汉语分词系统

张明远
120L030501
哈尔滨工业大学

摘要

词是语言中最小的能够独立运用的音义结合体。有别于大部分西方语言，汉语文本是基于单字的，词与词之间没有显性的界限标志，因而分词就成为了汉语文本分析处理中首先要解决的问题和后续所有中文信息处理工作的基础。本次实验任务就是对汉语分词技术进行实践。对于词典的构建，本文实现了基于哈希表的 Trie 树存储。通过得到的词典实现了正反向最大匹配方法及其速度优化。基于得到的分词结果，实现了结果精确率、召回率、F 值自动评价。对于统计语言模型，还实现了 Bigram 模型。通过实现未登录词识别、分词系统集成和分词后处理，提升了 Bigram 模型的性能。

1 绪论

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。在英文的行文中，单词之间是以空格作为自然分界符的，而中文只是字、句和段能通过明显的分界符来简单划界，唯独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，不过在词这一层上，中文比之英文要复杂的多、困难的多，因此分词是汉语文本分析处理中首先要解决的问题。

如何面向大规模开放性应用是中文分词研究面临的主要问题，细分下来，汉语自动分词的主要困难来自如下三个方面：分词规范、歧义切分和未登录词的识别。(宗成庆, 2013)

1.1 汉语分词规范问题

“词”这个概念一直没有被汉语言学界准确定义。“词是什么”（词的抽象定义）及“什么是词”（词的具体界定），这两个基本问题众说纷纭，迄今拿不出一个公认的、具有权威性的词表来。从计算的严格意义上说，自动分词是一个没有明确定义的问题。

1.2 歧义切分问题

歧义字段在汉语文本中普遍存在，因此，切分歧义是汉语自动分词研究中一个不可避免的“拦路虎”。汉语词语边界的歧义切分问题比较复杂，处理这类问题时往往需要进行复杂的上下文语义分析，甚至韵律分析，包括语气、重音、停顿等。

1.3 未登录词问题

未登录词又称为生词（unknown word），可以有两种解释：一是指已有的词表中没有收录的词；二是指已有的训练语料中未曾出现过的词。在第二种含义下，未登录词又称为集外词（out of vocabulary,OOV），即训练集以外的词。对于大规模真实文本来说，未登录词对于分词精度的影响远远超过了歧义切分。

2 研究现状

中文分词是中文信息处理领域的重要基础部分。经过多年的研究与发展，目前较为成熟的分词方法可分为以下三种: 基于字符串匹配的分词方法、基于理解的分词方法和基于统计的分词方法。(孙茂松，邹嘉彦, 2001)

2.1 基于字符串匹配的方法

基于字符串匹配的方法，又名机械分词方法，这种方法使用一个机器词典与待分词串进行匹配，匹配成功就意味着分出一个词。常用的几种机械分词方法如下：

1）正向最大匹配法（FMM）
2）逆向最大匹配法（BMM）
3）最少切分（FWM）
4）双向最大匹配法（BM）

2.2 基于理解的分词方法

这种分词方法是通过让计算机模拟人对句子的理解，达到识别词的效果。其基本思想就是在分词的同时进行句法、语义分析，利用句法信息和语义信息来处理歧义现象。由于汉语语言知识的笼统、复杂性，难以将各种语言信息组织成机器可直接读取的形式，因此目前基于理解的分词系统还处在试验阶段。

2.3 基于统计的分词方法

基于统计的分词方法是在给定分词语料库的前提下，利用统计模型学习词语切分的概率规则，按一定的指标计算，从而实现对未知文本的切分。。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成为了主流方法。主要的统计模型有：

1）N 元文法模型（N-gram）
2）隐马尔可夫模型（HMM）
3）最大熵模型（ME）
4）条件随机场模型（CRF）

3 方法和模型

本次实验使用的语料库为 1998 年 PFR 人民日报标注语料库，PFR 语料库 1 月份的语料公开于人民日报社、北京大学计算语言学研究所和富士通研究开发中心有限公司的主页上。下面的所有工作都是基于此语料库进行。

3.1 词典的构建

在这个部分我们要进行分词词典的构建，此词典主要供 3.2-3.4 部分的基于机械匹配的分词系统使用，主要使用正则表达式的方法从语料库中提取出词。3.5 节之后的词典由于没有限制，使用了 python 内置的 dict 结构，利用 json 存储，不在叙述。

3.1.1 分词单位标准

PFR 语料库的分词规范参阅《现代汉语语料库加工 词语切分与词性标注规范》(北京大学计算语言学研究所, 1999)。这里主要说明词典中加入的词的取舍问题。为了提高模型的泛化能力，防止过拟合，不是全部的词都可以加入词典，有一部分词需要进行处理。词典的构建符合下列几条标准：

1）语料中正常由汉字组成的词，全部加入词典。

2）数字串、字幕串以及各种组合情况的非汉字字符串采用标记替换的方法将其替换为单个字符后加入词典。例如使用“#”号替换数字串，那么“2021 年”这个词会以“# 年”的格式加入词典。

3）由于专有名词被分为更细分的词组，故不将专有名词作为完整的单个词语加入字典。

3.1.2 词典文件格式

词典文件 dict.txt 为 gbk 编码的文本文档，每一行为一个单个的词。按编码顺序进行排序。

图 1: dict.txt 文件

3.1.3 词典分析

对词典的分析主要从其规模、选择性、顺序性、编码等方面进行。

1）**词典规模**：词典中共收集了 51446 个词，覆盖了大部分常用词汇，这说明在识别常用词组成的语句时词典会取得很好的效果。

2）**词典选择性**：通过分词标准中叙述的词的选择原则，词典中的词数从 70217 个下降到了 51446 个词，这样提高了查询词典时的效率和词典对开放测试的泛化能力，使词典的时间性能和准确率都有了提升。

3）**词典顺序性**：通过在形成词典时对其进行排序，词典中的词具有高度顺序性，这极大地方便了后续读取词典形成相关数据结构的实现，也方便了我们观察词典的构成。

4）**词典编码**：词典是 gbk 格式的字符串文件，关键信息按行分布的。这样做符合语料库的编码，可以方便读取词典方法的最少代码量实现

3.2 正反向最大匹配分词实现

本节实现正反向最大匹配分词算法，分词词典采用 Trie 树存储。程序预设了一个预处理词典，分词时先对句子中的非汉字串进行替换，然后进行分词，分词结束后再进行反替换。

3.2.1 词典数据结构及其算法

Trie 树，是一种多叉树结构。Trie 树的核心思想是空间换时间，利用字符串的公共前缀来减少无谓的字符串比较以达到提高查询效率的目的。Trie 树的关键字一般都是字符串，Trie 树把每个关键字保存在一条路径上，而不是一个结点中。另外，两个有公共前缀的关键字，在 Trie 树中前缀部分的路径相同，每个节点的所有子节点包含的字符互不相同。所以 Trie 树又叫做前缀树（Prefix Tree）。一个包含字符串“add”，“adbc”，“bye” 的字典树结构如下图所示

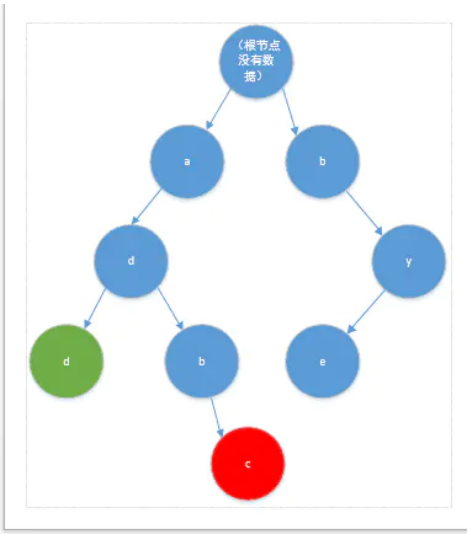

图 2: Trie 树示例

Trie 树节点定义为一个包含三个属性的 python 类，分别为 element_list，character，has_the_word。element_list 为 python 列表，保存该节点的所有子节点，character 为一个字符，表示该节点代表的字符，has_the_word 为布尔值，表示该节点代表的字符串是否存在。如下图：

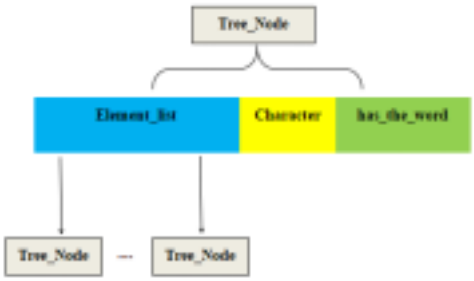

图 3: Trie 树节点定义

Trie 树作为存储结构要实现的主要操作为 CRUD 方法，即创建（Create）、更新（Update）、读取（Retrieve）和删除（Delete）方法。其中 Update 方法可以用 Create 和 Delete 替代，因而下面主要叙述 Create，Retrieve 和 Delete 方法实现。由于树结构的递归性，三种操作都可以使用递归快速解决。下面给出 Retrieve 算法的伪代码如下，其余不再展示：

Algorithm 1 Trie 树 Retrieve 方法

Input: $Trie\_Tree$: 待查找字符串的 Trie 树;
key: 待查找的字符串;

Output: 是否存在这样的字符串，布尔值

1: **if** key 为空串 **then**
2: 返回节点的 has_the_word;
3: **end if**
4: ch = $key[0]$;
5: 在 $Trie\_Tree$ 节点的 $element\_list$ 中找到 $character$ 为 ch 的子树 $subtree$
6: **if** $subtree$ 不存在 **then**
7: 返回 False
8: **end if**
9: 调用 $subtree$Retrieve 方法查找 $key[1:]$

3.2.2 FMM 和 BMM 算法及实现

最大匹配是指以词典为依据，取词典中最长单词为第一个次取字数量的扫描串，在词典中进行扫描（为提升扫描效率，还可以跟据字数多少设计多个字典，然后根据字数分别从不同字典中进行扫描）。其分词原理是: 单词的颗粒度越大，所能表示的含义越确切。最大匹配算法有三种:

1、正向最大匹配（FMM）
2、逆向最大匹配（BMM）
3、双向匹配（BM）

三种算法原理都一样，FMM 算法，是从前向后扫描的过程，对于输入的一段文本从左至右、以贪心的方式切分出当前位置上长度最大的词。而 BMM 算法则相反，是从后向前扫描的过程。

例如“市场中国有企业才能发展”句，采用 FMM 得出的结果为“市场/中国/有/企业/才能/发展/”，而如果采用 BMM，则会得出“市场/中/国有企业/才能/发展/”的结果。

右侧给出 FMM 算法和 BMM 算法的通用伪代码

利用 3.1 生成的词典对给定文件 199801_sent.txt 进行分词，得到 FMM 的

Algorithm 2 FMM 和 BMM 算法

Input: $dict$: 分词词典;
$sentence$: 待分词的字符串;

Output: $result$: 分词结果

1: 设自动分词词典 $dict$ 中最长词条所含汉字个数为 I;
2: 取 $sentence$ 当前字符串序数中的 I 个字作为匹配字段，查找分词词典 $dict$。
3: **if** 如果词典中找到这样的一个 I 字词 **then**
4: 匹配成功，匹配字段作为一个词被切分到 $result$，跳转 6
5: **end if**
6: **if** 如果词典中找不到这样的一个 I 字词 **then**
7: 匹配失败
8: **end if**
9: 匹配字段去掉最后一个汉字，I–
10: 重复 2-4，直至切分成功为止
11: 为 I 再次赋初值，跳转 2，直到切分出所有词为止

分词结果 seg_FMM.txt 和 BMM 的分词结果 seg_BMM.txt 如下:

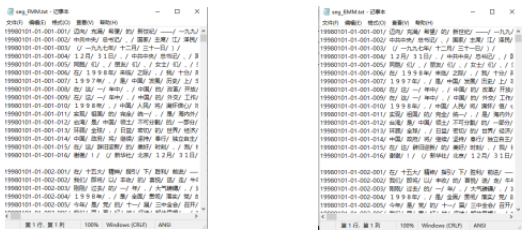

图 4: seg_FMM.txt 和 seg_BMM.txt

3.2.3 实验心得

在实验的过程中，我亲自实现了 FMM 和 BMM 算法，了解到仅仅是基于词的物理属性的机械匹配就能实现像样的分析，让我非常震撼。模型的简化一直是工程上的重要问题，中文分词算法仅靠简单的模型便能完成分词这个听起来就很复杂的任务，真的很了不起。在词典的实现中，我第一次实现了 Trie 树这个在之前听过但没使用过的数据结构，使我的能力有

了上涨。然而实现的模型在未优化的情况下的极低效率也让我明白了实际应用中优化的重要性，这次试验对我有很大的启发作用。

3.3 正反向最大匹配分词效果分析

3.3.1 分词结果评价标准

在机器学习中的模型评价指标有准确率（precision）召回率（recall）和 F 值。

于是有二分类的混淆矩阵：

表 1: 混淆矩阵

真实值 | 预测值	Positive(1)	Negative(0)
Positive(1)	TP	FP
Negative(0)	FN	TN

其中，TP 代表预测是正样本（1），真实为正样本（1），预测对了；

FN 代表预测是正样本（1），真实为负样本（0），预测错了；

FP 代表预测是负样本（0），真实为正样本（1），预测错了；

TN 代表预测是负样本（0），真实为负样本（0），预测对了。

精准率（Precision）又称查准率：预测为正的样本中真实为正的样本

$$Precision = \frac{TP}{TP + NP}$$

召回率（Recall）又称查全率：真实为正的样本中预测为正的样本

$$Recall = \frac{TP}{TP + FN}$$

在某些情况下，P 和 R 是矛盾的，F 值是综合衡量两者的指标。

$$F1 = \frac{2PR}{P + R}$$

但是在分词中标准答案和分词结果数不一定相等，因此要做一个思维转换。对于长度为 n 的字符串，分词结果是一系列单词。设每个单词按照其在文本中的起止位置可以记作区间 [i , j], 其中 $1 \leq i \leq j \leq n$。那么标准答案所有区间构成集合 A 为正类，其它情况作为负类。同时，分词结果所有单词构成的区间集合为 B。因而可得：

$$TP \cup FN = A$$

$$TP \cup FP = B$$

$$A \cap B = TP$$

因此相应的计算公式如下：

$$Precision = \frac{|A \cap B|}{|B|}$$

$$Recall = \frac{|A \cap B|}{|A|}$$

根据这个公式，便可计算出相应分词结果的 Precision，Recall 和 F1 值，进而对分词结果进行评判。由于分词程序不作要求，这里不再给出。

3.3.2 正反向最大分词性能指标及分析

由上述计算方法，我们得出了之前实现的 FMM 和 BMM 分词性能指标（封闭测试）如下：

表 2: FMM 和 BMM 分词性能指标

模型	Precision	Recall	F1 值
FMM	0.977	0.961	0.969
BMM	0.979	0.964	0.972

可以看到 BMM 在整体各项指标上均高于 FMM 的结果。下面来结合分词结果具体进行分析为什么会有这种差异。

汉语构词: 汉语构词模式往往惯于在词语后添加字词从而构成新词，而较少在词前进行这种操作，这也导致在分词时，使用 BMM 造成分词歧义的可能性低于 FMM，例如“我们在经济社会发展中还面临不少困难”这句，如果使用 FMM 分词，会被分为“我们/在/经济社/会/发展/中/还/面临/不少/困难/”，而 BMM 则是正确的“我们/在/经济/社会/发展/中/还/面临/不少/困难/”，这就是由于“经济社”构成了一个更长的新词，便导向了错误的结果。

汉语语法: 汉语一句话一般侧重的词语在句子的后面。汉语语法中大量含有例如偏正结

构短语的语法，这使得汉语句倾向于把长词放置在句末。较长的词语往往出现在句子的后部这一特点也是导致 BMM 分词由于 FMM 分词的一大影响因素。例如“中国愿意加强同联合国和其他国际组织的协调”，会被 FMM 错误的分为“中国/愿意/加强/同联/合国/和/其他/国际/组织/的/协调/”

由于汉语中的以上特征，后向匹配可以适当提高精确度。统计结果表明，单纯使用正向最大匹配的错误率约为 1/169，单纯使用逆向最大匹配的错误率约为 1/245。

3.4 基于机械匹配的分词系统的速度优化

上文实现的基于 Trie 树的分词系统，虽然成功实现了分词，但是其效率过慢，总共花费了 19924.93s，即 5h32min4.93s 才能完成分词工作，这在实际应用时可以说是不可接受的。因而有必要对分词程序的效率进行一定的优化。

通过应用相关工具进行 700 个句子的划分来进行分析，得到了分词程序初版的时间性能分析，下面列出前三名占用时间的方法表格，其中自用时间不包含方法调用其他方法花费的时间。

表 3: 分词程序时间性能分析

方法	自用时间	时间占比
getSubTree	985609ms	93.5%
<listcomp>	13589ms	1.3%
get	8849ms	0.8%

可以看出来，在我们的分词程序中 getSubTree 方法占据了绝大多数的时间，该方法是用来在 Trie 树中寻找符合相应字符的子树的方法，由于 Trie 树节点的字数列表被定义为 list 类型，只能进行顺序查找，因而对于一个含有 n 个子树节点的 Trie 树节点，顺序查找的时间复杂度为 O（n），而对于一次查找长度为 m 的字符串的操作，若 Trie 树中最少的叉数为 n 的话，其时间复杂度就至少为 O（mn），这样的时间复杂度对于大量使用了查找算法的 Trie 树无疑是不理想的。因而尝试对词典查找算法进行优化。

一种思路是替换 Trie 树中的子树存储方法，这里采用哈希表替换列表以实现对算法的优化。哈希表，是根据关键码值 (Key value) 而直接进行访问的数据结构。也就是说，它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。这个映射函数叫做哈希函数，存放记录的数组叫做哈希表。给定表 M，存在函数 f(key)，对任意给定的关键字值 key，代入函数后若能得到包含该关键字的记录在表中的地址，则称表 M 为哈希（Hash）表，函数 f(key) 为哈希 (Hash) 函数。使用 Hash 表存储，查找算法的时间复杂度能降低到 O（1）级别。

建立 Hash 表的关键是找到合适的 Hash 函数，这里以较为常用的 ELFHash 算法 (GMFTBY, 2016) 作为基础构建我们的 Hash 算法。下面给出 ELFhash 的伪代码：

Algorithm 3 ELFhash 算法

Input: str: 一个字符串

Output: $result$：hash 结果

初始化 $hascode$ 和 x 为 0;
取 str_len 为 str 的长度
for $i = 0$ to $str\_len - 1$ **do**
    $hashcode$ = ($hashcode$ « 4) + ord(str[i]);
    x = $hashcode$ & 0xF000000000000000;
    if x != 0 **then**
        $hashcode \triangleq (x$ » 56)
        $hashcode$ &= x
    end if
end for
返回 $hashcode$

通过 ELFhash 我们把一个字符串映射为了一个长整数，但是这整数可以为负数，且一般超过我们能为 Hash 表分配的空间大小，因而我们对其进行对 Hash 表空间大小取余数的操作，就得到的了对于大小为 n 的 Hash 表，能把字符串映射到 0~n-1 标号空间的 hash 函数。而对于 hash 冲突的情况，这里采用了线性探测

再散列的方法，即在发生冲突时通过顺序查找下一个位置的方法寻找键值。

由经验数据，hash 表在装载因子（hash 表中关键字个数和 hash 表长度之比）在 75% 以下的时候有着较高的效率，而频繁扩大 hash 表的空间也会造成一定的效率问题，因而这里在实现 hash 表的时候借鉴了 C++ STL 库中 vector 的实现方法，使用动态扩容技术，在 hash 表建立时初始化一定大小的空间，当 hash 表的装载因子大于 10% 时进行扩容两倍操作，从而兼顾效率和空间占用的平衡。

在确定了 hash 表的建立扩容和 hash 函数的情况下，hash 表其余操作便基本和使用数组没有太大的区别了，因而不在一一列出。以上就是 hash 表的全部内容。

另一种思路就是完全放弃 Trie 树的结构，完全使用 hash 表进行词典的存储，这样相较于第一种思路有着更大的内存占用率，却能够直接进行词的查找，不需要再进行树搜索的过程，因而也更快速。

下面给出优化前和两种思路实现的程序花费时间的对比：

表 4: 分词程序时间对比

方法	**时间 (s)**
纯 $Trie\_Tree$	19924.93
$Trie\_Tree + Hash\_Table$	442.67
纯 $Hash\_Table$	157.80

可以看到程序从原来的要跑 5 个多小时直接降低到了 3-7 分钟的水平，分词效率有了极大的提升。可见算法时间复杂度的下降能够给程序带来指数级的优化倍数提升。

要考虑程序未来的提升方向，我们可以继续对程序进行时间性能分析，通过对 $Trie\_Tree + Hash\_Table$ 版本的程序进行分析前三名占用时间的方法，我们得到了如下结果，其中 __getHashCode 为之前实现的 hash 函数：

我们发现，经过优化后，程序中最占有

表 5: 优化后的分词程序时间性能分析

方法	**自用时间**	**时间占比**
__getHashCode	66446ms	58.9%
<listcomp>	13945ms	13.1%
get	9050ms	8。5%

时间的变为了我们实现的方法 __getHashCode，回顾之前我们对此算法的实现，我们通过 ELFhash 算法将一个字符串映射为了一个长整数，再通过取模的方法将其映射到我们的 hash 表空间。hash 函数在将字符串映射的过程中花费了很多时间。

由此想到的第一种改进思路为将 ELFhash 算法的计算由 python 实现改为 c 语言实现，再通过 python 和 c 的接口进行调用操作，我们知道，python 是一门解释性语言，相对于编译型语言速度很慢，将 python 数值计算外包给 c 语言进行是很常见的做法，能极大的提高效率。第二种改进思路是寻找更合适的 hash 函数，能够将字符串映射为较小的范围，从而减少在取模运算的大量消耗。第三种思路是在建立词典时根据词典字符串能被 ELFhash 映射的最大范围确定 hash 表的空间占用大小，这样就能不进行取模运算，直接利用 ELF 得到的 hash 值进行查找。

3.5 基于统计语言模型的分词系统实现

在这一节里，我实现了基于 Bigram 模型的统计分词系统，首先通过训练语料训练出 bi-gram 词典，通过线性插值算法进行参数平滑，在求解分词结果时使用 viterbi 算法求得结果，从而实现 Bigram 分词。下面进行详细叙述。

3.5.1 统计语言模型 N-gram

统计语言模型试图捕获自然语言的统计规律以改善各种自然语言应用系统的性能，广泛地应用于语音识别、手写体文字识别、机器翻译、键盘输入、信息检索等领域。统计语言建模 (Statistical Language Modeling) 相当于对各种语言单位如字、词、句子或整篇文章进行概

率分布的估计。

给定所有可能的句子 s（还可以是其他语言单位),统计语言模型就是一个概率分布 p(s)。典型地，语言模型将一个句子的概率分解为条件概率的乘积:

$$p(s) = p(w_1, \cdots, w_n) = \prod_{i=1}^{n} p(w_i \mid h_i)$$

这里 w_i 是句中的第 i 个词，$h_i = \{w_1, w_2, w_{i-1}\}$ 称为历史。实际应用中，由于严重的数据稀疏和系统处理能力的限制，统计语言建模只能考虑有限长度的历史。通过将语言模拟成 N 1 阶马尔科夫源，N-gram 模型减少了参数估计的维数:

$$p(w_i \mid h_i) \approx p(w_i \mid w_{i-N+1}, \cdots, w_{i-1})$$

N 值的选择要考虑参数估计的稳定性和描述能力的折衷。Trigram 和 Bigram 是通常的选择。本次选用 Bigram 进行分词。

3.5.2 Bigram 概率估计及词典生成

Bigram 模型是一个 1 阶马尔可夫模型，在 Bigram 模型中假设给定句子 s 中的条件概率 $p(w_i \mid h_i) = p(w_i \mid w_{i-1})$。通过假定所有句子由一个空串 “” 开头，Bigram 模型中的所有概率都可以表示为 $p(w_i \mid w_{i-1})$ 的形式，因而 Bigram 词典便由这种形式的概率组成。

如何估计概率 $p(w_i \mid w_{i-1})$ 是模型建立首先要解决的问题。通常可以使用极大似然方法解决这一问题，对于任意词对 w_i w_{i-1}

$$p(w_i \mid w_{i-1}) = \frac{c(w_{i-1}, w_i)}{\sum_{w_i} c(w_{i-1}, w_i)}$$

其次由于语言模型的训练文本 T 的规模及其分布存在着一定的局限性和片面性，许多合理的语言搭配现象没有出现在 T 中。例如: 一个词串 $w_{i-N+1} \cdots w_i$ 没有出现在训练文本 T 中，该词串对应的上下文条件概率 $p(w_i \mid w_{i-N+1}, \cdots, w_{i-1})$ =0, 从而导致该词串所在的语句 S 的出现概率 p(S) = 0 。这种情况通常被称作数据稀疏问题 (Data Sparseness) 或零概率问题, 这种问题通常需要进行参数平滑，即对根据极大似然估计原则得到的概率分布进一步调整，确保统计语言模型中的每个概率参数均不为零，同时使概率分布更加趋向合理、均匀。在这里我使用了线性插值平滑，即利用利用低元 N-gram 模型对高元 N-gram 模型进行线性插值。平滑公式为 (λ 取 0.9):

$$p_{\text{interp}}(w_i \mid w_{i-1}) = \lambda \cdot p_{ML}(w_i \mid w_{i-1}) + (1-\lambda) \cdot p_{\text{interp}}(w_i)$$

下面介绍词典生成的相关事项。为了专注于分词算法的实现，Bigram 词典直接使用 Python 内置的 dict 进行存储，通过二重嵌套词典的方式实现 Bigram 概率存取，在词典存储方面，直接使用 python 的 json 模块对成型的 dict 进行存储，这样做方便了算法实现和模型存取，有利于加快分词程序速度和效率。为了方便之后的概率连乘求解，直接将概率转换为对数形式。形成的 Bigram 词典如下图:

图 5: Bigram 词典（json 格式）

3.5.3 Bigram 概率求解原理

之后的部分主要介绍 Bigram 分词的最大概率求解。假设我们已经拥有一个足够大的词典，和待切分的句子 s，分词的目标是将句子 $s = (c_1, c_2, \cdots, c_n)$ 切分为词序列 $s = (w_1, w_2, \cdots, w_m)$, 并且使得 $p(s)$ 最大，通过词典匹配的方法，我们可以将句子分为不同的序列, 这些序列可以统一地用有向无环图 (DAG), 即一个无回路的有向图来表示。例如对于句子

“研究所取得的成就”，生成的 DAG 如下图所示：

图 6: 句子“研究所取得的成就”生成的 DAG

求解该 DAG 可以使用 viterbi 算法。viterbi 算法是一种动态规划算法。它用于寻找最有可能产生观测事件序列的维特比路径——隐含状态序列，特别是在马尔可夫信息源上下文和隐马尔可夫模型中。维特比算法就是求所有观测序列中的最优，如下图所示，我们要求从 S 到 E 的最优序列，中间有 3 个时刻，每个时刻都有对应的不同观察的概率，下图中每个时刻不同的观测标签有 3 个。

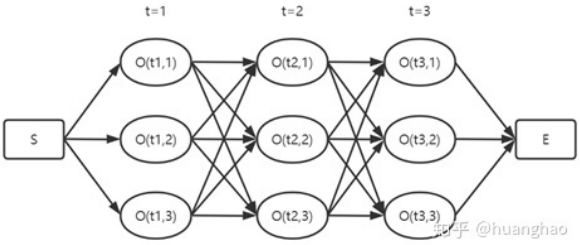

图 7: viterbi 算法 1

求所有路径中最优路径，最容易想到的就是暴力解法，直接把所有路径全部计算出来，然后找出最优的。这方法理论上是可行，但当序列很长时，时间复杂度很高。而且进行了大量的重复计算，viterbi 算法就是用动态规划的方法就减少这些重复计算。

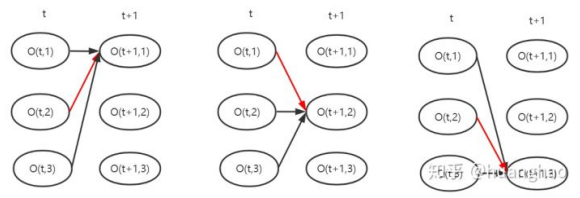

图 8: viterbi 算法 2

viterbi 算法是每次记录到当前时刻，每个观察标签的最优序列，如下图，假设在 t 时刻已经保存了从 0 到 t 时刻的最优路径，那么 t+1 时刻只需要计算从 t 到 t+1 的最优就可以了，图中红箭头表示从 t 时刻到 t+1 时刻，观测标签为 1，2，3 的最优。

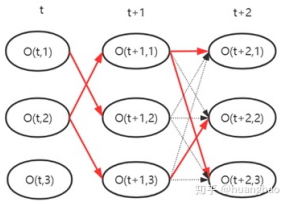

图 9: viterbi 算法 3

3.5.4 viterbi 算法实现

在 viterbi 算法的实现部分，将介绍算法的实现思路和整体流程。通过将长度为 n 的句子的 DAG 图保存在 $n \times n$ 的矩阵 dp 中，我实现了 DAG 图构建和 viterbi 算法的一体化实现，其中概率计算使用对数将连乘变为连加以防止概率下溢为 0。

首先遍历矩阵 dp 中的每一个满足 $0 \leq j \leq i \leq n$ 的单元 dp_{ij} 为每个节点在前驱节点中，选择前驱节点对应的权值与自身权值加和最小的前驱节点，维护一个最优前驱节点与累计的权。

再逆向从其维护的前驱节点，并加入到列表中，最后将列表逆序输出，即为最小负对数概率路径。返回最小负概率对数以及最优分词路径。组合最优分词路径即可得到 Bigram 分词结果

3.5.5 Bigram 效果分析

由于在实现 Bigram 时使用了线性插值平滑的方法，当 λ 为 0 时，Bigram 就退化到 Unigram 了，因而下面主要对 Bigram 和 Unigram 以及之前实现的 FMM 和 BMM 进行分词效果对比分析。通过封闭测试（训练语料和测试语料均为 1998 年 1 月数据），可以得到各分词算法的性能指标如右表所示，可以看到在封闭测试的条件下，虽然 Unigram 各项指标仍低于 FMM 和 BMM，但对于 Bigram 来说，则大大超过了 FMM 和 BMM 和 Unigram，F1 值达到

Algorithm 4 viterbi 前向动态规划

Input: $sentence$: 待分词字符串
$dict$:Bigram 词典

取 $sentence\_length$ 为 $sentence$ 的长度；
初始化动态规划矩阵 dp 和最小前驱结点矩阵 dp_back 为 $sentence\_length$ 维方阵；
for $i = 1$ to $sentence\_length$ **do**
    for $j = 0$ to $i - 1$ **do**
        如果 $sentence[j : i]$ 不存在，continue；
        for 所有可能的前驱词 **do**
            选择前驱节点对应的权值与自身权值加和最小的前驱节点
        end for
        dp 和 dp_back 记录前驱词和概率
    end for
end for

Algorithm 5 viterbi 后向逆推最优路径

Input: dp: 动态规划矩阵 dp_back: 最小前驱结点矩阵 $sentence$: 待分词句
Output: $result$：分词结果

初始化 $length$ 为句子的长度
寻找 $dp[length]$ 中最大概率的位置 pos
利用 dp_back 确定 pos 的上一个位置 $last\_pos$
向 $result$ 中添加由 pos 和 $last\_pos$ 确定的词 $word$
将 $word$ 从 $sentence$ 中删去
while $sentence$ 不为空串 **do**
    $pos = last\_pos$
    利用 dp_back 确定 pos 的上一个位置 $last\_pos$
    向 $result$ 中添加由 pos 和 $last\_pos$ 确定的词 $word$
    将 $word$ 从 $sentence$ 中删去
end while

了 0.998。

表 6: 各模型分词性能指标（封闭测试）

模型	Precision	Recall	F1 值
FMM	0.977	0.961	0.969
BMM	0.979	0.964	0.972
Unigram	0.967	0.961	0.964
Bigram	0.997	0.998	0.998

而 Unigram 和 Bigram 对于开放测试（训练语料为 1998 年 2，3 月数据，测试语料为 1998 年 1 月数据）的性能也很好，如下表，在开放测试上，Unigram 在各项指标上超过了 FMM 和 BMM，Bigram 则是表现得最好。

表 7: 各模型分词性能指标（开放测试）

模型	Precision	Recall	F1 值
FMM	0.931	0.881	0.905
BMM	0.934	0.883	0.908
Unigram	0.908	0.926	0.917
Bigram	0.923	0.958	0.940

从整体看来，Unigram 和 Bigram 在面对实际情况时表现的更好，这是由于 n-gram 模型比机械匹配方法在更深层次的方面考虑了上下文，从而提高了歧义切分等方面的性能。上文中列举的部分 FMM 和 BMM 难以处理的分词串，在 n-gram 模型中都能得到更好的处理。尤其是在封闭测试的情况下，Bigram 取得了 0.998 的 F 值，性能十分良好。

但是值得注意的是，上述所有模型在开放测试的情况下均产生了大幅度的性能下降。这是由于开放测试中面临着大规模的未登录词问题，这些未登录词不能被模型的词典匹配，因而只能被划为单字。在 3.6 节中，我们将会对未登录词进行处理。

还有一点，Bigram 在分词是存在着一些固有的分词错误，这些错误经常在固定的上下文中出现，因而可以应用分词后处理技术对其进行修正，相关问题也会在 3.6 节解决。

3.6 分词结果的再优化

在本节里，我们主要进行了未登录词识别模块的实现，分词后处理和多系统集成的工作来实现性能优化。其中未登录词识别模块涉及 2 阶 HMM（TnT）模型和 Character Based Generative Model 的实现，这两个模型均能进行未登录词识别任务。而分词后处理工作则是主要基于模式匹配的方法进行。多系统集成的工作主要通过对多个分词系统的分词结果通过最少分词法进行词语级系统融合。

3.6.1 2 阶 HMM（TnT）模型

HMM 隐马尔可夫模型是关于时序的概率模型，描述由一个隐藏的马尔可夫链随机生成不可观测的状态的序列，再由各个状态随机生成一个观测从而产生观测序列的过程。HMM 包含如下的五元组：

状态值集合 $Q = \{q_1, q_2, \cdots, q_n\}$，其中 N 为可能的状态数；

观测值集合 $V = \{v_1, v_2, \cdots, v_m\}$，其中 M 为可能的观测数；

转移概率矩阵 $A = [a_{ij}]$，其中 a_{ij} 表示从状态 i 转移到状态 j 的概率；

发射概率矩阵 $B = [b_j(k)]$，其中 $b_j(k)$ 表示在状态 j 的条件下生成观测 v_k 的概率；

初始状态分布 π

一般地，将 HMM 表示为模型 $\lambda = (A, B, \pi)$，状态序列为 I，对应测观测序列为 O。对于这三个基本参数，HMM 有三个基本问题：

概率计算问题，在模型 λ 下观测序列 O 出现的概率；

学习问题，已知观测序列 O，估计模型 λ 的参数，使得在该模型下观测序列 $P(O|\lambda)$ 最大；

解码（decoding）问题，已知模型 λ 与观测序列 O，求解条件概率 $P(I|O)$ 最大的状态序列 I

在中文分词任务中，通常将状态值集合 Q 置为 {B,E,M,S}，分别表示词的开始、结束、中间（begin、end、middle）及字符独立成词（single）；观测序列即为中文句子。比如，“今天天气不错”通过 HMM 求解得到状态序列“B E B E B E”，则分词结果为“今天/天气/不错”。通过上面例子，我们发现中文分词的任务对应于解码问题：对于字符串 $C = \{c_1, \cdots, c_n\}$，求解最大条件概率 $\max P(t_1, \cdots, t_n \mid c_1, \cdots, c_n)$ 其中，t_i 表示字符 c_i 对应的状态。

应如何求解状态序列呢？解决的办法便是 Viterbi 算法；Viterbi 算法本质上是一个动态规划算法，利用到了状态序列的最优路径满足这样一个特性：最优路径的子路径也一定是最优的。定义在时刻 t 状态为 i 的概率最大值为 $\delta_t(i)$，则有递推公式：$\delta_{t+1}(i) = \max[\delta_t(j)a_{ji}]\, b_i(o_{t+1})$ 其中，o_{t+1} 即为字符 c_{t+1}。

在这里实现了二阶 HMM 模型，二阶 HMM 的状态转移依赖于其前两个状态，分词模型如下：

$$\arg\max_r [\prod_{i=1}^{n} P(t_i \mid t_{i-1}, t_{i-2})\, P(c_i \mid t_i)] \times P(t_{n+1} \mid t_n)$$

二阶 HMM 模型的参数估计和求解思路大致和 n-gram 模型思路相似，就不再一一叙述，这里主要说明一下 2 阶 HMM 的参数平缓处理，分词模型采用 TnT(Trigrams'n'Tags) 方法 (Brants, 2000)，对条件概率 $P(t_3 \mid t_2, t_1)$ 采取了如下线性插值平滑（smooth）处理：$P(t_3 \mid t_2, t_1) = \lambda_1 \hat{P}(t_3) + \lambda_2 \hat{P}(t_3 \mid t_2) + \lambda_3 \hat{P}(t_3 \mid t_2, t_1)$。为了求解系数 λ，TnT 提出如下算法，算法中，如果分母为 0 则置式子的结果为 0：

```
set λ1 = λ2 = λ3 = 0
foreach trigram t1,t2,t3 with f(t1,t2,t3) > 0
    depending on the maximum of the following three values:
        case (f(t1,t2,t3)-1)/(f(t1,t2)-1):  increment λ3 by f(t1,t2,t3)
        case (f(t2,t3)-1)/(f(t2)-1):  increment λ2 by f(t1,t2,t3)
        case (f(t3)-1)/(N-1):  increment λ1 by f(t1,t2,t3)
    end
end
normalize λ1,λ2,λ3
```

图 10: TnT 算法 λ 估计

由上面的方法，我们实现了二阶 HMM 分词，而通过将上文 λ_3 设置为 0，就可以得到一阶 HMM 的结果，通过开放测试（训练语料为 1998 年 2，3 月数据，测试语料为 1998 年 1 月数据），我们得到了 HMM 的性能指标：可以

表 8: 各模型分词性能指标（开放测试）

模型	Precision	Recall	F1 值
HMM	0.822	0.821	0.822
2-HMM	0.822	0.820	0.821

看到，在分词指标上 HMM 远低于前文的各种模型，这是因为最大条件概率 P(I|O) 对应的状态序列不一定是分词正确的标注序列。此外，HMM 做了两个基本假设：齐次 Markov 性假设，即任意时刻 t 的状态仅与前一时刻状态相关，与其他时刻的状态、时刻 t 均无关；观测独立性假设，任意时刻 t 的观测仅依赖于该时刻 HMM 的状态，与其他的观测及状态均无关。HMM 受限于这两个假设，而不能学习到更多的特征，泛化能力有限。但是也正是由于这样，HMM 模型所受到的约束条件也较少，在未登录词识别方面有着很强的能力。通过将 FMM 识别出来的词赋以一定的权重加入 Bigram 词典中，Bigram 的性能指标获得了一定的提升。

表 9: Bigram 模型分词性能指标（开放测试）

Bigram 模型	Precision	Recall	F1 值
原模型	0.923	0.958	0.940
加入 HHM	0.938	0.958	0.948

3.6.2 Character Based Generative Model

中文分词的统计学习模型大致可以分为两类：Word-Based Generative Model 与 Character-Based Discriminative Model(Wang and Su, 2009). Word-Based Generative Model 即为我们所熟知的 n-gram 模型。Character-Based Discriminative Model 采用类似与 POS（Part-of-Speech）的序列标注的方法来进行分词，即 HMM 模型。鉴于两种 CWS 模型的利弊：Word-Based Generative Model 高召回登录词、低召回未登录词；Character-Based Discriminative Model 高召回未登录词，低召回登录词。近些年结合两者提出了 Character-Based Generative Model(Wang and Su, 2009)：

$$\arg\max_{t_1^n} P\left([c,t]_1^n\right) = \arg\max_{t_1^n} \prod_{i=1}^{n} P\left([c,t]_i \mid [c,t]_{i-k}^{i-1}\right)$$

这种方法（后面简写为 CBGM 模型）可以实现 n-gram 和 HMM 的一体化结合，实现登录词和未登录词的同时高效识别。

模型的实现和 Bigram 相似，求解方法一样使用 viterbi 算法，区别只在于 Character-Based Generative Model 使用词和词性取代单纯的词来共同作为当前状态。可以看出模型在封闭测试和开放测试 (同前所述) 中均取得了较好的效果。而将模型加入 Bigram 后，可以发现 Bigram 模型也获得了一定的提升

表 10: Character-Based Generative Model 模型分词性能指标

CBGM 模型	Precision	Recall	F1 值
封闭测试	0.946	0.948	0.947
开放测试	0.996	0.997	0.997

表 11: Bigram 模型分词性能指标（开放测试）

Bigram 模型	Precision	Recall	F1 值
原模型	0.923	0.958	0.940
加入 HMM	0.938	0.958	0.948
加入 CBGM	0.942	0.959	0.950
加入 CBGM 和 HMM	0.942	0.957	0.949

3.6.3 分词后处理

由于马尔科夫假设在自然语言语法结构中有些粗糙，统计语言模型的效果总会存在瑕疵，而高阶语言模型虽然能减少这些问题，但实现困难，这就使我们试图寻找基于现有模型的优化方法，一种方法就是进行分词后处理，典型

的做法是基于转换错误驱动的标注学习 (Brill, 1995)。其基本思想为：

（1）正确结果是通过不断修正错误得到的

（2）修正错误的过程是有迹可循的

（3）让计算机学习修正错误的过程，这个过程可以用转换规则（transformation）形式记录下来，然后用学习得到转换规则进行词性标注。

转换规则由两部分组成–改写规则（rewriting rule）–激活环境（triggering environment）。在本实验中基于分词结果的 “BMES” 标注，使用如下形式的转换规则：

$(< c_1, t_1 >< c_2, t_2 >) :< w_3, t_3, t_4 >$。

这表示当前两个字分别为 c_1 c_2，词性为 t_1 t_2 时，将其后的字序列 w_3 的序列标注从 t_3 改为 t_4。

比如 “继续向前迈进” 句，在 Bigram 中往往被分为 “继续/向/前/迈进/”，其状态序列为 “BSSSBS” 而正确答案为 “继续/向前/迈进/”，即 “BSBSBS”，由此我们得到规则 $(<$ 继 $, B ><$ 续 $, S >) :<$ 向前 $, SS, BS >$，在以后再次遇到这种问题，在识别到 “继续” 这两个字以及它们的状态 BS 后，便通过将 “向前” 的状态序列由 SS 转为 BS 解决。在经验上，这种方法没有发生过拟合

构建转换规则的学习器算法如下：

Algorithm 6 转换规则学习器算法

Input: C_0: 已被模型标注好的语料结果
C_1: 人工正确标注的语料
Output: T: 转换规则序列

for C_0 中的每一条语句 s_0 **do**
  获得 s_0 在 C_1 中对应的语句 s_1；
  获得 s_0 和 s_1 的状态序列 T_0，T_1；
  if T_0 与 T_1 存在差异 **then**
    识别出其中的转换规则 T_{01}；
    将 T_{01} 加入 T 中；
  end if
end for

与之对应的，有应用转换规则序列进行分词后处理的算法。

Algorithm 7 分词后处理算法

Input: C_0: 已被模型标注好的语料结果
T: 转换规则序列
Output: C_1: 经过转换器处理的语料结果

for C_0 中的每一条语句 s_0 **do**
  获得 s_0 的状态序列 T_0；
  for T 中的每一条规则 t **do**
    if t 与语句 s_0 存在匹配部分 **then**
      应用 t 转化语句 s_0，得到结果 s_1
      将 s_1 写入 C_1
    else 将 s_0 写入 C_1
    end if
  end for
end for

于是我们在开放测试的条件下对其进行验证，可以发现模型性能有了进一步提升。

表 12: Bigram 模型分词性能指标（开放测试）

Bigram 模型	Precision	Recall	F1 值
原模型	0.923	0.958	0.940
经过后处理	0.933	0.959	0.945

3.6.4 多系统集成

在前面的实验中我们实现了基于 Bigram 模型的分词系统，基于 2 阶 HMM 的未登录词识别，基于 Character Based Generative Model 的分词未登录词识别一体化模型和分词后处理系统。Bigram 模型和未登录词识别模块（HMM 或 Character Based Generative Model）的结合是通过将未登录词识别模块的分词结果以一定的权重加入 Bigram 词典实现的。因而我们现在有着 Bigram+HMM 和 Bigram+CBGM 和单纯的 CBGM 这三套分词系统，及相应训练得到的各自的后处理规则集。现在的问题是通过多系统集成的方式进一步提升系统性能。

本节通过最少分词原则来进行多系统融合工作。通过几个分词系统的结果构建词网格，

在词网格中搜索最短路径的方法实现词语级系统融合。由于此项操作等价于在边权重恒为 1 的 Unigram 词网格中进行搜索，而 Unigram 的实现已在前文给出，因而在这里不在给出实现。

下面进行多系统融合的性能分析。通过将 Bigram+HMM，Bigram+CBGM 两个模型按最少分词方法融合，我们可以得到二重融合系统 BHBC，通过将 Bigram+HMM，CBGM 两个模型按最少分词方法融合，我们可以得到二重融合系统 BHBC，通过将 CBGM，Bigram+CBGM 两个模型按最少分词方法融合，我们可以得到二重融合系统 BC。将系统 BHBC 再和 CBGM 融合，得到了三重融合系统 BHC。在开放测试下有如下结果。可以发现经过系统融合，模型

表 13: 融合模型分词性能指标对比（开放测试）

模型	Precision	Recall	F1 值
Bigram	0.923	0.958	0.940
CBGM	0.946	0.948	0.947
Bigram+HMM	0.938	0.958	0.948
Bigram+CBGM	0.942	0.959	0.950
二重融合 BHBC	0.944	0.953	0.948
BHBC+ 后处理	0.944	0.954	0.949
二重融合 BH+C	0.958	0.951	0.954
BH+C+ 后处理	0.959	0.951	0.955
二重融合 BC	0.959	0.955	0.957
BC+ 后处理	0.959	0.955	0.957
三重融合 BHC	0.958	0.950	0.954
BHC+ 后处理	0.959	0.951	0.955

的性能有了更进一步的提升。

4 结论

4.1 机械匹配分词

在本次实验中，首先实现了基于机械匹配的分词模型 FMM 和 BMM。在这项工作里，我们通过读取语料库形成 Trie 树存储的分词词典，并对分词词典做了详细的分析，可以发现，形成的分词词典较为合理，可以满足接下来的分词需求。接下来我们基于此分词词典使用简单的正则表达式匹配进行了 FMM 和 BMM 模型的建立，并据此实现了对测试语料的分词操作。

在完成了分词后，首先要进行的是对分词结果进行详细的评价。我们通过将机器学习常见的评价指标 Precision，Recall 和 F 值在中文分词中进行适应化，实现了中文分词自动评价工作。当然仅依靠这三个指标是不够的，因而我们从句子的角度，结合具体语句的划分错误对 FMM 和 BMM 的性能进行了分词结果评测。在综合评价后，我们发现，BMM 模型在分词性能上要好于 FMM 模型。这之中的主要原因有两点：一是汉语构词倾向在前后向的差异，新词倾向于在已有词后添加字词形成新词。二是汉语语法的倾向性，偏正结构等重点在后的句子结构使 BMM 的划分结果错误路径更少，不容易落入陷阱。

在理论分析后，我们对模型的在实际应用中的能力进行了优化，主要是分词的速度方面。通过使用相关分析工具，我们发现制约程序速度的瓶颈是查找算法的实现，即 Trie 树在实现时使用了顺序查找的方法。通过将顺序查找改为 Hash 表查找，我们将查找时间复杂度降为 O（1），程序运行时间从 5 个多小时减少到 3-7 分钟，时间性能的极大提升为模型的部署提供了条件。最后还进行了模型进一步改进的展望，可以通过用 C 语言重写计算模块和更换 Hash 函数的方法进一步改进效率。

4.2 统计语言模型分词

在机械分词模型后，我们实现了基于统计语言模型的分词。相对于机械分词，统计语言模型能够在分词时更好的使用上下文信息，取得更好的效果。在这部分工作中我们实现了 Bigram 分词。我们首先基于训练语料，对 Bigram 分词概率进行极大似然估计，同时运用线性插值的方法，平滑了模型参数。在 Bigram 平滑参数 λ 为 0 时，模型退化到 Unigram 模型。基于得到的概率词典，我们通过对给定的句子进行 DAG 词网格构建的方法完成对句子所有分

词可能的建模，进而可以使用 viterbi 算法进行最大概率求解，得出最大概率的句子，完成分词。通过模型评价，我们发现，统计模型在封闭测试和开放测试中均好于之前实现的 FMM 和 BMM 模型，特别是在封闭测试中，Bigram 达成了 F 值 0.998 的成绩，而在开放测试中，Bigram 只达到了 0.940 的 F 值。这说明，一方面，在已有相关分词数据经验时，Bigram 能更完全地基于上下文进行分词；另一方面，在遇到未登录词即未登录词组合时，Bigram 的性能就会剧烈地下滑。另外，Bigram 模型本身固有的缺陷——即不能完全的建模语言现象——也使其在封闭测试时不能达到 100% 的正确性能。

4.3 分词性能优化

为了解决 Bigram 遇到的上述问题，我们对分词模型进行了进一步的优化。首先对于 Bigram 未登录词问题，我们采用加装未登录词识别模块的方案来解决。机器学习理论中存在两种方法，生成式方法和判别式方法。在未登录词识别中，判别式方法表现为基于 HMM 的序列标注方法（Character-Based Discriminative Model），生成式方法表现为基于 Character Based Generative Model 的序列标注方法。两种方法均可以进行未登录词识别，且 CBGM 方法整体性能优于 HMM 方法。在加装两种模型后 Bigram 模型的性能获得了提升。

为了解决 Bigram 模型的固有问题，有两种思路。一种是提升 n-gram 模型的 n 为更高，然而这种方法较为复杂，难以实现。另一种思路为对 Bigram 分词的错误进行基于转换错误驱动的标注学习，通过修正错误的方法获得性能提升。在实际应用中，第二种思路更为实用，通过分词后处理，Bigram 模型的性能获得了提升。

最后，我们对实现的各个模型进行了词语级系统融合。在最少分词原则下，通过动态规划求解最短路径的方法，获得了几个融合系统。实践表明，通过系统融合，模型获得了更好的分词性能。

5 未来工作

基于统计语言模型 n-gram 的工作在经验充足（训练语料覆盖）的情况下能很好的进行模型分词，但是面对未登录词就显得效果不太好了。Bigram 在反复优化后提升效果越来越小，在本实验的开放测试中 F 值仍然无法突破 0.96 的界限。未来的工作其一可以寻找更大的训练语料库训练模型，其二可以尝试更换模型。目前公认的最好的分词模型要数 CRF（条件随机场）模型，可以尝试通过 CRF 模型进行进一步的分词。或者使用更高级的机器学习模型，如近年来兴起的深度学习模型，也是一条可行的方向。

参考文献

Thorsten Brants. 2000. Tnt: a statistical part-of-speech tagger. *Proceedings of ANLP-2000, Seattle, WA*.

Eric Brill. 1995. Transformation-based error-driven learning and natural language processing: A case study in part-of-speech tagging. *Computational Linguistics*, 21(4):543–565.

GMFTBY. 2016. Elfhash - 优秀的字符串哈希算法. *CSDN* 社区.

Chengqing Zong Wang, Kun and Keh-Yih Su. 2009. Which is more suitable for chinese word segmentation, the generative model or the discriminative one? *PACLIC*.

北京大学计算语言学研究所. 1999. 现代汉语语料库加工规范——词语切分与词性标注.

孙茂松，邹嘉彦. 2001. 汉语自动分词研究评述. 《当代语言学》.

宗成庆. 2013. 《统计自然语言处理》, 2 edition. 清华大学出版社.