

**实验报告**

**实 验（一）**

题 目 汉语分词系统

专 业 自然语言处理

学　　 号 1161000309

班　　 级 1603104

学 生 高靖龙

指 导 教 师 杨沐昀

实 验 地 点 G208

实 验 日 期 10.9.2018-10.30.2018

**计算机科学与技术学院**

**目 录**

[第1章 实验基本信息 - 3 -](#_Toc529040202)

[1.1 实验目的 - 3 -](#_Toc529040203)

[1.2 实验环境与工具 - 3 -](#_Toc529040204)

[1.2.1 硬件环境 - 3 -](#_Toc529040205)

[1.2.2 软件环境 - 3 -](#_Toc529040206)

[第2章 实验内容 - 4 -](#_Toc529040207)

[2.1 词典的构建 - 4 -](#_Toc529040208)

[要求 - 4 -](#_Toc529040209)

[PFR语料库格式 - 4 -](#_Toc529040210)

[隔断正则表达式 - 4 -](#_Toc529040211)

[字典文件格式说明 - 4 -](#_Toc529040212)

[字典统计分析 - 4 -](#_Toc529040213)

[字典内标点符号统计 - 5 -](#_Toc529040214)

[2.2 正反向最大匹配分词实现 - 5 -](#_Toc529040215)

[内容 - 5 -](#_Toc529040216)

[实现心得 - 5 -](#_Toc529040217)

[2.3 正反向最大匹配分词性能分析 - 6 -](#_Toc529040218)

[内容 - 6 -](#_Toc529040219)

[实现思路 - 6 -](#_Toc529040220)

[实现心得 - 6 -](#_Toc529040221)

[FMM与BMM分词精度差异分析 - 7 -](#_Toc529040222)

[2.4基于机械匹配的分词系统的速度优化 - 8 -](#_Toc529040223)

[内容 - 8 -](#_Toc529040224)

[优化方案 - 8 -](#_Toc529040225)

[2.5 基于统计语言模型的分词系统实现 - 10 -](#_Toc529040226)

[内容 - 10 -](#_Toc529040227)

[前缀词典 - 10 -](#_Toc529040228)

[全切分有向图DAG - 10 -](#_Toc529040229)

[UnigramModel - 11 -](#_Toc529040230)

[BigramModel - 12 -](#_Toc529040231)

[性能对比 - 13 -](#_Toc529040232)

[参考文献 - 14 -](#_Toc529040233)

# 第1章 实验基本信息

## 1.1 实验目的

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、性能评价和优化等环节。本次实验所要用到的知识如下：

* 基本编程能力（文件处理、数据统计等）
* 相关的查找算法及数据结构实现能力
* 语料库相关知识
* 正反向最大匹配分词算法
* N 元语言模型相关知识
* 分词性能评价常用指标

## 1.2 实验环境与工具

### 1.2.1 硬件环境

处理器： Corei7

内存： 8G

系统类型： 64位

### 1.2.2 软件环境

系统： manjaro gnome linux1.80(主)、win10(辅)

IDE： jetbrains pycharm

文本编辑器： noteapp、atom、vim

编程语言： python3

# 第2章 实验内容

## 2.1 词典的构建

### 要求

输入文件：199801\_seg.txt（1998 年 1 月《人民日报》的分词语料库，有版权限制！）

输出：dic.txt（自己形成的分词词典）

### PFR语料库格式

PFR语料库标注词性的格式为“词语/词性”，语料中除了词性标记以外，还有“短语标记”，以方括号括住若干 “词语/词性”对，紧跟“/短语类别”。每行开头是时间标识以“/m”结尾，然后是空格，之后是该行句子的词性、短语标记。

### 隔断正则表达式

"^.\*?/m +|/[a-zA-Z\]]+ \*|\["

### 字典文件格式说明

每行一个记录，为“词 词频”，即词+空格+词频，按词频从大到小排列。

### 字典统计分析

字典总长度：55310

词长分布：不同长度的词在字典中出现的频率

### 字典内标点符号统计

11种标点符号： ， 。 、 （ ） ？ ！ ——— …… — ——

## 2.2 正反向最大匹配分词实现

### 内容

输入文件：199801\_sent.txt（1998 年 1 月《人民日报》语料，未分词）

dic.txt(自己形成的分词词典)

输出：seg\_FMM.txt 和 seg\_BMM.txt(正反向最大匹配分词结果，格式参照分词语料)

### 实现心得

1. **字符串索引**

编写实现中，对于字符串的索引str[i,j]的变化要清晰，特别是BMM，实现上较为反直觉。

1. **最大词长截断**使用字典最大词长截断字符串，能得到可接受的速度，如果不截断的话则无法处理长串。
2. **匹配失败**

如果某个位置匹配失败，即没有任何字典词以它为前缀（FMM）或后缀（BMM），则要将其作为一个独立的词。

1. **F/B相似**FMM\BMM两者结构非常相似，区别主要集中在循环方向以及匹配成功一个词后的匹配目标更新。
2. **BMM分词逆序**BMM的分词结果不能直接输出，还要进行顺序的倒置，这是因为BMM是由后向前匹配查词的。

## 2.3 正反向最大匹配分词性能分析

### 内容

输入文件：199801\_seg.txt（1998 年 1 月《人民日报》的分词语料库）

seg\_FMM.txt、seg\_BMM.txt

输出：score.txt(包括准确率（precision）、召回率（recall）， F 值的结果文件)

### 实现思路

1. TP+FP=len（tarseg）,TP+FN=len（corpusseg）

故对于precision、recall、F，我们只需要统计TP。

1. 分别从corpus、targetsegfile中读取分词序列，存储到clist，tlist中。
2. 使用两个指针i、j同步的在两列表上移动，并统计移动到当前位置的**总字数差值ct。**ct=clist[：i]的总字数-tlist[：i]的总字数。
3. ct为0则匹配成功，TP+1。否则说明当前分词失败，进入重定位模式。
4. 重定位模式负责重新找到下一个成功分词的开头，即下一个ct为0的地方。实现上即当ct>0时说明corpus超前，就移进j；反之移进i。

### 实现心得

1. 一个python容易惯性思维写错的地方：

*#****TODO 特别容易写错的地方！下面是错误实例****#if word == '' or '\n':***if** word == **'' or** word==**'\n'**:  
 **continue**

在判断语句中，’\n’单独出现时被作为True判定。则循环永远进入，continue语句被执行，跳过其后的代码。

1. 实现细节上，对于ct的更新不能简单的更新为  
   ct += len(corpusseg[i]) - len(targetseg[j])

因为在重定位阶段，一次只移动i、j之一，这样就将旧的词语新词对比，不符合设计初衷。解决方案是针对i、j重定位中移动的情况区别处理。

*# 使用flag区分匹配、重定位j、重定位i这三种情况*

flag=0  
*# 移进的字符数量差corpos-target*ct=0  
  
**while True**:  
 **if** i==lcorpus **or** j==ltarget:  
 **break  
  
 if** flag==0:  
 ct += len(corpusseg[i]) - len(targetseg[j])  
 **elif** flag==1:  
 ct -= len(targetseg[j])  
 **else**:  
 ct += len(corpusseg[i])  
 **if** ct==0:  
 **if** flag==0:  
 TP+=1  
 i+=1  
 j+=1  
 flag = 0  
 **elif** ct>0:  
 j+=1  
 flag = 1  
 **else**:  
 i+=1  
 flag = 2

1. 实现中做了两个版本，第一个版本不是基于list的而是基于字符串的，看起来非常不直观（基于list的已经非常不直观了！）所以写了第二个版本，第一个版本作为一个对比函数使用。

### FMM与BMM分词精度差异分析

1. 分词精度统计结果

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **TP** | **TP+FP** | **TP+FN** | **P** | **R** | **F** |
| **FMMSEG** | 1088311 | 1113018 | 1121447 | 0.977802 | 0.970452 | 0.974113 |
| **BMMSEG** | 1090501 | 1112991 | 1121447 | 0.979793 | 0.972405 | 0.976085 |
| **FMMBMM** | 1083746 | 1113018 | 1112991 | 0.973700 | 0.973723 | 0.973712 |

BMM性能对于FMM有提升：分词数减少、TP增多。

1. 精度差异分析
   1. 理论差异  
      FMM和BMM的差别在于查询的方向不同，即FMM倾向前缀匹配而BMM倾向于后缀匹配。
   2. 理论差异对分词的影响  
      实验中由于训练集和测试集是相同的，所以不存在集外词，因此性能影响就主要在于分词歧义。

对于无歧义的词显然FMM、BMM没有差别，对于组合型的词FMM和BMM都不分割。但是对于交集型歧义，如ABC，FMM倾向：AB C，而BMM倾向A BC。  
因此性能的差异是出现在对集中型歧义的分割上。

* 1. 对词分布的猜测  
     由于BMM精度较高，猜测样本中集中型歧义中A BC出现概率更高。
  2. 分析分布  
     实际上BMM与FMM算法的运行过程就是对集中型歧义的探究。

通过统计即可分析出前缀集中、后缀集中、组合歧义、在seg被分割的组合歧义、在seg中没有分割的组合歧义在文本的占比。

SEG=无歧义+后缀集中+前缀集中+非切分组合歧义+切分组合型歧义=1121447

BMMSEG.TP=无歧义+后缀集中+非切分组合歧义=1090501

FMMSEG.TP=无歧义+前缀集中+非切分组合歧义=1088311

SEG-BMMSEG.TP=切分组合型歧义+前缀集中 =30946  
SEG-FMMSEG.TP=切分组合型歧义+后缀集中=33136  
FMMBMM.TP =无歧义词+非切分组合型歧义=1083746

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **无歧义+非切分组合** | **前缀集中** | **后缀集中** | **切分组合** | **合计** |
| 1083746 | 4565 | 6755 | 26381 | 1121447 |
| 0.966381 | 0.004070 | 0.006023 | 0.023524 | 1 |

对上述等式进行变换：

* 1. 结论：由于在文本中集中歧义划分时，A BC型比AB C型占比更多，所以后项匹配比前项匹配性能更好。

## 2.4基于机械匹配的分词系统的速度优化

### 内容

输入文件：199801\_sent.txt（1998 年 1 月《人民日报》语料，未分词）

输出：timeCost.txt（分词所用时间）

### 优化方案

1. 截断优化查询
   1. 字典最大词长截断

对于一个字典，若词长最大为maxL，则我们在FMM、BMM是，查询的词长最大也应是maxL。

* 1. 字典词长截断  
     受最大词长截断的启发，我们存储字典中出现的词长L1，L2，，Ln，我们只对目标文本中长度在这些范围中的词进行查询。
  2. 性能提升分析  
     **词长分布的稀疏性**

我们之前对于字典进行过分析，可知字典的词长为1-26，因此空间代价可以忽略不计。同时词长19、20、21、24、26没有出现，因此循环次数比最大词长截断减少20%。  
**可选的进一步筛选**  
我们看到有的词长在字典中的词频实际上非常少，我们可以直接省去这些词长，而不会带来很大的准确性损失。  
但是即使不这么做，使用简单的最大词长截断，算法总运行时长仍然能控制在3min内（包括FMM、BMM、IO、准确性分析、计时函数）。

1. 散列表优化：一种时空的tradeoff  
   使用散列表对查询速度进行优化，是一种时空的tradeoff。
   1. 算法描述  
      **散列**  
      散列表算法使用散列函数对内容进行散列，将散列结果作为数组索引，将内容存储到对应数组索引处。因此数组并不是完全填满的，会出现冗余的空间。  
      **冲突**由于散列函数通常不满足单射，因此会出现多个内容的散列相同，对同一数组索引竞争，这种现象成为散列冲突。  
      **冲突的解决**  
      本次实现使用线性开放定址法，即冲突发生是，线性寻找下一个可用位置存储内容。负载因子取0.5-0.75之间，经测试超出该范围性能提升不显著。
   2. 性能提升分析

散列表拿空间换时间，所以能提升性能，特别的是需要设计好散列函数，散列函数决定hashlist的性能。  
这里参考java.String.hashCode()的实现对字符串进行散列。

* 1. 算法实现  
     核心的散列函数，经典实现。  
     31作为乘数的原因：  
     cpu能通过位移来优化计算。31\*t=t<<5-1  
     31散列具有统计上的优势。

**def** getHashCode(words):  
 *"""  
 散列函数。参考java.string.hashCode()实现* **:param** *words: 词，字符串* **:return***: 散列值  
 """* s = 17  
 **for** c **in** words:  
 s = 31 \* s + ord(c)  
 **return** s

## 2.5 基于统计语言模型的分词系统实现

### 内容

输入文件：test\_sent.txt（1998 年人民日报局部语料，未分词，最终测试集）

dev\_seg.txt（1998 年人民日报局部语料，分词，用于调试优化语言模型） 199801\_seg.txt

dic.txt（自己形成的分词词典）

输出：seg\_LM.txt（利用统计语言模型分词结果，格式参照分词语料）

须对程序中的重点实现代码进行说明（可用流程图对算法进行

辅助说明）；对比分析各种不同分词方法的性能；

### 前缀词典

1. 算法原理  
   前缀词典：元词典记录词和词频，将元词典中词语的所有前缀词都添加词典中，如果前缀词没有在原词典出现则其频率设为0，若有则设为原词频。
2. 算法实现  
   简单的对每一个词，反向遍历，提取出前缀添加到词典中,并设置词频。

### 全切分有向图DAG

1. 算法原理  
   对于字符串s，考察其每个字s[k]，对于s[k:i+1]，i=0,1,2..,依次到前缀词典lfreq中查找，若查到且频率非零，则对k存储i，若没有查到则说明没有以s[k]为开头的词在lfreq查尽了。最终得到每个k的一组i，s[k:i+1]表示一个可能分词。
2. 算法实现  
   DAG存储：dict{k : [k , j , ..] , m : [m , p , q] , ...},s[k:j+1]表示一个可能分词。  
   按上述原理编写循环分支。
3. 实现细节  
   可能存在对于某个字s[k]，在lfreq中没有其前缀词，则此时也必须给出分词，将该字本身s[k:k+1]作为一个分词存入DAG。

### UnigramModel

1. 算法原理  
   在DAG的基础上，动态规划寻找1-GramModel最大概率分词序列。
   1. **概率公式**
   2. **动态规划转移方程**

定义问题L(j)为sentence[j:]的1-GramModel最大概率分词序列。

* 1. **重叠子问题**若DAG[j1]、DAG[j2]均包含i0则子问题重叠。

1. 算法实现  
   路径存储：route=[]，route[idx词首索引]=(子问题概率，词尾索引)

*#初始化，该实现中所有值均log化，有效防止下溢出*route[N] = (0, -1)  
logtotal = log(ltotal)  
**for** idx **in** range(N - 1, -1, -1):

*动态规划转移方程：getFreq对概率进行了加一平滑* route[idx] = max((log(getFreq(lfreq,sentence[idx:x + 1])) - logtotal + route[x + 1][0], x) **for** x **in** DAG[idx])

### BigramModel

1. 算法原理  
   在DAG的基础上，动态规划寻找2-GramModel最大概率分词序列。
   1. **概率公式**
   2. **动态规划转移方程**

定义问题组,

转移方程

* 1. **重叠子问题**若DAG[j1]、DAG[j2]均包含i0则子问题重叠。

1. 算法实现
   1. **路径存储：**routeDAG=[[]]，routeDAG[idx]=[,,,]
   2. **平滑算法：**加一平滑
   3. **初始化：**

routeDAG[N - 1] = []  
routeDAG[N - 1].append((N - 1, **"TAIL"**, 1))

* 1. **动态规划迭代：**

value, y = max((getPYX(sentence[idx:x + 1], sentence[x + 1:y + 1], dict) \* v, y) **for** y, m, v **in** routeDAG[x + 1])

routeDAG[idx].append((x, y, value))

* 1. **二元条件概率字典构造：**  
     根据文件，获取包括^$的二元概率列表

格式：

dict{条件词:{(目标词1,概率),(目标词2,概率),,,(目标词n,概率)}}  
 遍历seg文件，统计条件概率，对于所有频率均+1，实现平滑算法。

* 1. **分词结果构造**由reouteDAG[0]开始递归查询，获取索引形式的分词形式，之后将该序列转换成词序列。

### 性能对比

1. 集内性能

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | TP | Pall | Tall | P | R | F |
| FMM | 1088311 | 1113018 | 1121447 | 0.9778 | 0.9704 | 0.9741 |
| BMM | 1090501 | 1112991 | 1121447 | 0.9797 | 0.9724 | 0.9760 |
| Bigram | 1088308 | 1113019 | 1121447 | 0.9778 | 0.9704 | 0.9741 |
| Unigram | 1103483 | 1113484 | 1121447 | 0.9910 | 0.9839 | 0.9874 |

* 1. 性能排序：在本次实验中，Unigram>BMM>FMM≈Bigram
  2. 性能分析：为什么Bigram性能表现不如Unigram？
     1. Bigram概率空间比Unigram大一个数量级，数据量 可能对Unigram还算充足但是在Bigram中已经稀疏了。
     2. 平滑算法选择的较为简单，导致Bigram不能更好的反映语言分布。使用Good-Turing smoothing或简单的调整加x平滑法中的x都能带来小幅度的提升。
     3. 在计算概率过程中程序可能存在问题，但是暂时没有排查到。

1. 集外性能

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | TP | Pall | Tall | P | R | F |
| FMM | 3387 | 3948 | 3734 | 0.857903 | 0.90707 | 0.881802 |
| BMM | 3383 | 3948 | 3734 | 0.85689 | 0.905999 | 0.88076 |
| Bigram | 3384 | 3949 | 3734 | 0.856926 | 0.906267 | 0.880906 |
| Unigram | 3436 | 3957 | 3734 | 0.868335 | 0.920193 | 0.893512 |

针对语料库，分割出训练集和测试集，在测试集上性能排名和集内性能排名差别不大。

# 参考文献

[1]Thomas H.Cormen．Introduction to Algorithms[M]．北京：机械工业出版社，2013：142-160

[2]Christopher D. Manning，Hinrich Schutez．统计自然语言处理基础[M]．北京：电子工业出版社，2005：1-289

[3]周志华．机器学习[M]．北京：清华大学出版社，2016：23-46

[4]宗成庆．统计自然语言处理 (2nd ed.) [M]．北京：清华大学出版社，2013：1-289

[5] Python3.7.1 Documentation. Docs.python.org. https://docs.python.org/3/, November 3, 2018.