 199801_sent.txt 作为测试整个测试集
- 实验第五部分实现了基于 HMM 和一元文法的统计语言模型分词、基于 HMM 和二元文法的统计语言模型分词、基于 HMM 的统计语言模型分词。都对未登录词做了相应的处理，在一元文法算法的训练集为 199801_seg&pos.txt 文件的 9/10 条件下，算法的准确率大约在 90%左右，通过未登录词处理，可以将准确率提高到 94%，二元文法情况类似，最终都可以实现较好的分词性能

4.1 词典的构建-实用性考量

词是词典的基本单位，但是不是全部的词都可以加入词典。基于这样的一个事实：分词词典的构建是要服务于分词任务的，或提高算法运行速度、或提高分词的准确率，我在词典的构建中采用了这样几条策略：

- **选择性**，通过进一步实验测试，我选择**不将量词加入词典、将人名加入词典、不将未出现过的单字词加入词典**（可以提高分词事件和空间性能），**不将重复的词加入词典**（提高空间性能）。不将量词加入词典仍然可以达到和对比情况相近的准确率，而且可以带来时间上的提高和空间上的优化：词数从 70217 减少到 48706，f_value 大约是 90.5%。同时没有将词频加入词典，因为差异性的实现和实用性的要求，词频的加入不会改变实验前 4 部分的分词结果，反而会降低实验时间性能

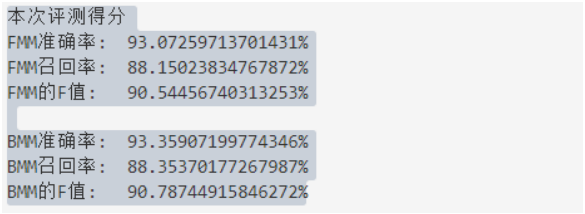

图 4.1.1：筛选加入词典的词得到的测评

- **差异性**，不同的算法或者说分词策略采用不同的词典。具体表现是构建 3 个离线词典分别用于不同的实验阶段，普通的 dic.txt 用于实验前 4 部分，uni_dic.txt 用于一元文法的算法实现，bigram_dic.txt 用于

二元文法的算法实现。差异性词典的好处在于具体实验中不必过分执着于如何利用线下的唯一的词典，而可以将侧重点放在高效的算法实现上

- **易读性**，词典是 utf-8 格式的，字符串形式的，关键信息按行分布的。这样做可以方便实验第 2 部分的最少代码量实现（因为处理离线词典所需要的代码行数更少），同时可以便于通过观察词典中的词结构实时修改词典中词的放置策略，以提高算法性能

- **有序性**，词典的内容是高度排序的，通过简单的词典中词的排序，可以进一步实现复杂度更低的查找匹配算法，便于实验第 4 部分的进一步处理

- **有效性**，离线词典中的词全部来源于标准训练集的一部分，同时词典中的词是唯一的，这样可以极大的减小词典的在线数据结构的空间复杂度

```
with open(dic_path, 'w', encoding='utf-8') as dic_file:
    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 4.1.2：生成词典的主要代码逻辑

```
同
同一
同一个
同一天
同一性
同业
同义词
同义语
```

图 4.1.3：生成词典的样例

4.2 正反向最大匹配分词-最少代码量

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_seg&pos.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%（刚好与训练文件互补）。

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：

- **测试代码**，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 **lab_code/Test_My_Code.py** 文件主函数中的 **test_part_2()**的注释，并运行此程序即可

```
if __name__ == '__main__':
    # test_part_1()  # 测试程序运行生成词典，词典产
    test_part_2()  # 测试代码第二部分，即最少代码量
    # test_part_3()  # 测试代码第三部分，即正反向分
    # test_part_4()  # 测试代码第四部分，即首先运行
    # run_part_4()# 运行模块4的机械匹配分词算法
    # test_part_5_1()  # 测试一元文法+未登录词识别
    # test_part_5_2()# 测试二元文法+未登录词识别
    # test_part_5_3()  # 测试纯HMM分词
```

图 4.2.1：测试实验第 2 部分

- **FMM 代码逻辑**，通过读取词典文件生成在线词典的数据结构，开始分词；将测试文本的每行作为循环内容，不断将该行减去一个得到的分词，得到分词的条件是该词在词典中存在且该词后面没有字可以和该词组合形成一个新的词典中的词，直到该行完全分词，继续下一行。

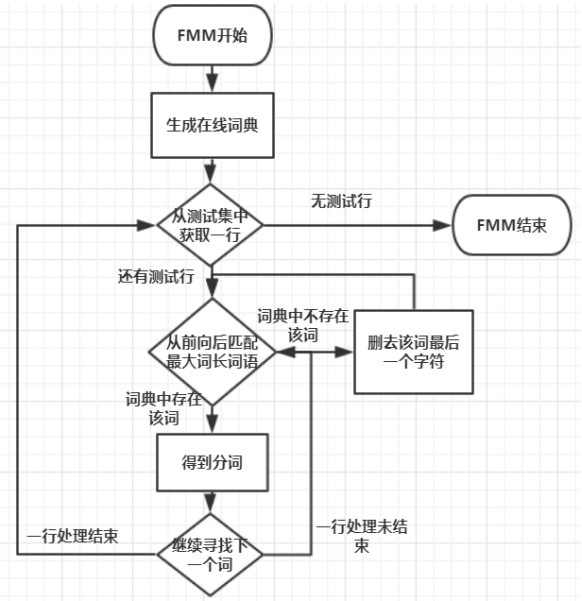

图 4.2.2：FMM 流程图

- **BMM 代码逻辑**，与 FMM 不同的是，BMM 采用从一行测试文本的末尾开始分词，找到最长的在词典中的词，直到该行分词结束，流程图如图 4.6 所示：

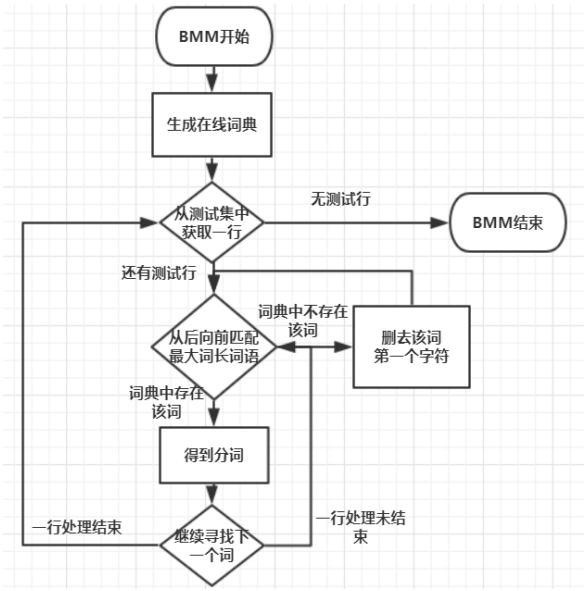

图 4.2.3：BMM 流程图

- **做到最少代码量**，我的策略是：**Python 语言+List 数据结构+遍历查找**算法实现，最终实验第 1 部分的代码行数为 **51** 行

- **数据结构与查找算法**，全局变量 **Words** 列表保存离线词典中的所有词；并通过 python 自带的关键字 **in** 判断一个词是否在词典中

```
with open(fmm_path, 'w', encoding='utf-8') as fmm_file:
    for line in txt_lines:
        seg_line, line = '', line[:len(line) - 1]  # 去掉最后一
        while len(line) > 0:
            try_word = line[0:len(line) if len(line) < Max_Len
            while try_word not in Words:
                if len(try_word) == 1:  # 字串长度为1，跳出循环
                    break
```

图 4.2.4：相应的代码逻辑

- **量词处理**，为了降低空间复杂度，将量词从离线词典中剔除，因此需要对机械分词得到的结果进行单独的处理，以提高分词的性能。处理逻辑中判断连续的量词、字母、英文符号组合可以视为一个词。处理结果是分词性能由 **80%**左右提升到 **91%**左右，如图所示

```
BMM准确率：  93.179253931316 55%
BMM召回率：  69.895992435813 51%
BMM的F值：   79.875475561701 18%

本次评测得分
FMM准确率：  93.072597137014 31%
FMM召回率：  88.150238347678 72%
FMM的F值：   90.544567403132 53%
```

图 4.2.5：处理前后对比

4.3 正反向最大匹配分词-效果分析

该部分代码的基础是分词效果评价**公式(1)、公式(2)和公式(3)**，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。代码文件为 **Part_3.py**，复用部分为 **Public.py** 文件中的 **calc** 方法。

该部分代码实现的是对测试集的分词结果进行评分，分词结果保存在 **io_file/seg** 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）

- **测试代码**，为教师和助教检查代码方便，专门提供了检查接口。运行方式：取消 **lab_code/Test_My_Code.py** 文件主函数中的 **test_part_3()**的注释，并运行此程序即可

```
if __name__ == '__main__':
    # test_part_1()  # 测试程序运行生成词典，词典产
    # test_part_2()  # 测试代码第二部分，即最少代码
    test_part_3()  # 测试代码第三部分，即正反向分词
    # test_part_4()  # 测试代码第四部分，即首先运行
    # run_part_4()# 运行模块4的机械匹配分词算法
    # test_part_5_1()  # 测试一元文法+未登录词识别
    # test_part_5_2()# 测试二元文法+未登录词识别
    # test_part_5_3()  # 测试纯HMM分词
```

图 4.3.1：测试第 3 部分代码

- **评价代码逻辑**，分词文本与标准答案文本进行**预处理**去除所有的空行；逐行进行比对分析，判断分词文本的一行与标准答案文本的一行**相同词数**，而比较相同词数的方法主要是滑动窗口法：即将标准答案一行的每个词和分词文本一行的每个词作为窗口，若上下两个有关于

字数的下标重合，则说明分词正确，否则依次滑动窗口下标，不断对比，直到末尾字符；最终得到分词文本标准答案文本的**总词数**和**正确词数**，利用**公式(1)**计算准确率、利用**公式(2)**计算召回率、利用**公式(3)**计算 F 值

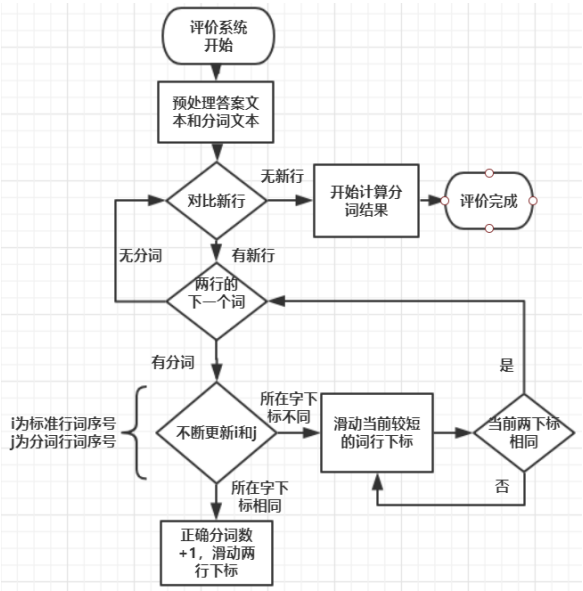

图 4.3.2：评价代码流程图

- **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（在多次测试中无例外）

```
本次评测得分
FMM准确率:   93.20921490042564%
FMM召回率:   88.20394843224014%
FMM的F值:    90.6375329491469%

BMM准确率:   93.40416938515555%
BMM召回率:   88.33519421754717%
BMM的F值:    90.79899125655004%

本次评测得分
FMM准确率:   93.07259713701431%
FMM召回率:   88.15023834767872%
FMM的F值:    90.54456740313253%

BMM准确率:   93.35907199774346%
BMM召回率:   88.35370177267987%
BMM的F值:    90.78744915846272%
```

图 4.3.3：分词性能差异

- **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”

```
19980131-04-001-002/ 吴/ 子长/
19980131-04-002-004/ 这种/ 绚烂/ 与/ 神妙/ 是/ 无法/ 用语/ 言/ 表达/ 的/
19980131-04-003-003/ 当时/ 间/ 的/ 脚步/
19980131-04-003-013/ 你/ 将/ 新/ 的/ 春天/ 一夜间/
19980131-04-003-023/ 你/ 就/ 说/
```

图 4.3.4：FMM 分词错误示例

```
19980131-04-001-002/ 吴/ 子长/
19980131-04-002-004/ 这种/ 绚烂/ 与/ 神妙/ 是/ 无法/ 用/ 语言/ 表达/ 的/
19980131-04-003-003/ 当/ 时间/ 的/ 脚步/
19980131-04-003-013/ 你/ 将/ 新/ 的/ 春天/ 一夜间/
19980131-04-003-023/ 你/ 就/ 说/
```

图 4.3.5：BMM 分词正确示例

4.4 基于机械匹配分词系统的速度优化

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（对完整的测试集进行分词的时间由 5 个小时缩短到 8 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，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。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；

```
class Node:

    def __init__(self, is_word=False, char='', init_list_size=60):
        self.char = char
        self.is_word = is_word
        self.now_words = 0  # 表示填充的字数
        self.child_list = [None] * init_list_size
```

采用hash存储

4.4.1：采用 Trie 树+Hash 存储数据结构

减少匹配次数，传统机械匹配分词天然缺点是匹配次数过多。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）

- **测试代码**，为教师和助教检查代码方便，专门提供了检查接口。运行方式：取消 **lab_code/Test_My_Code.py** 文件主函数中的 **run_part_4()**的注释，注释掉其他执行代码并运行此程序即可

```
if __name__ == '__main__':
    # test_part_1()  # 测试程序运行生成词典，词典产生文件为io
    # test_part_2()  # 测试代码第二部分，即最少代码量实现机械
    # test_part_3()  # 测试代码第三部分，即正反向分词效果分析
    # test_part_4()  # 测试代码第四部分，即首先运行模块4的机械
    run_part_4()  # 运行模块4的机械匹配分词算法
    # test_part_5_1()  # 测试一元文法+未登录词识别
```

图 4.4.2：运行第四部分优化的分词代码

- **优化 FMM 代码逻辑**，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，由**公式(4)**可知查找成功时，查找长度为 **O(2)**（取决于装载因子，实验装载因子为 **2/3**）

$$S_{nl} \approx \frac{1}{2}(1+\frac{1}{1-\alpha}) \quad (4)$$

$$U_{nl} \approx \frac{1}{2}(1+\frac{1}{(1-\alpha)^2}) \quad (5)$$

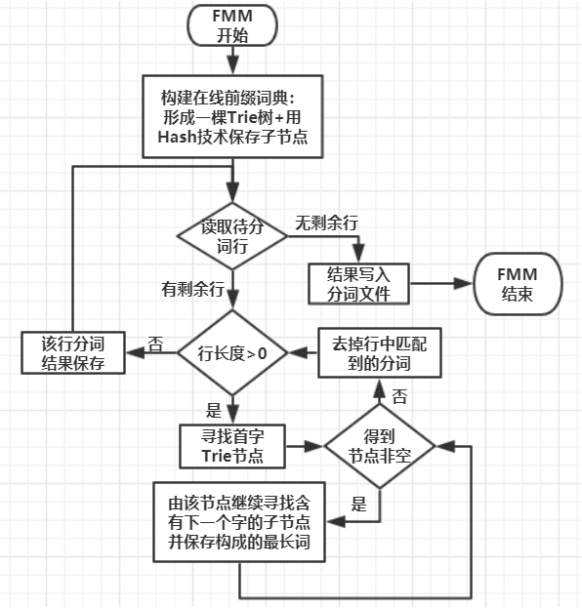

图 4.4.3：优化后 FMM 分词算法的逻辑

- **优化 BMM 代码逻辑**，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**后缀**的上一个字（区别于前缀 Trie 树），父节点的子节点列表同样采用 **Hash 思想**以及**线性冲突探测技术**。对于待分词行，若最后一个字在 Trie 树中，则说明可能存在以该字为后缀的词，继续判断前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），并将此行后面的分词去掉，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。无论是 FMM 还是 BMM 采用的 Hash 思想都是以空间换时间的算法策略，实验设置（可修改配置）装载因子为 2/3，即列表空间中存在 33.3%浪费情况，但这可被容忍，因为通过高效的查找结构，使得算法性能优化了**上千倍**

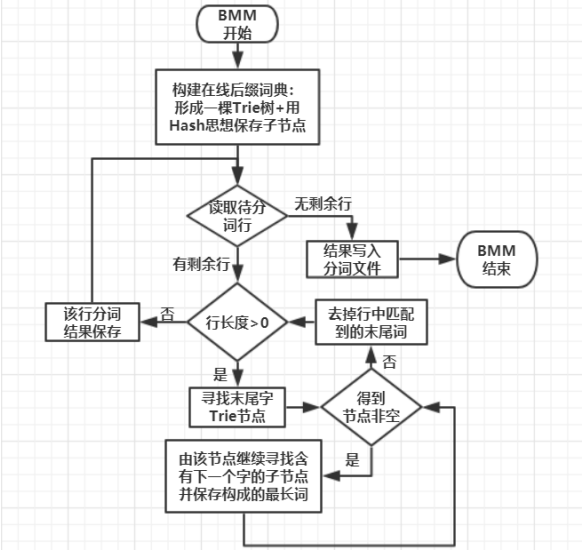

图 4.4.4：优化后 BMM 分词算法的逻辑

- **优化对比程序**，运行方式：取消 **lab_code/Test_My_Code.py** 文件主函数中的 **test_part_4()**的注释，注释掉其他执行代码并运行此程序即可

```
if __name__ == '__main__':
    # test_part_1()  # 测试程序运行生成词典，词典产生
    # test_part_2()  # 测试代码第二部分，即最少代码量
    # test_part_3()  # 测试代码第三部分，即正反向分词
    test_part_4()  # 测试代码第四部分，即首先运行模块
    # run_part_4()  # 运行模块4的机械匹配分词算法
```

图 4.4.5：测试实验第 4 部分

- **time_optimize 函数**，此函数存在于 Part_3.py 文件中，目的是分别运行优化前后得程序得到对应的运行时间（使用 **time.time()**函数实现）。程序的运行时间已经输出在 **io_file/time_cost.txt** 文件中，经过测试可知，优化前得单个 FMM 程序的执行时间在 5h 左右，优化后的单个 FMM 程序时间在 8s 左右，可以看到时间性能是原来的上千倍。值得注意的是以上时间的获取是通过将 **199801_sent.txt** 整个文本作为测试集运行得到得时间结果。后期为了测试集的有效性，将训练集和测试集**划分为 9：1 的部分**，因此后期时间运行变为大致原来的 1/10

```
优化后：采用hash查找提高查找速度；采用前缀匹配，减少查找次数
FMM耗时 8.714634418487549s
BMM耗时 8.781436204910278s

优化前：为满足最少代码量的要求，采用简单的list查找的方式进行词是否在词典中的判断
FMM耗时 15661.08213434434s
BMM耗时 17121.78143620491s
```

图 4.4.6：优化前后时间对比

- **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

4.5 基于统计语言模型分词系统实现-概述

该部分实现了所有的功能，包括基于 MM 的一元文法分词、基于 MM 的二元文法分词、基于 HMM 的未登录词别以及基于 MM 的分词和基于 HMM 的未登录词识别综合相对最优处理分词系统。

此部分代码文件名为 **Part_5_X.py**，其中 X 可谓 1、2、3 分别表示一元文法分词系统、二元文法分词系统、HMM 分词和未登录词处理系统。

此部分代码的训练集和测试集全部来源于标准给定文件 **199801_seg&pos.txt** 和 **199801_sent.txt**，且其中训练集和测试集路径分别为 **io_file/hmm/train.txt**（标准文件的其中 9/10）和 **io_file/hmm/test.txt**（标准文件的剩余 1/10）。

- **测试代码**，为教师和助教检查代码方便，专门提供了检查接口。运行方式：取消

lab_code/Test_My_Code.py 文件主函数中的 **test_part_5_X()**的注释，注释掉其他执行代码并运行此程序即可运行 Part_5_X 的代码，且如果训练文本改变的话，需要终端输入 T 以重新生成训练集、测试集和标准答案集

```
if __name__ == '__main__':
    # test_part_1()  # 测试程序运行生成词典，词典产生文
    # test_part_2()  # 测试代码第二部分，即最少代码量实
    # test_part_3()  # 测试代码第三部分，即正反向分词效
    # test_part_4()  # 测试代码第四部分，即首先运行模块
    # run_part_4()  # 运行模块4的机械匹配分词算法
    test_part_5_1()  # 测试一元文法+未登录词识别
    test_part_5_2()  # 测试二元文法+未登录词识别
    test_part_5_3()  # 测试纯HMM分词
```

图 4.5.1：测试相对应实验部分的代码

- **词典依赖**，实验第 5 部分产生了两个词典文件，分别是 **io_file/dic/dic/uni_dic.txt** 和 **io_file/dic/bigram_dic.txt**，编码格式为 **utf-8**。不同于最基本的 dic.txt 文件，**uni_dic.txt** 作为一元文法的依赖词典，保存了训练集中所有的词和其对应的词频，具体组织方式如图 4.5.2 所示；**bigram.txt** 作为二元文法的依赖词典，保存了训练集中所有的词及与其相连的前一个词和后一个词及两者同时出现的词频，若一个词为句首词，则前词为 BOS；若为句尾词，则后词为 EOS；词典总条目为 461412，具体组织方式如图 4.5.3 所示：

```
安理会 74
安生 16
安盟 4
安石 1
安礼 1
安稳 6
```

图 4.5.2：一元文法词典

```
日子 长 1
日子 ， 5
日寇 打击 1
日寇 珍珠港 1
日寇 的 1
日寇 粉碎 1
日寇 面对 1
日寇 ， 3
日寇 9月 1
```

图 4.5.3：二元文法词典

- **分词结果文件**，分词结果文件在 io_file/seg/文件中，**seg_mwf.txt**、**seg_bigram.txt**、**seg_hmm.txt** 分别是一元文法分词文件、二元文法分词文件、HMM 分词文件

4.6 基于统计语言模型的一元文法实现

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。

- **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_Freq** 数据结构中，并更新总词数

```
for line in lines:
    word, freq = line.split()[0:2]  # 离线词典每行的属性通过空格分隔
    Word_Freq[word] = int(freq)  # 将该词存入到词典中
    Word_Num_Count += int(freq)
```

图 4.6.1：生成在线词典结构

- **一元文法算法思想**，对每一行测试文本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**平滑处理**如图 4.6.2，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，算法流程如图 4.6.3

```
for idx in range(n - 1, -1, -1):  # 动态规划求最大路径
    route[idx] = max((log(Word_Freq.get(line[idx:x + 1], 0) or 1) - log_total +
                      route[x + 1][0], x) for x in dag[idx])
return route
```

图 4.6.2：平滑处理以及动态规划求解

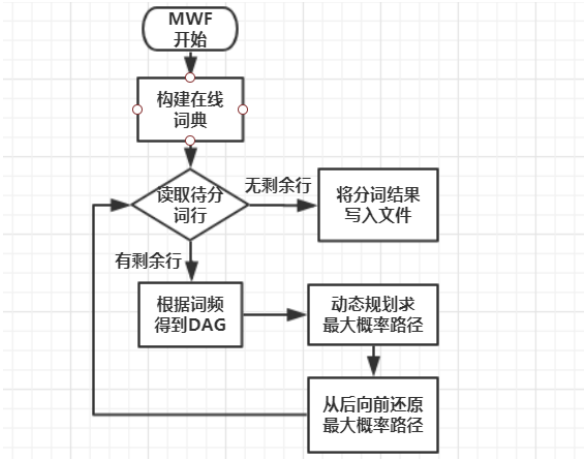

图 4.6.3：一元文法流程图

- **分词结果，召回率很低**，为 70%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

```
1/ 9/ 9/ 8/ 0/ 1/ 0/ 1/ -/ 0/ 1/ -/ 0/ 0/ 1/ -/ 0/ 0/ 1/ 迈向/
1/ 9/ 9/ 8/ 0/ 1/ 0/ 1/ -/ 0/ 1/ -/ 0/ 0/ 1/ -/ 0/ 1/ 1/ 实现/
1/ 9/ 9/ 8/ 0/ 1/ 0/ 1/ -/ 0/ 1/ -/ 0/ 0/ 2/ -/ 0/ 0/ 4/ 1 9 9
1/ 9/ 9/ 8/ 0/ 1/ 0/ 1/ -/ 0/ 1/ -/ 0/ 0/ 3/ -/ 0/ 0/ 3/ 党/ 和
1/ 9/ 9/ 8/ 0/ 1/ 0/ 1/ -/ 0/ 1/ -/ 0/ 0/ 4/ -/ 0/ 0/ 2/ 向/ 广
1/ 9/ 9/ 8/ 0/ 1/ 0/ 1/ -/ 0/ 1/ -/ 0/ 0/ 5/ -/ 0/ 0/ 2/ 元旦/
1/ 9/ 9/ 8/ 0/ 1/ 0/ 1/ -/ 0/ 2/ -/ 0/ 0/ 2/ -/ 0/ 0/ 1/ 忠诚/
```

图 4.6.4：分词结果

```
标准文件:    ../io_file/hmm/std.txt
对比文件:    ../io_file/seg/seg_mwf.txt
准确率:  94.3022353853748%
召回率:  70.69629747107295%
F值:     80.81064442908561%
```

图 4.6.5：分词评价

4.7 基于统计语言模型的二元文法实现

一阶马尔可夫链即二元文法的假设是一个词语的出现仅依赖于前一个词语。一元文法的假设是一个词出现的概率与其余词无关，这显然是一个过强假设，一元文法对上下文的信息利用是十分不足的。二元文法的假设相对弱化了一元文法的强假设，使得该模型从原理上讲更加的合理化。同时二元文法可以利用更多上下文信息，从而进一步提高性能。

- **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_Freq** 数据结构中，并更新总词数

```
for line in lines:
    word, freq = line.split()[0:2]  # 离线词典每行的属性通过空格分隔
    Word_Freq[word] = int(freq)  # 将该词存入到词典中
    Word_Num_Count += int(freq)
```

图 4.7.1：生成在线词典结构

- **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，计算一个词出现的概率依赖的公式为**公式 5**

$$p(w_i|w_{i-1}) = \frac{c(w_{i-1}w_i)}{\Sigma_{w_i} c(w_{i-1}w_i)} \tag{5}$$

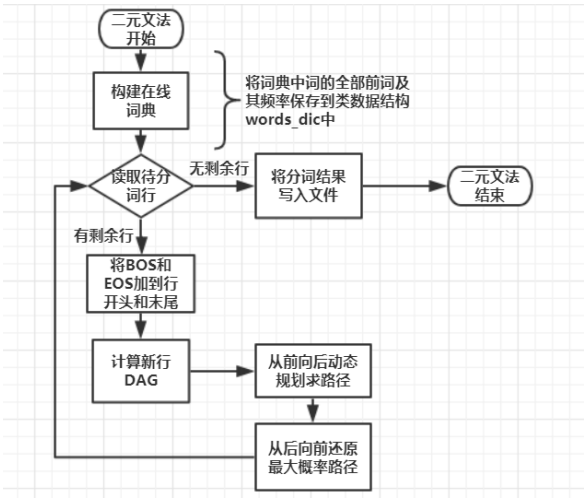

图 4.7.2：二元文法程序流程图

- **条件相关最大概率求解**，二元文法在条件相关的最大概率路径求解上与一元文法有所区别。二元文法**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，如图 4.7.3 所示

```
for word in pre_words:
    if word == 'BOS':
        route[word] = (0.0, 'BOS')
    else:
        pre_list = word_graph.get(word, list())  # 取得该词对应的前词列表
        route[word] = (-65507, 'BOS') if not pre_list else max(
            (pre_graph[pre][word] + route[pre][0], pre) for pre in pre_list)
```

图 4.7.3：二元文法动态规划

- **0 概率平滑处理**，二元文法在概率平滑处理上与一元文法有所区别。采用**对数概率**的处理方式避免浮点数下溢，概率结果计算方式变为相应的**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采用**加 1 平滑**处理，分母整体加上总词数，分子词频加 1，如图 4.7.4 所示

```
pre_word_freq = Part_5_1.Word_Freq.get(pre_word, 0)  # 前词词频
condition_word_freq = DicAction.words_dic.get(word, {}).get(pre_word, 0)  # 组合词频
return log(condition_word_freq + 1) - log(pre_word_freq + Part_5_1.Word_Num_Count)
```

图 4.7.4：概率平滑+对数处理

- **还原最大概率路径**，与一元文法有所区别。二元文法从得到的动态规划求解的结果 route 的**末尾**’EOS’开始还原，不断向前寻找最大概率词，直到寻找到’BOS’开始字符串，还原结束，经过的每一个节点即是最终的分词结果，如图 4.7.5 所示

```
position = 'EOS'
while True:
    position = line_route[position][1]
    if position == 'BOS':
        break
    seg_line = line[position[0]:position[1]] + '/ ' + seg_line
```

图 4.7.5：分词结果还原

- **二元文法分词结果分析**，二元文法分词结果对于测试文本前的一个标识数字串无法处理，从来使得分词总词数大大增加，导致分词召回率很低，准确率比较高（测试文本为标注文本的 1/10 部分），但是对于整体的处理来看，二元文法的优势还是比较明显的（因为二元文法利用了**更多的上下文信息**）

```
标准文件:    ../io_file/hmm/std.txt
对比文件:    ../io_file/seg/seg_bigram.txt
准确率:  94.1030251745293%
召回率:  70.61087888248055%
F值:     80.6816807738815%
```

图 4.7.6：分词结果评测

```
1/ 9/ 9/ 8/ 0/ 1/ 0/ 1/ -/ 0/ 1/ -/ 0/ 0/ 1/ -/ 0/ 0/ 1/
1/ 9/ 9/ 8/ 0/ 1/ 0/ 1/ -/ 0/ 1/ -/ 0/ 0/ 1/ -/ 0/ 1/ 1/
1/ 9/ 9/ 8/ 0/ 1/ 0/ 1/ -/ 0/ 1/ -/ 0/ 0/ 2/ -/ 0/ 0/ 4/
1/ 9/ 9/ 8/ 0/ 1/ 0/ 1/ -/ 0/ 1/ -/ 0/ 0/ 3/ -/ 0/ 0/ 3/
1/ 9/ 9/ 8/ 0/ 1/ 0/ 1/ -/ 0/ 1/ -/ 0/ 0/ 4/ -/ 0/ 0/ 2/
1/ 9/ 9/ 8/ 0/ 1/ 0/ 1/ -/ 0/ 1/ -/ 0/ 0/ 5/ -/ 0/ 0/ 2/
1/ 9/ 9/ 8/ 0/ 1/ 0/ 1/ -/ 0/ 2/ -/ 0/ 0/ 2/ -/ 0/ 0/ 1/
```

图 4.7.7：分词错误示例

4.8 基于 HMM 的未登录词处理

HMM 算法对于未登录词的识别具有一定的特点和优势。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 算法**对测试集的每一行进行寻找最有可能产生观测事件序列（测试行）的维特比路径——隐含状态序列（分词结果），如下图公式所示

$$\delta_t(i)=\max_{i_1,i_2,\cdots,i_{t-1}} P(i_t=i,i_{t-1},\cdots,i_1,o_t,\cdots,o_1\mid\lambda),\quad i=1,2,\cdots,N$$

图 4.8.1：定义 t 时刻，状态 i 的单路径最大值

$$\begin{aligned}\delta_{t+1}(i)&=\max_{i_1,i_2,\cdots,i_t} P(i_{t+1}=i,i_t,\cdots,i_1,o_{t+1},\cdots,o_1\mid\lambda)\\&=\max_{1\le j\le N}[\delta_t(j)a_{ji}]b_i(o_{t+1}),\quad i=1,2,\cdots,N;\ t=1,2,\cdots,T-1\end{aligned}$$

图 4.8.2：对应的 t+1 时刻

- **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_Dic** 集合，用于判断一个词是否在词典中

- **算法设计思想**，从参数文本中读取 HMM 模型训练的参数 **Π：初始状态概率**，**A：状态转移概率**和 **B：发射概率**；不断处理每一行，然后利用 **Viterbi 算法**输出该行测试文本最大可能的分词隐藏状态；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，具体流程如截流程图 4.8.3 所示

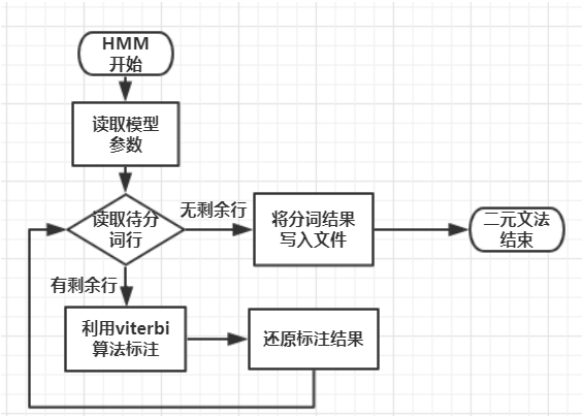

图 4.8.3：HMM 流程图

- **分词结果**，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 **78%**左右，召回率为 **75%**左右（保存在评测文件 **io_file/score.txt** 中），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，尤其是对于**未登录词识别**较为准确

```
标准文件:    ../io_file/hmm/std.txt
对比文件:    ../io_file/seg/seg_hmm.txt
准确率:  78.43946125096961%
召回率:  75.86747830238546%
F值:     77.13203491345311%
```

图 4.8.4：HMM 分词评测

- **参数训练**，模型参数的训练文本来源于 199801_seg&pos.txt 标准文件的 9/10 部分，通过对训练文本标记后的数据进行相应的统计，并对分词概率采用对数计算+概率相加的方式解决概率浮点值下溢的情况，模型参数全部保存在 **io_file/hmm/**文件里，参数计算过程如图 4.8.5

```
    for i in range(len(line_tag)):
        State_Count[line_tag[i]] += 1  # 计算状态总出现次数
        B[line_tag[i]][line_word[i]] = B[line_tag[i]].get(line_word[i], 0) + 1
        if i > 0:
            A[line_tag[i - 1]][line_tag[i]] += 1  # 转移概率变化
for state in States:
    Pi[state] = Min if Pi[state] == 0 else log(Pi[state] / Word_Count)
    for state_1 in States:  # 计算状态转移概率
        A[state][state_1] = Min if A[state][state_1] == 0 else log(
            A[state][state_1] / State_Count[state])
    for word in B[state].keys():  # 计算发射概率
        B[state][word] = log(B[state][word] / State_Count[state])
```

图 4.8.5：模型参数计算

```
B -0.6225411154825581
M -3.14e+100
E -3.14e+100
S -0.769119769439755
B
 B -3.14e+100
 M -1.7562833207173973
 E -0.1895703529496381
 S -3.14e+100
```

图 4.8.6：模型参数截图

- **优化方向**，HMM 模型参数的训练极大地影响了分词的效果，因此提高**模型参数的拟合程度**是需要综合考虑的一个方向；同时优于模型只采用 4 个状态来表述测试文本的隐藏状态，对于描述词语的状态来说是比较少的。但是如果采用此行作为隐状态的话，算法复杂度又太高，因此需要在词的**隐状态数目**上做一个合理的 **tradeoff**，以实现模型的最优化

```
if len(word) == 1:
    line_tag.append('S')
    Pi['S'] += 1
else:
    line_tag.append('B')
    line_tag.extend(['M'] * (len(word) - 2))
    line_tag.append('E')
    Pi['B'] += 1
```

图 4.8.7：隐状态只有 4 个

4.9 基于统计语言模型分词系统实现-综合

一元文法和二元文法对于未登录词的处理都是十分欠缺的，而 HMM 模型则可以很好地解决这个问题。

- **优化实现**，可在二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

- **结果概述**，运用类似的思想可以达到较好的分词效果，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

很大的提高，最终可以达到 95%和 96%的准确率，如下图所示

```
标准文件:    ../io_file/hmm/std.txt
对比文件:    ../io_file/seg/seg_mwf.txt
准确率:  96.0219660108596%
召回率:  93.63503524153344%
F值:     94.81348025136214%
```

图 4.9.1：一元文法的基础上的优化

```
标准文件:    ../io_file/hmm/std.txt
对比文件:    ../io_file/seg/seg_bigram.txt
准确率:  95.82275580001411%
召回率:  93.55094102561895%
F值:     94.67322154051146%
```

图 4.9.2：二元文法基础上的优化

5 实验结论及未来工作

5.1 实验总结

- 实验通过**简短的代码**实现了复杂度很大的机械匹配分词，后续算法对**查找结构和存储结构**进行调整和优化通过减少查找次数和减少匹配次数实现了时间性能上 2500 倍的优化；相应的**评价代码**也为整个实验的分词系统性能表现提供了数据支持和比较

- 同时不同于机械匹配分词的思想，**统计语言模型**分词系统的实现从概率上对汉语言分词提供了新的解决方案

- 未登录词的问题仍然无法通过简单的 MM 算法解决，利用 HMM 的未登录词识别的思想对 MM 模型生成的**分词解空间进行扩充**可以得到很好的算法性能。

5.2 未来工作

- 虽然在本实验中，最终的算法在开放测试集中达到了 95%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**

- **模型平滑处理**，目前模型为了避免出现 0 概率的问题，采用+1 平滑处理使得每个词频的概率至少为 1，未来实验可以采用平滑技术的核心方法：Good-Turing 估计法进行数据平滑处理

$$p(\omega_i|\omega_{i-1}) = \frac{1+c(\omega_{i-1}\omega_i)}{\sum_{\omega_i}[1+c(\omega_{i-1}\omega_i)]} = \frac{1+c(\omega_{i-1}\omega_i)}{|V|+\sum_{\omega_i} c(\omega_{i-1}\omega_i)}$$

图 5.2.1：目前模型数据平滑处理

对于任何一个出现r次的n元语法，都假设它出现了r^*次。

$$r^* = (r+1)\frac{n_{r+1}}{n_r}$$

图 5.2.2：拟采用的数据平滑处理方式

- **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 95%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）

参考文献

梁以敏.基于统计的汉语词性标注方法的研究[DB/OL].https://kns.cnki.net/KCMS/detail/detail.aspx?dbcode=CMFD&dbname=CMFD9904&filename=2004094913.nh&uid=WEEvREdxOWJmbC9oM1NjYkZCbDdrNTBKaXo4QnNZWGJIckpkZzNPMjFISDk=$R1yZ0H6jyaa0en3RxVUd8df-oHi7XMMDo7mtKT6mSmEvTuk11l2gFA!!&v=MTI4NzZyV00xRnJDVVJMT2VadWRwRnkzbFVick1WMTI3R3JPeEd0ak5ySkViUElSOGVYMUx1eFlTN0RoMVQzcVQ=,2004-03-01.

coolwriter.哈希表查找——成功和不成功时的平均查找长度[EB/OL].https://blog.csdn.net/coolwriter/article/details/81604058,2018-08-12.

长颈鹿 Giraffe.基于二元语法模型的中文分词[EB/OL].https://www.cnblogs.com/giraffe/archive/2011/12/09/2282337.html?utm_source=tuicool,2011-12-09.

薛沛雷.用 python 实现 NLP 中的二元语法模型[EB/OL].https://blog.csdn.net/firparks/article/details/54563683,2017-01-15.

orangleliu.[Python]全角半角转换的 Python 实现[EB/OL].https://blog.csdn.net/orangleliu/article/details/38225161,2014-07-28.

52nlp.HMM 学习最佳范例四：隐马尔科夫模型[EB/OL].http://www.52nlp.cn/hmm-learn-best-practices-four-hidden-markov-models,2009-06-23.