

# 自然语言处理技术 入门与实战

兰红云◎编著

**实用技术要点:** 语义模型详解、自然语言处理系统基础算法和系统案例实战

**完整解决方案:** 问题解决的原理、实现的算法原理、具体算法的实现

# 自然语言处理技术 入门与实战

兰红云◎编著

电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

## 内 容 简 介

本书主要从语义模型详解、自然语言处理系统基础算法和系统案例实战三个方面，介绍了自然语言处理中相关的一些技术。对于每一个算法又分别从应用原理、数学原理、代码实现，以及对当前方法的思考四个方面进行讲解。

本书面向的读者为有志于从事自然语言处理相关工作的在校学生、企事业单位工作人员等人群。本书的结构特点是由浅入深地进行相关内容的介绍，以满足不同层次读者的学习需求。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。  
版权所有，侵权必究。

### 图书在版编目(CIP)数据

自然语言处理技术入门与实战 / 兰红云编著. —北京: 电子工业出版社, 2017.10  
ISBN 978-7-121-32763-6

I. ①自… II. ①兰… III. ①自然语言处理—研究 IV. ①TP391

中国版本图书馆 CIP 数据核字(2017)第 233723 号

策划编辑: 张慧敏

责任编辑: 牛 勇

特约编辑: 顾慧芳

印 刷:

装 订:

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱 邮编: 100036

开 本: 787×980 1/16 印张: 12.5 字数: 280 千字

版 次: 2017 年 10 月第 1 版

印 次: 2017 年 10 月第 1 次印刷

定 价: 59.00 元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888, 88258888。

质量投诉请发邮件至 [zltz@phei.com.cn](mailto:zltz@phei.com.cn), 盗版侵权举报请发邮件至 [dbqq@phei.com.cn](mailto:dbqq@phei.com.cn)。

本书咨询联系方式: 010-51260888-819, [faq@phei.com.cn](mailto:faq@phei.com.cn)。

# 前 言

随着移动互联网的飞速发展，特别是物联网（Internet of Thing, IoT）的飞速发展，人与设备的语言交互场景也越来越多，并且越来越成为核心。这种语言的交互既包括纯文字的，也包括语音的。自然语言处理（Natural Language Processing, NLP）就是以电子计算机、编程语言为工具对人类特有的书面和口头形式的自然语言信息进行各种类型处理和加工的技术。当然，随着技术的不断发展，其处理领域也出现了跨形态的组合。比如通过与图形图像处理技术的结合，可以实现看图说话、在线答题等应用。所以，自然语言处理是一门涉及语言学、计算机科学，当然还有数学的交叉性科学。

自然语言处理的目的是为了让计算机能够理解人的语言，然后做出相应的处理或者应答。根据应用场景的不同，自然语言处理可以分为如下三点：（1）信息抽取，包括自动摘要、自动检索、舆情分析等；（2）语言理解，包括机器翻译、人机对话、语义理解等；（3）跨形态组合，包括看图说话、语音自动合成、辅助教学等。这些应用都是利用自然语言处理技术，对所需要处理的信息进行挖掘和分析，找出人们想要的东西，进而作出响应。而落实到具体的应用，又会衍生出很多不同的应用系统，由此衍生出来的应用系统包括但不限于：信息自动抽取系统、信息自动检索系统、文本信息挖掘系统、机器翻译系统、人机对话系统、图片描述自动生成系统、语音自动识别系统、语音自动合成系统、计算机辅助教学系统，等等。

因为自然语言自身的复杂性，比如：很多歧义、结构复杂多样、表达千变万化……导致其处理方法纷繁复杂，要考虑非常多的情景。所以上述这些系统之间又存在交叉，或者上下关联，或者前后依赖等复杂的关系。而这些复杂的应用对于一个初学者来说，是非常庞杂和难以掌握的，在学习的过程中难免会因为其中某一个细节不能掌握，而影响整个进程的进度；或者虽然理解了算法的数学原理，但是怎么在实际场景中应用，以及当前算法能解决哪些实际问题，还是不了解。在笔者学习的过程中，发现目前出版的一些书籍，或

者是偏理论性的，会介绍很多自然语言处理技术发展的历史，比如符号逻辑的发展轨迹、语义网络的发展轨迹、语言学派和统计学派的“恩怨情仇”，会让初学者在学习的过程中抓不住重点，有时候又感觉它们好像就是一回事；又或者介绍的内容过于偏技术，开篇就把其中涉及的一些数学知识全都介绍一遍，因为这其中有很多数学知识是比较高阶的，比如隐马尔科夫链、条件随机场、数理逻辑推理等，在介绍数学知识的过程中，又难免会涉及相关的证明。本来其数学形式就比较复杂，再加上连环的证明就更难懂了，对于数学基础稍微薄弱一点的读者，就感觉没有学习的欲望和必要了。但是在实际应用中，其实这些烦琐的证明根本不需要，有时候只需要记住一个结论，然后根据自己数据的情况，优化模型中的参数就可以了。所以笔者就想结合自己学习过程中和实际工作中的一些经验和教训，从应用的角度来对自然语言处理中的一些技术进行介绍。在介绍的时候，为求尽量避免烦琐和突兀的数学证明，从应用的角度尽可能简洁明了地对一个算法或者处理系统进行简要的介绍，先让大家对这个方法有一个直观感性的认识，然后再深入了解其中的难点，进而深入学习和攻克难点。

本书采用以应用为主、算法和实现为辅的形式对自然语言处理中的一些技术进行介绍。对于算法数学原理的介绍，都是穿插在每一个应用的介绍中，对每一部分的数学知识进行分别介绍和讲解，没有开篇便对所涉及的数学知识进行一个全面的介绍，这样大家就不会因为某一个部分的数学知识不完备，或者掌握起来有困难而放弃整个知识体系的学习，这样大家就可以独立学习和掌握。同时因为知识遗忘的必然性，笔者将数学知识融入应用中进行介绍，就更容易让读者记住。否则前后脱节之后，就忘记了之前讲解的数学原理，即使在应用中又要重新学习，也并不一定能够知道具体的应用原理。

因此，笔者完全从应用的角度来进行各个内容的组织，没有涉及太多的处理技术起源、变革、发展等历史信息。这一方面是因为各个技术都有自己的长处和缺点，这个是理论学派争论的焦点，但不是应用层面应该关心或者需要表明立场的地方；另一方面作为主要介绍应用实战的书，这里更多的是想让读者了解对于同一个问题目前的一些处理方法和这些方法之间的优劣，以及相互的关联，以便找到解决问题更好的方法，这样也更有利于整个事情的发展。所以从做事的本身来说，我们需要关心的是事情怎么能够做起来，没有做起来是因为什么，所以我们更多关心的是“术”的事情，而对于“道”的层面更多的是了解，是取众家之长，来“集大成”，而不能剑走偏锋。

目前，随着源工具的不断增多，大家对底层应用的开发需求在逐渐降低，所以本书先从上层应用介绍入手，让读者能够直接用起来，这样更有利于读者边实践边学习，也可以避免大家因为学习底层技术太难而阻碍后期应用的学习。从企业的角度出发，缺的也不是底层通用的处理技术和能力，更多的是缺少对实际业务的处理能力，业务跑起来之后，整个系统便会随着业务的发展而不断发展。所以本书采用以应用贯串始终的方式来进行相关技术的介绍和说明。

具体来说，本书主要从以下三个方面介绍了自然语言处理中相关的一些技术。

- 1) 语义模型详解：主要是从应用的角度介绍自然语言处理中的一些语义处理模型，比如关键词提取、计算词距离、文本自动生成等。
- 2) 自然语言处理系统基础算法：这一部分主要是从基础系统搭建的角度对相关算法进行介绍。包括分词、词性标注、句法分析等。这两部分介绍的内容又分别从使用原理、实现原理、具体的代码实现，以及对当前方法的思考这四个角度进行介绍。
- 3) 系统案例实战：介绍了搭建一个舆情分析和挖掘系统所要涉及的环节、各个环节的算法实现，以及部分实现代码。

本书在写作过程中力求普及并与实践相结合，尽可能地照顾到不同层次不同专业的读者。另外，本书是以应用场景来组织各个内容的，每一个章节都包含一个完整的应用解决方案：问题解决的原理、实现的算法原理、具体算法的实现，所以读者可以根据自己的需要独立地学习各个章节的内容。在各个章节的学习过程中，笔者强烈建议读者在学习具体方法之前，一定要认真地理解所要解决问题的具体场景。要理解当前场景的输入是什么、输出是什么，为什么会是这样的结构，只有弄明白了这些，才会对算法有更深入的理解，也才能更好地使用所学习的算法，做到举一反三。因为算法本身是一种数据处理逻辑，所以只要具有相同处理逻辑的问题都可以用同样的算法，比如最大熵模型发挥了巨大的作用是人们找到了其适用的场景，而不是对模型进行各种变形以让其去适合具体的应用。

本书在写作的过程中参考了很多国内外学者的论文和著作。如果没有他们的出色工作，没有他们极为宝贵的研究成果，本书是写不出来的。在本书出版之际，谨向他们表示衷心的感谢。

## VI 前 言

---

在本书写作过程中，笔者常为自己的学识不足而苦恼。自然语言处理作为一门交叉性边缘性学科，涉及语言学、计算机科学、数学等各个方面的知识，笔者学识浅陋，论述之中倘有不当，恳请读者批评指正。有任何意见和建议请发到 392071814@qq.com，不胜感激。

最后，谨向帮助、支持和鼓励我完成本书的我的家人、同事、领导、朋友以及出版社的领导、编辑致以深深的敬意和真挚的感谢！

作者

2017年9月于杭州

---

轻松注册成为博文视点社区用户（[www.broadview.com.cn](http://www.broadview.com.cn)），扫码直达本书页面。

- **提交勘误：**您对书中内容的修改意见可在 [提交勘误](#) 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动：**在页面下方 [读者评论](#) 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/32763>



# 目 录

## 第 1 篇 语义模型详解

<b>第 1 章 关键词抽取模型</b>	<b>3</b>
1.1 TF-IDF 算法实现关键词抽取	4
1.2 TextRank 算法实现关键词抽取	11
1.3 基于语义的统计语言模型实现关键词抽取	16
<b>第 2 章 短语抽取模型</b>	<b>22</b>
2.1 基于互信息和左右信息熵实现短语抽取	23
2.2 TextRank 算法实现短语抽取	28
2.3 LDA 算法实现短语抽取	31
<b>第 3 章 自动摘要抽取模型</b>	<b>38</b>
3.1 决策树算法实现自动摘要	39
3.2 基于逻辑回归算法实现自动摘要	44
3.3 贝叶斯算法实现自动摘要	50
<b>第 4 章 深度学习——计算任意词距离模型</b>	<b>55</b>
4.1 FP-Growth 算法实现词距离计算	56
4.2 N-Gram 算法实现词距离计算	61

---

4.3	BP 算法实现词距离计算	65
<b>第 5 章</b>	<b>拼音汉字混合识别模型</b>	<b>70</b>
5.1	贝叶斯模型实现拼音汉字混合识别	71
5.2	HMM 模型实现拼音汉字混合识别	75
5.3	RNN 神经网络模型实现拼音汉字混合识别	80
<b>第 6 章</b>	<b>文本自动生成模型</b>	<b>87</b>
6.1	基于关键词的文本自动生成模型	88
6.2	RNN 模型实现文本自动生成	93
<b>第 2 篇 自然语言处理系统基础算法</b>		
<b>第 7 章</b>	<b>Dijkstra 算法</b>	<b>101</b>
7.1	算法应用原理介绍	102
7.2	算法数学原理介绍	102
7.3	算法源码说明	106
7.4	算法应用扩展	107
<b>第 8 章</b>	<b>AC-DoubleArrayTrie 算法</b>	<b>108</b>
8.1	算法应用原理介绍	109
8.2	算法数学原理介绍	111
8.3	算法应用扩展	116
<b>第 9 章</b>	<b>最大熵算法</b>	<b>117</b>
9.1	算法应用原理介绍	118
9.2	算法数学原理介绍	119

9.3 算法源码说明	124
9.4 算法应用扩展	125

## 第 10 章 CRF 算法 126

10.1 算法应用原理介绍	127
10.2 算法数学原理介绍	130
10.3 算法源码说明	135
10.4 算法应用扩展	136

## 第 11 章 马尔可夫逻辑网算法 137

11.1 算法应用原理介绍	138
11.2 算法数学原理介绍	142
11.3 算法源码说明	144
11.4 算法应用扩展	145

## 第 12 章 DIPRE 算法 147

12.1 算法应用原理介绍	148
12.2 算法数学原理介绍	151
12.3 算法源码说明	152
12.4 算法应用扩展	153

## 第 13 章 LSTM 算法 155

13.1 算法应用原理介绍	156
13.2 算法数学原理介绍	158
13.3 算法源码说明	163
13.4 算法应用扩展	165

<b>第 14 章 TransE 算法</b>	<b>166</b>
14.1 算法应用原理介绍	167
14.2 算法数学原理介绍	170
14.3 算法源码说明	172
14.4 算法应用扩展	174

### 第 3 篇 系统案例实战

<b>第 15 章 搭建舆情分析与挖掘的系统</b>	<b>177</b>
15.1 系统功能设计简述	178
15.2 系统模块实现详解	181
15.3 系统实现源码说明	186

# 第 1 篇

## 语义模型详解

在本书的开始,我们想先从应用的角度来向读者介绍自然语言处理,同时对于这些应用的实现技术进行由浅入深的介绍。希望这样能让读者对于自然语言处理技术能解决什么问题,以及对于这些问题现在已有的解决方法有哪些,有一个初步的了解。这一篇将尽量少用公式,或者使用一些简单的公式,让大家在了解技术实现的同时,不至于被繁杂的公式所吓到。毕竟霍金大神曾说过,每一个公式都将会吓退一半的读者。



# 第 1 章

## 关键词抽取模型

首先我们向读者介绍的是关键词提取模型，关键词提取能让我们快速地了解一篇文章，或者从大量的语料中快速找到其想要说明的主题。特别是在信息爆炸的时代，能够有效提取文本的关键词，则对于快速、及时、高效地获取信息是非常有帮助的。

## 1.1 TF-IDF 算法实现关键词抽取

TF-IDF 算法是关键词提取算法中基础并且有效的一种算法，因为它的实现简单，并且效果显著，所以应用非常广泛。

### 1.1.1 场景

假设现在有一批短文本，比如很多条一句话新闻。现在需要提取这些一句话新闻的关键词。有哪些方法可以使你采用呢？这里介绍一种非常基础的，也非常好用的算法，叫做 TF-IDF 算法。

TF-IDF (term frequency - inverse document frequency) 是一种用于资讯检索与资讯探勘的常用加权技术。TF-IDF 是一种统计方法，用以评估一字词对于一个文件集或一个语料库中的其中一份文件的重要程度。字词的重要性随着它在文件中出现的次数呈正比地增加，但同时也会随着它在语料库中出现的频率呈反比地下降。

### 1.1.2 原理

**TF-IDF 的主要思想是：**如果某个词或短语在一篇文章中出现的频率 (Term Frequency, TF) 高，并且在其他文章中很少出现，即反文档频率 (Inverse Document Frequency, IDF) 低，则认为此词或者短语具有很好的类别区分能力，适合用来分类。那么对于这篇文章来说，这个词也就可以算作该文章的一个关键性的词语。基于上述思想，就提出了 TF-IDF 算法，具体计算公式如下：

$$tfidf_{i,j} = tf_{i,j} \times idf_i$$

其中，

$tfidf_{i,j}$ ：是指词  $i$  相对于文档  $j$  的重要性值。

$tf_{i,j}$ ：指的是某一个给定的词语在指定文档中出现的次数占比。即给定的词语在该文档中出现的频率。这个数字是对**词数** (term count) 的归一化，以防止它偏向长的文档。计算公式如下：

$$tf_{ij} = \frac{n_{i,j}}{\sum_k n_{k,j}}$$

其中

$n_{i,j}$  是该词在文件  $d_j$  中出现的次数。

$\sum_k n_{k,j}$  是在文件  $d_j$  中所有字词的出现次数之和。

$idf_i$ : 指的是词  $i$  的逆向文档频率, 是用总文档数目除以包含指定词语的文档的数目, 再将得到的商取对数得到。这是一种度量词语重要性的指标。计算公式如下:

$$idf_i = \log \frac{|D|}{|\{j: t_i \in d_j\}|}$$

其中

$|D|$  为语料库中的文档总数。

$|\{j: t_i \in d_j\}|$  为包含词语  $t_i$  的文档数目。

至此, 我们对 TF-IDF 算法有了一个初步的了解, 下面从一个具体的例子来看看这个算法在实际例子中的应用。

### 1.1.3 实例

在开篇的场景部分, 我们提了一个场景, 对短文本进行关键词提取。这里就以这样的例子进行算法说明。

先看看测试数据 (以下数据摘自百度百科)。

文档 1: 程序员 (Programmer) 是从事程序开发、维护的专业人员。一般将程序员分为程序设计人员和程序编码人员, 但两者的界限并不非常清楚, 特别是在中国。软件从业人员分为初级程序员、高级程序员、系统分析员和项目经理四大类。

文档 2: 现在网络流行上把男程序员称为“程序猿”, 女程序员称为“程序媛”。目前从事 IT 技术行业的大多数为男性, 女性多数从事其他 (如: 会计, 行政, 人力资源等) 种类的工作, 在 IT 技术里女程序员是很受欢迎的, 因此现在人们爱称女程序员为“程序媛”。

因为 TF-IDF 对词的顺序不关心, 所以分词部分就不作说明了。假设我们对上述两个文

档完成了分词，并且将每个文档中的词按照空格分隔存储在一起。并且对每一句话存储一行。

实现 TF 特征计算的代码如图 1-1 所示：

```
HashMap<String, Integer> wordMap=new HashMap<String, Integer>();
int totalCount=0;
for(int i=0; i<numLine; i++){
    String str=inputData.nextLine(); //提取每一行
    String[] strArray=str.split(" "); //按空格切割
    totalCount+=strArray.length; //统计词的总长度
    for(int j=0; j<strArray.length; j++){ //遍历每一个词并进行统计
        if(!wordMap.containsKey(strArray[j])){ //如果不存在，添加
            wordMap.put(strArray[j], 1);
        }
        else{ //如果已存在，count+1
            int v=wordMap.get(strArray[j]);
            wordMap.put(strArray[j], v+1);
        }
    }
}
HashMap<String, Double> tfMap=new HashMap<String, Double>();
Iterator it=wordMap.keySet().iterator();
while (it.hasNext()){
    String key=(String) it.next();
    int v=wordMap.get(key);
    tfMap.put(key, 1.0*v/totalCount); //TF Map
}
```

图 1-1

这里是以一个文档文本的处理为例，多个文档则在外面再加一层循环即可。首先对每一行按照空格进行切分，获得每一行的词组，然后对其中的每个词进行词频统计，计入 wordMap 中。其中代码部分为图 1-1 的第 1~17 行。

1. 首先声明了一个 map 结构体 wordMap，该结构体是一个<key,V>对。在 map 结构体中 key 是唯一的，所以同一个词在 map 中只有一个记录，通过 V 值的增加计算 key 出现的次数。
2. 然后对文档进行逐行遍历。并按照空格对每一行中的词进行分割。

3. 之后将每个词添加到 `wordMap` 中，实现对整个文档的词频统计。

在得到词频统计结果 `wordMap` 之后，就遍历整个 `map`，对每一个词计算其 TF 特征值。其中代码部分为图 1-1 的第 18~24 行。

1. 同样首先声明一个 `map` 结构体 `tfMap`，用于存放每个词的 TF 值。其中 TF 值是 `double` 类型的。
2. 如果对词频统计 `map`，对 `wordMap` 进行遍历，用每个词出现的次数，除以总的词频和，得到每个词的 TF 特征值。

实现 TDF 特征计算的代码如图 1-2 所示：

```
HashMap<String, Double> idfMap=new HashMap<String, Double>();
Iterator it1=wordMap.keySet().iterator();
while (it1.hasNext()){
    String key=(String) it1.next(); //取当前 word
    //在其余小类中查找是否出现过 key
    int otherClassNum=1; //记录每个 word 出现的总类别数
    for (int m=0; m<totalClassNum-1; m++){ //遍历所有的类
        int beginning=a[m]; //当前类的起始行
        int ending=a[m+1]; //终止行

        InputStreamReader I=new InputStreamReader(new
FileInputStream("originalData.txt"), "gbk");
        BufferedReader sr = new BufferedReader(I);
        String txt=sr.readLine(); //readLine 读取一行，字符串
        int next=1; //记录读取的行数
        while(txt!=null){
            if(next>=beginning-1 && next<ending){
String []str2Array=txt.split(" ");
int l;
for (l=0; l<str2Array.length; l++){
    if(str2Array[l].equals(key)){
        otherClassNum++;
        break; //找到了就跳出循环
    }
}
```

图 1-2

```
}
if(l<str2Array.length) //说明已经找到
    break;
    }
    txt=sr.readLine(); //继续读取下一行
    next++;
    }
}
idfMap.put(key, Math.log10(1.0*(totalClassNum-1)/otherClassNum));
}
```

图 1-2 (续)

我们根据 TF 特征计算过程中统计的词频统计结果作为基础,来判断每个词在其他类里面出现的次数。具体代码为图 1-2 的第 1~34 行。

1. 首先声明一个 map 结构体,用于存储每个词对应的 IDF 特征值。
2. 然后遍历词频统计结果 wordMap,对每个词进行遍历查找,查找每个词在多少个文档中出现过。
3. 最后计算词的 IDF 特征值,并且存入结果 idfMap 中。

实现 TF\*TDF 特征计算并选出前五名的代码如图 1-3 所示:

```
HashMap<String, Double> tfIdfMap=new HashMap<String, Double>();
Iterator it2=tfMap.keySet().iterator();
Iterator it3=idfMap.keySet().iterator();
while(it2.hasNext() && it3.hasNext()){
    String key1=(String) it2.next();
    String key2=(String) it3.next();
    Double v=tfMap.get(key1)*idfMap.get(key2); //tf*idf
    tfIdfMap.put(key1, v); //idfTfMap
}
//排序
List<Entry<String, Double>> infoIds =
    new ArrayList<Entry<String, Double>>(tfIdfMap.entrySet());
Collections.sort(infoIds, new Comparator<Entry<String, Double>>() {
    public int compare(Entry<String, Double> o1, Entry<String, Double> o2)
    {
        if(o2.getValue() - o1.getValue())>0){
            return 1;
        }
        return -1;
    }
});
```

图 1-3

```
    }  
  });  
  
  int selectNum;  
  if(infoIds.size()<5){ //防止有些小类问题数量少  
    selectNum=infoIds.size();  
  }  
  else  
    selectNum=5;  
  
  for (int q=0; q<selectNum; q++){ //只取前五名  
    String id = infoIds.get(q).toString();  
    int equalIndex=id.indexOf('=');  
    char []strChar=id.toCharArray();  
    String fea=new String();  
    for (int d=0; d<equalIndex; d++){  
      fea=fea+strChar[d];  
    }  
    System.out.println(fea);  
    output.print(fea);  
    output.print(' ');  
  }  
}
```

图 1-3 (续)

首先计算  $TF*IDF$  特征值，经过上面两个步骤，我们已经知道每个文档中每个词的  $TF$  特征值，并且存储在 `tfMap` 中，同时对该结果中的每一个词遍历所有文档、计算其出现的次数，并计算其  $IDF$  特征值，并且存储在 `idfMap` 中。下面我们就开始计算两个特征的乘积值，具体代码为图 1-3 中的第 1~9 行：

1. 遍历其中任何一个 `map` 即可，这里以 `tfMap` 为遍历对象，遍历 `tfmap`，取出每个词对应的  $TF$  特征值。
2. 然后在 `idfMap` 中取出对应的  $IDF$  特征值。
3. 将两个值相乘之后，作为出现词语的  $TF-IDF$  特征值存入到 `tfIdfMap` 中。

然后是对  $TF-IDF$  特征 `map` 进行排序。因为这里使用的 `hashMap` 是无序的结构体，所以需要先将数据结构转换为 `List` 的结构体，再进行排序操作。具体代码为图 1-3 中的第 11~20 行。

1. 首先将  $TF-IDF$  特征值存入到一个 `List` 结构体 `infoIds` 中。
2. 然后重写 `List` 结构体对应的 `compare` 算法，实行根据每个词对应的  $TF-IDF$  特征值由大到小地进行排列。

最后选出 TF-IDF 特征值前五名的词，作为文档集合的关键词。具体代码为图 1-3 中的第 22~37 行。

1. 如果提取到的词数量小于 5，则全部取出。
2. 如果提取到的词数量大于等于 5，则取排序后列表的前五名作为结果输出。

### 1.1.4 拓展

从上例可以知道，TF-IDF 算法会倾向于选出某一特定文档内的高频率词语，同时该词语在整个文档集合中分布是比较集中的。因此，TF-IDF 倾向于过滤掉常见的词语，保留“独有”的词语。

但是这也造成了 TF-IDF 自身的一些缺陷。因为 IDF 的主要思想是：如果包含词条  $t$  的文档越少，也就是  $n$  越小，IDF 越大，则说明词条  $t$  具有很好的类别区分能力。如果某一类文档  $C$  中包含词条  $t$  的文档数为  $m$ ，而其他类包含  $t$  的文档总数为  $k$ ，显然所有包含  $t$  的文档数  $n=m+k$ ，当  $m$  大的时候， $n$  也大，按照 IDF 公式得到的 IDF 的值会小，就说明该词条  $t$  类别区分能力不强。但是实际上，如果一个词条在一个类的文档中频繁出现，则说明该词条能够很好地代表这个类的文本特征，这样的词条应该给它们赋予较高的权重，并选来作为该类文本的特征词，以区别于其他类的文档。比如对于如下几个短文本：

1. 鲜花多少钱？
2. 白百合多少钱？
3. 水仙花多少钱？
4. 苹果多少钱？

如果按照 TF-IDF 算法，鲜花、苹果这些主体词会成为关键词，但是从这些语句的总体来看，它们又都属于询问价格的类型，所以“多少钱”应该成为关键词。这就是 IDF 的不足之处。

改进的方法可以通过改变文档结构，比如将上述短文本归并为一个文档，这样就可以在增加 TF 值的同时，也增加 IDF 值。但是这样就会增加模型计算的成本，需要大量的人为经验加入其中。或者可以改进计算公式，将 IDF 的计算由文档的逆向频率变成词的逆向频率（即 IWF），这样可以适当地降低逆向频率的权重，对上述情况进行一部分改进。当然这两种改进思路都是在基于 TF-IDF 计算框架的前提下进行的，如果跳出这个框架，又会有

更多的方法来进行克服和改进。

## 1.2 TextRank 算法实现关键词抽取

1.1 节我们介绍了 TF-IDF 算法，其对有多段文本的关键词提取非常有效，但是对于单篇或者文档分割较少的文本则表现得不是特别好，下面就介绍一种适合处理该场景的算法。

### 1.2.1 场景

如果需要提取关键词的语句只有一句话。那么基于 TF-IDF 可以知道，所有关键词的重要度都为 0，因为 IDF 值为 0。针对这种情况，我们介绍另外一种算法：TextRank 算法，来实现关键词提取。当然这两种算法的区别绝不止是上述场景所说的，这里只是提出一个引子。

TextRank 是一种基于图排序的算法。其基本思想来源于谷歌的 PageRank 算法，通过把文本分割成若干组成单元（单词、句子）并建立图模型，利用投票机制对文本中的重要成分进行排序，仅利用单篇文档本身的信息即可实现关键词提取、做文摘。

### 1.2.2 原理

TextRank 利用投票的原理，让每一个单词给它的邻居（术语称窗口）投赞成票，票的权重取决于自己的票数。所以如上所述，它是一个图排序模型，我们假设每一个词是一个顶点（Vertex），那么所有的词就构成了一个网络，在这个网络里面每一个顶点会有指向其他顶点的边，也会有由其他顶点指向自己的边。通过计算每个顶点所连接的指向自己的顶点的权重和，最终得到该顶点的权重值。

但是这里有一个问题，就是初始值的确定。是赋值为 0，还是给一个非 0 的初始值。因为这个思想是源自谷歌的 PageRank 算法。既然是源自 PageRank 算法，那么 PageRank 所固有的问题，TextRank 也不会例外。比如因为目标的权重取决于自身的权重，所以就会存在一个“先有鸡还是先有蛋”的悖论，对于这个问题 PageRank 采用矩阵迭代收敛的方式解决了这个悖论。TextRank 也不例外。那么在矩阵迭代的过程中，如果初始矩阵为 0，那么后续计算就比较麻烦，所以这里要给初始值一个非 0 的值。这里就引入了一个阻尼系数的

概念。在图模型中，该参数表示从某一个指定的顶点，到任意一个其他顶点的概率。所以 TextRank 具体公式如下：

$$WS(V_i) = (1-d) + d \cdot \sum_{V_j \in In(V_i)} \frac{w_{ij}}{\sum_{V_k \in Out(V_j)} w_{j,k}} WS(V_j)$$

其中，

$d$ : 表示阻尼系数，一般设置为 0.85（这个完全是经验值）。

$V_i$ : 表示图中的任一节点。

$In(V_i)$ : 表示指向顶点  $V_i$  的所有顶点集合。

$Out(V_j)$ : 表示由顶点  $V_j$  连接出去的所有顶点集合。

$w_{ij}$ : 表示顶点  $V_i$  和  $V_j$  的连接权重。

$WS(V_i)$ : 表示顶点  $V_i$  的最终排序权重。

因为我们这里计算的是单个词的重要性，如果没有特殊的语义要求，每个单词之间的连接权重是一样的，假设都为 1，那么上面的公式又可以写成如下的形式：

$$WS(V_i) = (1-d) + d \cdot \sum_{V_j \in In(V_i)} \frac{1}{|Out(V_j)|} WS(V_j)$$

其中，

$d$ : 表示阻尼系数，一般设置为 0.85（这个完全是经验值）。

$V_i$ : 表示图中的任一节点。

$In(V_i)$ : 表示指向顶点  $V_i$  的所有顶点集合。

$|Out(V_j)|$ : 表示由顶点  $V_j$  连接出去的所有顶点集合个数。

$WS(V_i)$ : 表示顶点  $V_i$  的最终排序权重。

上述形式也即 PageRank 的计算公式。所以这两者之间其实是一致的，PageRank 是 TextRank 的一种特殊情况。

### 1.2.3 实例

在开篇的场景部分，我们提了一个场景，对一句话文本进行关键词提取。我们这里就以这样的一个例子进行算法说明。

先看看测试数据（以下数据摘自百度百科）：

一句话：程序员（Programmer）是从事程序开发、维护的专业人员。一般将程序员分为程序设计人员和程序编码人员，但两者的界限并不非常清楚，特别是在中国。软件从业人员分为初级程序员、高级程序员、系统分析员和项目经理四大类。

因为分词在这里不是重点，所以分词部分就不做特别说明了。假设我们对上述一句话完成了分词，并且将各个词按照空格分隔存储在一起。

因为 TextRank 算法的一个点是给“邻居”，所以对邻居的获得就是整个计算的关键。这里选择用滑动窗口的方式对每个单词取邻居。假设，我们取一个长度为  $k$  的滑动窗口，则  $w_1, w_2, \dots, w_k$ 、 $w_2, w_3, \dots, w_{k+1}$ 、 $w_3, w_4, \dots, w_{k+2}$  等都是窗口。在一个窗口中的任意两个单词对应的节点之间存在一个无向无权的边。在这个邻居上面构成图，可以计算出每个单词节点的重要性。最重要的若干单词可以作为关键词。

实现连接矩阵计算的代码如图 1-4 所示：

```
String[] strArray = str.split(" "); // 按空格切割
for (int j = 0; j < strArray.length; j++) {
    initialMap.put(strArray[j], 1 - d);
    int lower = Math.max(0, j - windowsLength); // 计算窗口的下限
    int upper = Math.min(strArray.length, j + windowsLength); // 计算窗口的
    上限
    for (int l = lower; l < j; l++) {
        putMap(wordMap, strArray, j, l); // 将窗口内的单词加入到单词 j 的连出集合
    当中
    }
    if (j < strArray.length - 1) {
        for (int l = j + 1; l < upper; l++) {
            putMap(wordMap, strArray, j, l);
        }
    }
}
```

图 1-4

这里是以一个文档文本的处理为例，多个文档则在外面再多加一层循环即可。首先对每一行按照空格进行切分，获得每一行的词组，然后统计每个词前后窗口内的单词，计入 wordMap 中。其中代码部分为图 1-4 的第 1~14 行：

1. 首先声明了一个 map 结构体 wordMap，该结构体是一个<key,V>对。这里的 V 是一个集合。因为我们在开始的假设词与词之间的连接权重都是 1，所以这里选择了去重的集合作为存储结构，这样可以对重复出现的词进行去重，只保留一个值。
2. 然后对文档进行逐行遍历。并按照空格对每一行中的词进行分割。
3. 之后对每一个，取出其前后窗口的词，并且将其放入到 wordMap 中，进行记录，以便进行后续计算时候的查询。
4. 同时在这一个过程中，还需要对每一个词赋值，一个初始权重，用于后续的迭代。

实现迭代计算的代码如图 1-5 所示：

```
for (int i = 0; i < max_iter; ++i) // 设定迭代次数
{
    double max_diff = 0;
    for (Map.Entry<String, Set<String>> entry : wordMap.entrySet())
    {
        String key = entry.getKey();
        Set<String> value = entry.getValue();
        for (String out : value) {
            int size = wordMap.get(out).size(); // 计算节点 key 连出节
点的个数

            if (key.equals(out) || size == 0)
                continue; // 如果是 key 节点本身或者没有连出节点，则权重不更新
            initialMap.put(
                key,
                initialMap.get(key)
                    + d
                    / size
                    * (score.get(out) == null ? 0 : score
                        .get(out))); // TextRank 计算公式
        }
        // 计算迭代前后两次变化
        max_diff = Math.max(max_diff, Math.abs(initialMap.get(key)
            - (score.get(key) == null ? 0 : score.get(key))));
    }
}
```

图 1-5

```
    }  
    score = initialMap;  
    if (max_diff <= min_diff)  
        break;  
}
```

图 1-5 (续)

1. 首先设定最大迭代次数，并依次进行逐步迭代。
2. 按照连出矩阵，对每一个单词节点更新其排序权重。
3. 对于连出到自身或者连出为空的单词节点不进行计算，因为这部分节点在图中属于孤立节点，所以只要保持其初始值不变即可，既不影响别人，也不受别人的影响。
4. 对于有连出到其他词的单词节点，则按照 TextRank 公式，逐步更新其排序权重，更新的方式按照简化的情况进行：对单词与单词之间的联系权重为 1 的情况进行计算更新。
5. 同时根据前后两次迭代之间单词的权重变化值，来判断是否提前结束循环过程。也即如果迭代步数还没有达到最大的迭代次数，但是各个单词节点的权重均已不再变化，也即整个迭代收敛到一个稳定值，则结束整个迭代过程，不再进行迭代。

## 1.2.4 拓展

从上面可以看出，TextRank 算法对于一段文本中多次出现的词，会赋予更大的权重，因为它连出的节点会更多，所以当各个节点初始权重一致的时候，则最终出现次数多的词权重会更大。

这也造成该算法的一些弊端，比如对于类似于“的”、“你、我、他”等常用词，就会出现比较大的误差，因为这些词一般没有什么特别的含义，仅仅只是一个连接词或者指代词，而它们的权重一般会比较大。对于这种情况，可以在最开始构建连接矩阵的时候进行处理。比如去掉里面的停用词或者其他符合一定规则的词。也可以通过正向过滤，选择自己需要的词性的词，比如只选“名词、动词、形容词、副词”这些类型的词，对于其他类型的词均过滤掉。得出实际有用的词语。

对于单词之间相似度的计算也是决定最终效果好坏的一个关键因素。对于单词的相似度计算可以采用基于编辑距离、语义词典、余弦相似度等传统方法，也可以采用基于 Embedding 的方法。特别是目前基于深度学习的 Word2vec、skip-gram 等算法的兴起，此类方法有了更好的效果和更加实用的工具。这里就不深入展开了。

## 1.3 基于语义的统计语言模型实现关键词抽取

上面介绍的两种算法更多反映的是文本的统计信息，对于文本之间的语义关系考虑得比较少，所以这里就介绍一种能够体现文本语义关系的关键词提取算法。

### 1.3.1 场景

对于如下的文本，如何提取出更加符合其主题分布的关键词。

1. 鲜花多少钱？
2. 白百合多少钱？
3. 水仙花多少钱？

上面这三个语句，描述的都是鲜花这个主题下面的问题。所以如果希望提取的关键词更加符合其主题分布，那么应该是“鲜花”的权重最高。如果按照 TF-IDF、TextRank 的算法，“鲜花”、“白百合”、“水仙花”这三个词的权重是一样的（当然如果你认为“多少钱”应该是这三个语句的主题，那么这三个词的权重一致也就是理所当然的了）。针对这种情况，我们介绍一种基于 LDA（Latent Dirichlet Allocation）的关键词提取算法。

LDA 模型包含词、主题和文档三层结构，如图 1-6 所示。LDA 最早是由 Blei 等，以 pLSI 为基础，提出的服从 Dirichlet 分布的  $K$  维隐含随机变量表示文档主题概率分布、模拟文档的一个产生过程。后来 Griffiths 等对参数  $\beta$  施加了 Dirichlet 先验分布，使得 LDA 模型成为一个完整的生成模型。

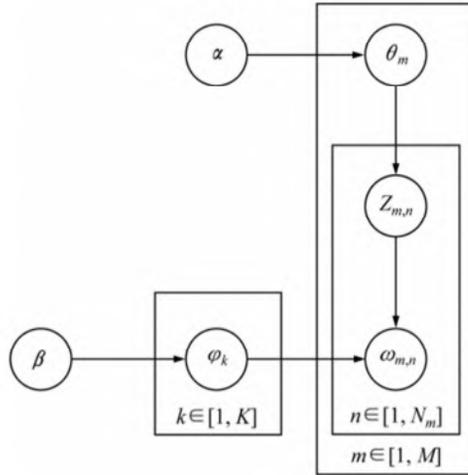


图 1-6 LDA 的图模型

其中,

1.  $\phi_k$  为主题  $k$  中的词汇概率分布,  $\theta_m$  为第  $m$  篇文档的主题概率分布,  $\phi_k$  和  $\theta_m$  服从 *Dirichlet* 分布,  $\phi_k$  和  $\theta_m$  作为多项式分布的参数分别用于生成主题和单词。
2.  $\alpha$  和  $\beta$  分别为  $\phi_k$  和  $\theta_m$  的分布参数,  $\alpha$  反映了文档集中隐含主题之间的相对强弱,  $\beta$  为所有隐含主题自身的概率分布。
3.  $K$  为主题数目。
4.  $M$  为文档集中文档的数目。
5.  $N_m$  为第  $m$  篇文档的词의总数。
6.  $\omega_{m,n}$  和  $Z_{m,n}$  分别为第  $m$  篇文档中第  $n$  个单词和其隐含主题。

### 1.3.2 原理

如上所述,在 LDA 模型中,包含词、主题、文档三层结构。该模型认为一篇文档的生成过程是:先挑选若干主题,再为每个主题挑选若干词语。最终,这些词语就组成了一篇文章。所以主题对于文章是服从多项分布的,同时单词对于主题也是服从多项分布的。基于这样的理论,我们可以知道,如果一个单词  $w$  对于主题  $t$  非常重要,而主题  $t$  对于文章  $d$  又非常重要,那么单词  $w$  对于文章  $d$  就很重要,并且在同主题的词  $w_i (i = 1, 2, 3, \dots)$  里面,

单词  $w$  的权重也会比较大。而这正好可以解决我们上面所描述的问题。

所以下面就要计算两个概率：单词对于主题的概率和主题对于文档的概率。这里我们选择 Gibbs 采样法来进行概率的计算。具体公式如下：

主题  $T_k$  下各个词  $w_i$  的权重计算公式：

$$P(w_i|T_k) = \frac{C_{ik} + \beta}{\sum_{i=1}^N C_{ik} + N \cdot \beta} = \varphi_i^{t=k}$$

其中

$w_i$ ：表示单词集中任一单词。

$T_k$ ：表示主题集中任一主题。

$P(w_i|T_k)$ ：表示在主题为  $k$  时，单词  $i$  出现的概率，其简记的形式为  $\varphi_i^{t=k}$ 。

$C_{ik}$ ：表示语料库中单词  $i$  被赋予主题  $k$  的次数。

$N$ ：表示词汇表的大小。

$\beta$ ：表示超参数。

文档  $D_m$  下各个词  $T_k$  的权重计算公式：

$$P(T_k|D_m) = \frac{C_{km} + \alpha}{\sum_{k=1}^K C_{km} + K \cdot \alpha} = \theta_{t=k}^m$$

其中

$D_m$ ：表示文档集中任一文档。

$T_k$ ：表示主题集中任一主题。

$P(T_k|D_m)$ ：表示在文档为  $m$  时，主题  $k$  出现的概率，其简记的形式为  $\theta_{t=k}^m$ 。

$C_{km}$ ：表示语料库中文档  $m$  中单词被赋予主题  $k$  的次数。

$K$ ：表示主题的数量。

$\alpha$ ：表示超参数。

在上述两个公式中，为了平滑非包含的单词和主题，所以分子中分别添加了 LDA 模型

中的超参数 $\alpha$ 和 $\beta$ 。如果觉得所计算的场景不需要，也可以不加这两个参数。

现在得到了指定文档下某主题出现的概率，以及指定主题下、某单词出现的概率。那么由联合概率分布可以知道，对于指定文档某单词出现的概率可以由如下公式计算得到：

$$P(w_i|D_m) = \sum_{k=1}^K \varphi_i^{t=k} \cdot \theta_{t=k}^m$$

基于上述公式，我们就可以算出单词  $i$  对于文档  $m$  的主题重要性。但是由于在 LDA 主题概率模型中，所有的词汇都会以一定的概率出现在每个主题，所以这样会导致最终计算的单词对于文档的主题重要性值区分度受影响。为了避免这种情况，一般会将单词相对于主题概率小于一定阈值的概率置为 0。至于这个阈值设置为多少，则可以根据自己的实际情况自由选择。

### 1.3.3 实例

基于本节开头提出的场景，我们来完成基于文章主题权重的关键词提取实例。同上面所述，分词在这里不是重点，所以分词部分就不做特别说明了。假设我们对上述一句话完成了分词，并且将各个词按照空格分隔存储在了一起。

这里借鉴于 1.2 节的处理，首先处理掉非重要词，采用正向过滤的方法，即选择特定词性的词，在这里我们选择词性为名词、形容词等词性的词。

在得到候选词表后，对语料库进行 Gibbs 采样，得到单词-主题，文档-主题的概率分布统计矩阵。

计算单词对于文档的主题概率权重的实现代码如图 1-7 所示：

```
for (int j = 0; j < strArray.length; j++) {
    String word = strArray[j]; // 获得当前要计算的单词
    int i = wordIndexMap.get(word); // 获得给定单词在单词-主题分布矩阵中的行号
    // 计算单词在指定文档 m 中的主题概率权重
    double word2TopicWeightSum = 0.0;
    // 遍历所有主题, 计算其对于单词 i 的频次总和
```

图 1-7

```
for (int k = 0; k < topicSize; k++) {
    word2TopicWeightSum += word2TopicCount[i][k];
}
double topic2DocWeightSum = 0.0;
//遍历所有主题,计算其对于文档 m 的频次总和
for (int k = 0; k < topicSize; k++) {
    topic2DocWeightSum += topic2DocCount[m][k];
}
double weightSum = 0.0;
//遍历所有主题,计算其单词 i 对于文档 m 的主题概率权重
for (int k = 0; k < topicSize; k++) {
    double word2TopicWeight = (word2TopicCount[i][k] +
beta)/(word2TopicWeightSum + wordSize*beta);
    double topic2DocWeight = (topic2DocCount[m][k] +
alpha)/(topic2DocWeightSum + topicSize*alpha);
    weightSum += word2TopicWeight*topic2DocWeight;
}
resultMap.put(word, weightSum);
}
```

图 1-7 (续)

这里以一个文档文本的处理为例,多个文档则在外面再多加一层循环即可。首先对输入文本按照空格进行切分,获得每一个单词,然后统计每个词的主题概率权重。其中代码部分为图 1-7 的第 1~20 行。

1. 因为对于每一个单词,在计算其相对于文档  $m$  的主题概率权重的时候,文档  $m$  都是确定的,所以在遍历每个单词之前先要对主题-文档的分布概率求和,计算其总的频次数,以备后续计算使用。如代码第 1~4 行所示。
2. 对于要计算的单词,首先要获得它在单词-文档的分布矩阵中的位置,也即双数组存储的行号。这里需要说明一点,因为我们待计算的文档,也会放到语料库中进行学习,所以不会存在待计算的单词不在已知单词库中的情况。对于因为主题概率分布太小而被过滤掉的单词,它的计数会被置为 0,而这一单元格的记录还是被保留的,所以这里不会出现空指针的问题。
3. 因为主题数量一般不会太大,所以就先进行了主题的遍历,求出指定单词相对于各个主题分布词频数的总和,以备后续计算使用。如代码第 11~13 行所示。

4. 最终我们将计算结果存储在一个 `map` 中，以备后续计算排序筛选，具体计算方式在之前的小节已经描述，这里不再赘述。

### 1.3.4 拓展

从上可以看出，基于 LDA 主题概率模型的关键词提取方法的准确度，会严重依赖于基础语料库，而这个语料库还需要有一定的丰富性，这样才可以使得计算的概率值具有一定的鲁棒性。这就会造成比较大的人力投入，对于模型的灵活扩展就会受到限制。

基于本框架本身的改进并不是很多，而借助于其他方法，对于上述缺陷的改进的方法还是比较多的，比如通过借助于知网，或者同义词林的方法获得单词的准确语义关系，这样就可以获得更好的语义关系。同时基于这些准确的语义关系，可以建立词语结构模板；然后基于这些模板运用频繁模式挖掘，发掘更多的符合这样词语结构模板的词语关系；并且根据预先设置的规则条件，给这些词语对添加对应的语义关系；这样就可以实现语义的批量扩展，增大基础的语料数据。

## 第 2 章

# 短语抽取模型

在完成文本的关键词提取之后，我们就会希望能将提取内容更加便于理解，那从我们的常识可以知道，短语会比单词更有利于对文章的理解，因为它会包含更多的信息。所以本章就着重介绍短语提取相关的算法。

## 2.1 基于互信息和左右信息熵实现短语抽取

首先介绍一种基于信息熵的算法，信息熵是对于分布纯净度的一个度量。这个值随着分布的纯净度增加而降低。所以当分布中只有一种情况时，那么信息熵就为最小，假设为0。

既然信息熵有这样的特性，那就可以用来衡量两个词是不是经常组合在一起的情况，因为如果这两个词是固定搭配的时候，那么它们互为对方的唯一连接，也就是从任何一个词的角度来说其连接词的分布都是唯一的。下面就介绍用这种方法来实现短语的提取。

### 2.1.1 场景

在第1章，我们介绍了对于关键词提取的一些方法。下面考虑另外一种场景，我们现在想提取一篇文章里面的关键短语，比如对于“鲜花多少钱？”，希望最终提取出来的结果是“鲜花多少钱”，而不是“鲜花”，或者“多少钱”。因为对于文章的理解，短语能体现更多的信息，对于理解文章的大意更有帮助。如果我们把短语看作多个关键词的拼接，那么上面的这个问题就可以转化为，寻找经常出现在一起的词组。对于这个问题，这里介绍一种比较基础的算法：基于互信息和左右信息熵实现关键短语抽取。

### 2.1.2 原理

互信息是体现两个或者多个元素之间的依赖关系。以两个元素为例（以下没有特殊说明，均以两个元素之间的关系为例进行说明），互信息计算公式：

$$MI = P(X, Y) \log \frac{P(X, Y)}{P(X) \cdot P(Y)}$$

其中

$P(X, Y)$ ：表示元素  $X$  和元素  $Y$  共同出现的概率。

$P(X)$ ：表示元素  $X$  单独出现的概率。

$P(Y)$ ：表示元素  $Y$  单独出现的概率。

从互信息的定义我们可以知道，其只是考虑两个单词共现的概率。现在我们考虑这样一种场景，比如在一个文档中，“鲜花-多少钱”这个单词对一共出现了3次，而且每次都是成对出现的，即这两个单词在整个文档中单独出现的次数也是3次。另外一个单词对“百合-好看”也一共出现3次，但是这两个单词在整个文档中分别出现了5次，即还有2次分别是和其他词结合在一起的。如果仅仅计算互信息，在整个文档单词数量很大的情况下，这两个单词对的互信息值会比较接近，都很大。但是实际上这两个单词对的差别应该很大。

由上面的问题描述，我们联想到一个概念：信息熵。信息熵是用来描述一个变量的确定性的，如果一变量的确定性越高，那么它的信息熵就越小，相反则越大。对应到我们的场景来说，如果单词  $X$  和单词  $Y$  组合的确定性越大（即这两次固定搭配的概率越大），那么它们的信息熵就越小，反之则越大。所以针对这个问题，我们分别对单词对中的单词加上信息熵，这样就可以对上述两种情况进一步区分。但是如上所述，固定搭配概率越大，其信息熵越小；而我们希望其作为关键短语的权值越大，但信息熵是一个  $0\sim 1$  之间的数，所以我们最终使用：1-信息熵，作为计算的权值。其计算公式如下：

左单词的信息熵：

$$E_{L-X} = 1 - (-P(X)_Y \log P(X)_Y)$$

其中

$P(X)_Y$ ：表示在单词  $Y$  左边的单词集合中， $X$  的概率。

右单词的信息熵：

$$E_{R-Y} = 1 - (-P(Y)_X \log P(Y)_X)$$

其中

$P(Y)_X$ ：表示在单词  $X$  左边的单词集合中， $Y$  的概率。

上述两个概率，相对于单词对  $X$ - $Y$ ，正好又可以描述为左右信息熵。所以结合第一步的互信息，这里就组成了基于互信息和左右信息熵的计算方法。但是完整的权重计算还需要再补充一些东西。上面的左右信息熵只是计算了一个词的，而一般一个词组中每个单词的左右单词都不止一个，所以一个词组的左右信息熵是上面单个词左右信息熵的和，具体的计算如下。

左信息熵计算公式:

$$E_L = \sum_{X \in \text{Left}(Y)} E_{L-X} = \sum_{X \in \text{Left}(Y)} 1 + P(X)_Y \log P(X)_Y$$

右信息熵计算公式:

$$E_R = \sum_{Y \in \text{Right}(X)} E_{R-Y} = \sum_{Y \in \text{Right}(X)} 1 + P(Y)_X \log P(Y)_X$$

如上所述,我们在互信息的基础上加上信息熵,是为了提高“纯净”词组的概率。所以对于这三个概率,我们最终采用相加的策略进行。因为我们希望最终的权重值和这三个值中任何一个值都是单调增的关系,那么在基础的实现方式里面,有加法和乘法可选,但是如上对于信息熵的描述,我们知道当词组非常“纯净”的时候,即左右词是唯一互连时,其信息熵值为 1,如果采用乘法,则其最终的权重值就等于互信息,这样就没有提高“纯净”词组的概率,所以这里我们采用加法策略进行,这样就可以实现,三个权重值任意一个值高的时候,整体权重是高的,同时对于“纯净”词组的概率也有提升,所以最终的权重计算公式如下:

$$w_{X-Y} = E_L + E_R + MI$$

其中

$w_{X-Y}$ : 表示由词  $X$  和词  $Y$  组成的词组的关键短语权重。

下面给出一个算法实现的 demo。

### 2.1.3 实现

按照惯例我们首先对文本进行分词,然后去除停用词,但保留文本的断句关系。即在最终存储的文本中,每一个断句存储一行。这样做是为了保留单词与单词之间的语义关系,避免把前后两句话中的单词合并成一个短语。假设我们对样本数据完成了分词,并按照如上规则将各个词按照空格分隔存储在一起。

实现互信息 MI 特征计算的代码如图 2-1 所示:

```
for (Entry<String, MIAndEntropyBean> entry : wordsMap.entrySet()) {
    //计算互信息
    String wordArrayString = entry.getKey();
    //获得总的词语计数
    int wordArrayCount = entry.getValue().getWordCount();
    String rightWordString = entry.getValue().getRightWord();
    String leftWordString = entry.getValue().getLeftWord();
    double leftWordCount = wordCountMap.get(leftWordString);
    double rightWordCount = wordCountMap.get(rightWordString);
    double MI =
        (wordArrayCount/totalWordArrays)/((leftWordCount/totalWords)*(rightWordCo
        unt/totalWords));
    //计算左信息熵
    double leftSum = getEntropy(leftEntropyMap, rightWordString);
    //计算右信息熵
    double rightSum = getEntropy(rightEntropyMap, leftWordString);
    //最终的权值
    double weightSum = MI + leftSum + rightSum;
    resultMap.put(wordArrayString, weightSum);
}
```

图 2-1

1. 如图 2-1 所示，我们将词组的基础信息存在一个 `map` 结构体中。同时声明了一个自己定义的结构体 `MIAndEntropyBean`，该结构体包含 3 个元素。
  - a) `leftWord`: 词组的左词。
  - b) `rightWord`: 词组的右词。
  - c) `wordCount`: 词组的计数。

对于每一个词组，我们都先计算其概率，也即左词和右词共同出现的概率，然后将其除以左词和右词单独出现概率的乘积，得到该词组的互信息 `MI`，如图 2-1 中第 9 行代码所示。

2. 在这里需要注意的是，在计算词组的概率和组成词组的各个单词的概率时，使用的分母是不一样的，因为它们取自不同的集合。计算词组的概率时，分母为总词组的个数；而计算各个词的概率时，分母为单个词的总数。

实现左、右信息熵特征计算的代码如图 2-2 所示：

```

for (Entry<String, MIAndEntropyBean> entry : wordsMap.entrySet()) {
    //计算互信息
    String wordArrayString = entry.getKey();
    //获得总的词语计数
    int wordArrayCount = entry.getValue().getWordCount();
    String rightWordString = entry.getValue().getRightWord();
    String leftWordString = entry.getValue().getLeftWord();
    double leftWordCount = wordCountMap.get(leftWordString);
    double rightWordCount = wordCountMap.get(rightWordString);
    double MI =
(wordArrayCount/totalWordArrays)/
((leftWordCount/totalWords)*
(rightWordCount/totalWords));
    //计算左信息熵
    double leftSum = getEntropy(leftEntropyMap, rightWordString);
    //计算右信息熵
    double rightSum = getEntropy(rightEntropyMap, leftWordString);
    //最终的权值
    double weightSum = MI + leftSum + rightSum;
    resultMap.put(wordArrayString, weightSum);
}
private static double getEntropy(
    Map<String, Map<String, Double>> leftEntropyMap,
    String rightWordString) {
    double leftSum = 0.0;
    Map<String, Double> leftMap = leftEntropyMap.get(rightWordString);
    double total = leftMap.get(WORD_COUNT);
    //计算信息熵
    for (Entry<String, Double> entry3 : leftMap.entrySet()) {
        leftSum += 1.0 +
entry3.getValue()/total*Math.log(entry3.getValue()/total);
    }
    return leftSum;
}

```

图 2-2

1. 如图 2-2 所示，我们在第 11 行实现了左信息熵 $E_L$ 的计算。因为左信息熵和右信息熵的计算逻辑完全一样，所以我们声明了一个公用的方法，供两个计算使用。
2. 在计算左信息熵时，我们首先得到词组中右词对应的左连接词表，这个词表由词和出现的频数构成。然后我们遍历词表，并计算每一个词的信息熵。

3. 最后加上和词组中右词对应的左连接词表的所有词的信息熵，得到最终的左信息熵。
4. 对于右信息熵的 $E_R$ 的计算逻辑完全一致，这里就不再赘述了。

## 2.1.4 拓展

由上述计算过程我们知道，该方法对于长尾词、异常组合的词会给予比较大的权重，但是这在大部分情况下是有问题的。一方面是因为我们对于词组的选择只是基于滑动的窗口，所以会有：“造成-比如”这样的词出现，而这样的词最终计算的结果会比较大，特别是当整个单词集合非常大的时候。

一般会根据词出现的频率过滤掉低频的词。当然对于词的过滤也可以做得更加智能一些。比如运用 1.3 节介绍的方法，先求出词组中各个词对应的主题概率，然后整个词组的主题概率就等于各个词的主题概率之和。然后再利用贝叶斯公式，求出给定词组下每个主题的概率分布，然后对于各个词组计算其余弦相似度，将相似度高的词聚为一组，每一组下面都只选一个主题概率最大的词，这样可以实现词的更精细的处理。

同时对于互信息的计算公式也可以进行调整，在我们的例子里面，对于三个权重值分别给予了相同的权重，并且对于最终计算的值也都没有做处理。也可以先对三个值进行归一化处理，然后再对其分别赋予权重，使得最终得到的值在 0~1 之间，也可以避免因为某个值过大，而对最终结果造成影响。

## 2.2 TextRank 算法实现短语抽取

这个算法已经算是老朋友了，在关键词提取中，我们已经介绍过它，在这里就看看从短语的角度，我们怎么用这个算法实现关键信息的提取。

### 2.2.1 场景

上面我们介绍了基于互信息和左右信息熵的关键短语提取算法。但是其中有一个问题，就是要人为地去筛选候选单词，以便提升所提取的关键短语的质量。那有没有什么方法，可以实现自动筛选呢。这里我们介绍一种方法：**TextRank** 算法。在第 1 章中我们已经介绍了使用 **TextRank** 算法进行关键词提取，本节我们仍然借鉴这个思路进行关键短语的提取。

对于 TextRank 算法的原理，这里就不再赘述了，下面就介绍具体实现的原理。

## 2.2.2 原理

这里我们沿用第 1 章中介绍使用 TextRank 提取关键词的例子，样例文本如下：

程序员（Programmer）是从事程序开发、维护的专业人员。一般将程序员分为程序设计人员和程序编码人员，但两者的界限并不非常清楚，特别是在中国。软件从业人员分为初级程序员、高级程序员、系统分析员和项目经理四大类。

假设对上面的文本分词后得到：程序员，程序设计，程序开发等词。首先利用 TextRank 算法进行关键词的重要性计算，这里计算的方式是区别于前面的。因为我们的短语是由关键词组成的，而按照语法的定义，词与词之间的组合顺序一般是从前往后的，所以在进行窗口选择的时候，我们只计算右半部分的窗口。在得到具体词的重要性权重之后，对于各个词按照权重从高到低排，按照排序后的词组从前往后，两两组合（这个长度可以自己设定，这里只是为了表述的方便，取二元组进行说明）。然后对照两个词在原文顺序中间隔的词个数，这些间隔的词即为给其投票的元素。再利用 TextRank 算法迭代求出最终的词组权重。

对于投票词的权重计算按照如下公式进行：

$$WS(V_{i-j}) = (1 - d) + d \cdot \sum_{V_k \in In(V_{i-j})} \frac{1}{|In(V_{i-j})| - 1} WS(V_k)$$

其中

$WS(V_{i-j})$ ：表示由词  $i$  和词  $j$  组合而成的词组的权重。

$d$ ：阻尼系数。

$V_k \in In(V_{i-j})$ ：表示原文中在词  $i$  和词  $j$  之间的单词，其中开头的词  $i$  和结尾的词  $j$  也包含在其中。

$|In(V_{i-j})|$ ：表示原文中在词  $i$  和词  $j$  之间的单词个数。

$WS(V_k)$ ：表示单词  $k$  的关键词权重。

这里虽然沿用了 TextRank 的框架，但是对于计算逻辑做了一些自己定义，所以这里也可以说是用改进的 TextRank 算法实现关键短语的提取。

## 2.2.3 实现

对于分词和由原文计算单个词的重要性权重部分，这里就不再赘述了，参见 1.2 节“TextRank 算法实现关键词抽取”部分。

在完成单个词的重要性权重计算之后，我们首先对各个单词按照权重，从高到低排序。完成排序之后，对各个词按照从前到后的顺序进行组合得到候选关键词短语。按照如上公式计算短语的权重。具体代码如图 2-3 所示：

```
//对关键词权重 map 进行排序，按照权重从高到低排序
List<Entry<String, Double>> rankList =
function.rankMapDescDouble(wordScoreMap);
String[] words = new String[rankList.size()];
int i = 0;
//将排序后的单词放到一个数组里面待处理，这样做是为了加快处理速度
for (Entry<String, Double> entry : rankList) {
    words[i] = entry.getKey();
    i++;
}
for (int j = 0; j < words.length-1; j++) {
    for (int j2 = j+1; j2 < words.length; j2++) {
        String w_i = words[j];
        //将 string 类型的单词转换成 int 型的整数
        int wordNumber_i = function.getWordOriginalIndex(w_i);
        int i_i = originalWordIndexs[wordNumber_i]; //获得词 i 在原文中的位置
        String w_j = words[j2];
        int wordNumber_j = function.getWordOriginalIndex(w_j);
        int i_j = originalWordIndexs[wordNumber_j]; //获得词 j 在原文中的位置
        int wordsLength = Math.abs(i_j - i_i) + 1; //计算两个单词之间间隔的词的
        个数
        double sum = 0.0;
        for (int k = i_i; k < i_j; k++) {
            sum += originalWordWeights[k];
        }
        double weightNum = (1-d) + d*sum/(wordsLength-1);
        resultMap.put(w_i+w_j, weightNum);
    }
}
```

图 2-3

1. 如图 2-3 第一行所示，先对单个单词的权重 `map` 进行排序，按照权重值由高到低进行排序。
2. 接着将各个单词，顺序地放到一个数组中，这样在后面进行遍历的时候，速度比直接访问 `list` 的速度要快一些。即图 2-3 代码的第 3~7 行。
3. 对于原文中单词的顺序和权重，我们分别建了一个一维的 `int` 数组——用于存储原文中各个单词的编号；和一个一维的 `double` 数组——用于存储原文中每个单词的关键词权重。所以在这里遍历的时候，我们首先将单词映射到一个 `int` 值，也即存储其序号的单元格位置。这样就可以直接取到单词的位置信息，再访问权重数组，就可以获得单词的权重。同时通过两个位置的差值，可以得到其间所包含的单词数量。也即上述代码第 10~16 行所实现的功能。
4. 最终按照我们这里提出的公式计算每个词组的权重，并放入到结果 `map` 中，待后续排序使用。当然这里只是演示了一层的计算，如果希望进行多层迭代，则只需要将每个词组中包含的单词对应的位置信息记录下来，然后在迭代的过程中重复第 3 步中的计算过程即可。

## 2.2.4 拓展

因为该计算是基于 `TextRank` 进行的，所以对于 `TextRank` 本身的缺点，该方法也都继承了，改进的思路依然是遵循前面的思路。

但是因为场景不一样了，这里涉及多个词，以及词到词的路径，所以改进的思路，也可以参照隐马尔科夫模型的思路，对词与词的组合进行计算，当然这里就涉及有监督和无监督的问题，所以孰优孰劣并不是那么清晰，只能说根据自己的情况，选择最适合自己的方法。

## 2.3 LDA 算法实现短语抽取

本节我们将介绍运用主题模型来实现关键短语的提取，因为从理解上看我们认为主题和短语有时会很相像，那么本节就详细介绍怎么利用主题模型提取文章的关键短语。

### 2.3.1 场景

在关键短语的抽取中，我们也希望抽取的短语更加符合语义的元素。比如对于如下的语句：

1. 鲜花多少钱？
2. 白百合多少钱？
3. 水仙花多少钱？

我们希望得到最终短语“鲜花多少钱”的权重最高，而不是三者的权重值一样。借鉴于关键词提取的方法，这里也介绍一种利用 LDA 进行关键短语提取的算法。

### 2.3.2 原理

由前面可知，LDA 本身是由文档、主题、词语三个层级组成的结构。而我们的短语则是由词组成的。所以借鉴 2.2 节中运用 TextRank 算法进行短语提取的思路：在关键词提取的基础上再嵌套一层处理。这里也是先运用 LDA 做词到文档的主题概率计算，再在词的基础上合并成短语，同时对短语进行优化和处理。具体步骤如下：

1. 利用训练文档集产生主题模型，预测新文档集的单词对主题的权重以及主题对文档的权重。这里公式就不再单独重复了，在下面步骤 2. 中的公式一并进行展示和说明。
2. 根据单词对主题的权重以及主题对文档的权重计算单词对文档的权重。计算公式如下：

$$P(w_i|D_m) = \sum_{k=1}^K P(w_i|T_k) \cdot P(T_k|D_m) = \sum_{k=1}^K \frac{C_{km} + \alpha}{\sum_{k=1}^K C_{km} + K \cdot \alpha} \cdot \frac{C_{ik} + \beta}{\sum_{i=1}^N C_{ik} + N \cdot \beta}$$

其中

$C_{ik}$ ：为语料库中单词  $i$  被赋予主题  $k$  的次数。

$N$ ：为词汇表的大小。

$\beta$ ：为超参数。

$C_{km}$ ：为语料库的文档  $m$  中单词被赋予主题  $k$  的次数。

$K$ : 为主题的数量。

$\alpha$ : 为超参数。

这一步在 1.3 节已经进行了详细的说明，这里就不再赘述了。

- 对单词按照先后顺序进行两两组合（这里也是为了方便说明，设定为两个单词组合，实际上可以根据自己的情况，自由设定），并计算其组合后的短语对文档的权重。这里也借鉴之前互信息计算组合词权重过程中存在的问题：对于低频词会产生比较大的权重。对于组合后的单词首先通过统计词频，选取词频大于指定阈值的短语。这里设置频次在前 75% 的词。
- 通过短语所属的主题进行短语聚合，去除相似度很高、但是概率较小的短语，只保留概率最大的短语。

(a) 由词计算短语相对于文档权重的公式：

$$P(H_r|D_m) = \sum_{w_i \in H_r} P(w_i|D_m)$$

其中

$P(H_r|D_m)$ : 表示短语  $H_r$  对于文档  $m$  的主题概率。

$H_r$ : 表示组合得到的任一短语  $r$ 。

$w_i$ : 表示词库中的任一单词。

$w_i \in H_r$ : 表示  $w_i$  是组成短语  $H_r$  的一个单词，也即对组成短语  $H_r$  的单词集合进行遍历汇总各个单词对于文档  $m$  的主题概率。

(b) 计算短语  $H$  的主题分布概率的公式：

$$P(H_r|T_k) = \sum_{w_i \in H_r} P(w_i|T_k)$$

其中

$P(H_r|T_k)$ : 表示短语  $H_r$  对于文档  $k$  的主题概率。

$H_r$ : 表示组合得到的任一短语。

$w_i$ : 表示词库中的任一单词。

$w_i \in H_r$ : 表示 $w_i$ 是组成短语 $H_r$ 的一个单词, 也即对组成短语 $H_r$ 的单词集合进行遍历、汇总各个单词对于主题 $k$ 的主题概率。

- (c) 上面计算的是给定主题 $k$ 、出现短语 $H_r$ 的概率, 而我们希望得到是当短语为 $H_r$ 的时候, 主题为 $k$ 的概率。这个刚好是(b)的对偶形式, 由贝叶斯公式可得, 短语 $H$ 的主题 $k$ 的概率公式:

$$P(T_k|H_r) = \frac{P(T_k)}{P(H_r)} \cdot P(H_r|T_k) = \frac{P(T_k)}{P(H_r)} \cdot \sum_{w_i \in H_r} P(w_i|T_k)$$

其中

$P(T_k|H_r)$ : 表示给定短语 $H_r$ 时, 其数据主题 $k$ 的概率。

$P(T_k)$ : 表示主题 $k$ 的概率。

$P(H_r)$ : 表示短语 $H_r$ 的概率。

- (d) 经过上述步骤, 我们会得到一个 $R \cdot K$ 维的矩阵。其中 $R$ 表示短语的个数,  $K$ 表示主题的个数。对于这个矩阵我们采用谱聚类的方式, 取出前 $n$ 个短语作为候选的关键短语。对于谱聚类的流程, 不是本章重点, 所以在这里不做详细说明。基本上有了相似度矩阵之后, 整个计算流程的核心步骤也算是完成了, 所以这里就介绍一下相似度矩阵计算的核心, 相似度算法, 有兴趣的读者可以找相关资料了解一下谱聚类的算法细节。其中短语和短语之间的相似度计算采用余弦相似度, 具体公式如下:

$$S(H_r, H_p) = \frac{\sum_{k=1}^K P(T_k|H_r) \cdot P(T_k|H_p)}{\sqrt{\sum_{k=1}^K P(T_k|H_r)^2} \cdot \sqrt{\sum_{k=1}^K P(T_k|H_p)^2}}$$

5. 经过上述步骤之后, 我们就对于待抽取的短语, 进行了按照主题聚类。然后对剩下的短语按照短语相对于文档的主题重要性进行排序和抽取。根据业务需要选出排名靠前的几个短语, 作为关键短语。

### 2.3.3 实现

就单个词对于文档的主题概率的计算来说, 就不做重复说明了。这里直接运用其计算

的结果，作为后续计算的中间变量。

下面对于计算短语的文档主题概率和由短语相对于主题的概率矩阵进行短语筛选进行详细的说明。

实现短语提取计算的代码如图 2-4 所示：

```

for (int i = 0; i < phrases.length; i++) {
String[] tStrings = phrases[i].split("-");//将用“-”连接的两个词分开
if (tStrings.length != 2) {
    continue;
}
String w_i = tStrings[0];
//将 string 类型的单词转换成 int 型的整数
int wordNumber_i = function.getWordOriginalIndex(w_i);
//获得词 i 在原文中的位置
int i_i = originalWordIndexs[wordNumber_i];
String w_j = tStrings[1];
int wordNumber_j = function.getWordOriginalIndex(w_j);
//获得词 j 在原文中的位置
int i_j = originalWordIndexs[wordNumber_j];
//获得短语对于指定文档 m 的主题概率权重
result[i] = word2DocWeight[i_i][m] + word2DocWeight[i_j][m];
for (int j = 0; j < word2TopicWeight[i_i].length; j++) {
    //获得主题 k 在训练文档中出现的次数，作为 P(T_k) 的值
    int topicNumber = topicCount[j];
    //获得短语 r 在待提取文档中出现的次数，作为 P(H_r) 的值
    int phraseNumber = phraseCount[i];
    double weight =
(word2DocWeight[i_i][j]+word2DocWeight[i_j][j])*topicNumber/phraseNumber;
    //将给定短语 H_r 时，其数据主题 k 的概率存入到结果矩阵中
    phrase2TopicWeight[i][j] = weight;
}
}
//通过谱聚类获得候选短语的列表
List<Integer> selectIndexs = Spectrum.cluster(phrase2TopicWeight,
clusterSize);
for (Integer index : selectIndexs) {
resultMap.put(phrases[index], result[index]);
}
}

```

图 2-4

1. 如图 2-4 中所示，先对预先按照“词 1-词 2”形式存储的短语进行分割，获得短语构成的词集合。即图 2-4 中第 1~5 行代码所示。
2. 在获得短语的组成单词之后，通过文本到数字的转换函数，获得各个单词对应的数字编码，根据这个数字编码，可以找到单词在原文中的序列号，通过这个序列号，可以获得单词对应的文档、主题的概率。根据由单词的文档概率计算短语的文档概率公式，计算得到短语对指定文档的主题概率。如图 2-4 中第 6~12 行代码所示。
3. 接下来就要计算短语对于每一个主题的概率。因为最终我们对短语聚类依赖于短语对于主题的条件概率，所以在这一步首先计算短语对于主题的概率，然后根据贝叶斯公式计算短语对于主题的条件概率，如图 2-4 中第 14~17 行代码所示。
4. 经过第 3 步，我们就得到了  $R \cdot K$  维短语对于主题的条件概率矩阵。对这个矩阵进行谱聚类，同时根据预先设定的聚类个数，返回最终具有代表性的短语编号集合，这个集合就作为最终的候选集合，然后根据这些短语的文档主题概率进行排序，按照需要选定排名前几位的短语作为关键短语即可。

### 2.3.4 拓展

在最开始的时候，我们希望最终得到的结果是“鲜花多少钱”的概率最大，先不讨论这个结论是不是一定正确。从实际计算结果上来看，上面的这个方法未必一定能返回我们想要的结果。这主要是由于算法本身的局限造成的。因为我这里是按照统计概率的方式，对文档、主题、词语（短语）进行抽样统计，所以在抽样的数据里面，鲜花是不是能明显地比白百合有更高的权重，这就依赖于样本数据的分布。

针对这种情况，我们可以通过模式识别的方式对词语上下文结构和惯性搭配先进行一个挖掘，然后根据挖掘的结果构建词语的关联关系，对上述结果进行校正，能达到更好的语义效果。当然这也就涉及另外的一个领域，符号表示学。比如我们可以定义一个结构“A 是 B 的一种”。那么只要符合这个结构的，我们都可以定义为  $B \in A$ 。这样当 A 和 B 出现同样的结构时，就可以根据业务场景直接进行替换合并。比如上文的例子：

1. 鲜花多少钱？
2. 白百合多少钱？
3. 水仙花多少钱？

如果在语料中我们知道：白百合是鲜花的一种；水仙花是鲜花的一种。那么如果我们当前的场景是我们需要按照泛类的主题进行短语抽取的话，那么就可以直接将白百合和水仙花替换成其父类鲜花，那么最终的关键短语就很明显了，便是：“鲜花多少钱”。

当然这里只是举一个很简单的例子，这个学科本身还是非常复杂的，有兴趣的读者可以深入了解，这里就不再展开讨论了。

## 第 3 章

# 自动摘要抽取模型

在完成了关键词、关键短语的提取之后，我们就希望对文本进行更进一步的操作。本章就介绍其中的自动摘要技术。通过自动摘要技术，我们可以实现文章梗概的自动提取，可以对文章进行语句层面的表达和概括。

## 3.1 决策树算法实现自动摘要

本节将介绍利用决策树算法实现对文章摘要的自动提取，该算法虽然简单，但是如果特征提取得恰当，仍然能够实现很好的效果。下面就对具体的实现进行详细的介绍。

### 3.1.1 场景

假设我们已经有 100 篇文章，以及对应的文章摘要的语料数据。现在新来了一篇文章，需要让机器实现自动摘要，有什么方法可以实现呢？这里介绍一种基本的分类算法来实现这个功能：决策树算法。

决策树其实可以分为分类树和回归树两类，分类树是指输出每个样本的类别，而回归树则是指输出数值结果，这里我们只讨论分类树。

在应用中，决策树通常是基于一套规则去将数据分门别类。在一个数据集中，决策树算法会利用每一个样本的属性变量，并确定哪一个属性是最重要的，然后给出一系列决策去最优地将数据划分成多个子集。

### 3.1.2 原理

首先我们用一个树形的图来简要地说明决策树实现的过程。假设我们对一篇文章中的每一句话都按照如表 3-1 所示的几个维度进行了标注。

表 3-1

句子编号\特征	是否在段首	是否在段尾	是否 TOP10 高频词	是否包含总结词	是否包含比较词
1	是	否	是	否	否
2	否	否	是	是	是
.....					

同时我们根据统计的数据发现如表 3-2 所示的规律。

表 3-2

特征\分布	语句包含在摘要中的比例	语句不包含在摘要中的比例
在段首	60%	40%
不在段首	10%	90%

续表

特征\分布	语句包含在摘要中的比例	语句不包含在摘要中的比例
在段尾	65%	35%
不在段尾	15%	85%
.....		

从上面的统计数据，我们从直观上看会有一个感觉，就是如果这个句子在段首或者段尾，那么就有很大的概率成为摘要中的组成句子。假如我们对所有的特征进行逐级分解，之后便得到如图 3-1 所示的一个树形的分类概率图。

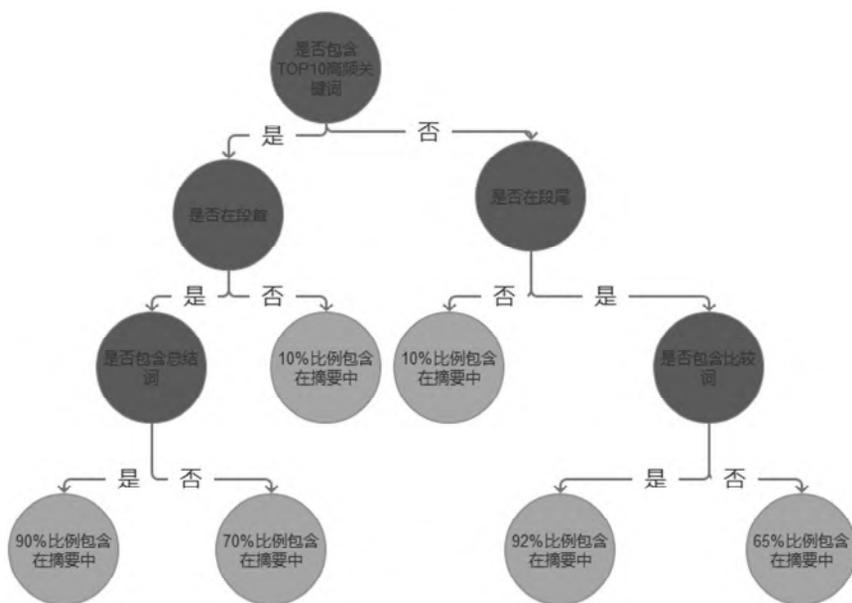


图 3-1

有了上面这样的一棵统计树，对于新来的语句，我们只要按照上面的特征对句子进行细分，最终句子落在哪个黄色节点上，就将相应的概率赋给该句子，等所有的句子都标记完成以后，按照概率的大小排序，选出前面的几个句子作为摘要，就完成了摘要的自动提取。

上面只是粗略地介绍了运用决策树实现自动摘要的大概流程，下面就对具体的步骤进行详细的说明。

1. 如图 3-1 所示，在整个训练的过程中，最重要的一步就是对数据按照特征值进行分裂。要进行这个步骤，就要提到一个我们之前已经介绍的概念：信息熵。信息熵能够很好地度量一个集合中元素的纯度，如果样本纯度越高，那么信息熵就越小。对

应到我们的场景,如果样本在某个特征下全都不是一类(属于摘要或者不属于摘要)的,按照这个特征对样本进行分类的话,最终的结果就是我们想要的。所以在决策树进行特征值分裂的时候,就是根据信息熵来进行的。下面重温一下信息熵的公式:

$$\text{info}(S) = - \sum_{c=1}^C p_c \log(p_c)$$

其中,

$\text{info}(S)$ : 表示样本 (Sample) 的信息熵。

$C$ : 表示类别的总数。

$p_c$ : 表示样本 (Sample) 中  $c$  类出现的概率。

2. 由于对于单个特征对样本分割的信息熵,等于该特征下每一个特征值的信息熵之和,所以单个特征值的信息熵计算公式如下:

$$\text{info}(F_v) = \sum_{i=1}^V p(F_{v_i}) \cdot \text{info}(F_{v_i}) = \sum_{i=1}^V p(F_{v_i}) \cdot - \sum_{c=1}^C p_c \log(p_c)$$

其中,

$\text{info}(F_v)$ : 表示单个特征  $v$  的信息熵。

$p(F_{v_i})$ : 表示特征  $v$  的值  $i$  发生的概率。

$p_c$ : 表示在特征  $v$  的值  $i$  下,类别  $c$  发生的概率。

3. 特征值的分裂是根据上下两层信息熵的差值大小来决定的:选择差值最大的特征值作为下一层分割的节点。而对于第一层特征值的筛选,是通过单个特征值的信息熵与整体样本的信息熵的差值来选择的。具体计算公式如下:

$$\text{gain}(F_v) = \text{info}(S) - \text{info}(F_v)$$

其中,

$\text{gain}(F_v)$ : 表示选择特征值  $v$  作为分割节点时,所获得的信息增益。

所以信息增益是一个介于  $(-1,1)$  之间的值,最终的分割节点为信息增益最大的特征值。

### 3.1.3 实例

实现信息增益计算的代码如图 3-2 所示:

```
// 选择剩余属性中信息增益最大的作为下一个分类的属性
for (int i = 0; i < remainAttr.size(); i++) {
    //按照信息增益的值来比
    tempValue = computeGain(remainData, remainAttr.get(i));

    if (tempValue > gainValue) {
        gainValue = tempValue;
        attrName = remainAttr.get(i);
    }
}
```

图 3-2

1. 在每一层计算的时候,都对剩余的属性值进行了遍历,对于当前分支下,之前已经处理过的属性值就不再进行计算了。
2. 计算的方式是按照信息增益的方式进行的,即通过计算当前属性值的信息熵与其父节点的信息熵差得到。同样的选择还有将最大信息增益的节点作为当前层的分裂属性。

实现树的循环构建计算的代码如图 3-3 所示:

```
// 移除非此值类型的数据
rData = removeData(remainData, attrName, valueTypes.get(i));

childNode[i] = new AttrNode();
boolean sameClass = true;
ArrayList<String> indexArray = new ArrayList<String>();
for (int k = 1; k < rData.length; k++) {
    indexArray.add(rData[k][0]);
    // 判断是否为同一类的
    if (!rData[k][attrNames.length - 1]
```

图 3-3

```
.equals(rData[l][attrNames.length - 1])) {  
    // 只要有 1 个不相等, 就不是同类型的  
    sameClass = false;  
    break;  
}  
}  
  
if (!sameClass) {  
    // 创建新的对象属性, 引用同一个对象会出错  
    ArrayList<String> rAttr = new ArrayList<String>();  
    for (String str : remainAttr) {  
        rAttr.add(str);  
    }  
  
    buildDecisionTree(childNode[i], valueTypes.get(i), rData,  
rAttr, isID3);  
}
```

图 3-3 (续)

1. 在完成当前层属性值分裂之后, 就重新筛选新建立的分支、接下来便进行数据集的计算。也就是图 3-3 中第 14 行代码所示, 先筛选属性值为当前分裂节点值下的数据, 然后按照分裂属性不同的值, 构建不同的分支。
2. 然后判断新分支下的数据是不是都是同一类别的, 如果是同一类别的, 则停止该分支的分裂, 直接存储为分类节点。如果不是, 则重复上述过程直到整个过程结束为止。
3. 从第 2 步的过程, 很容易发现对于每一个分支最终都会停止在如下的两种情况下:
  - a) 当前分支将所有属性值都遍历完了;
  - b) 当前分支的数据只有一个类别。

但这两种情况, 就会造成一个问题, 分裂过度 (会出现过拟合), 计算深度不可控 (导致整个计算过程的不可控)。

4. 针对分裂过度的情况, 一般会采取修枝的策略, 控制分枝的数量。针对树的深度不可控的情况, 就需要对树的延伸深度进行限制, 以便能够确定算法在可控的时间内完成计算。由于篇幅原因, 更多的细节就不在这里表述了, 有兴趣的读者可以参考其他资料进行深入学习。

### 3.1.4 拓展

构造决策树的关键步骤是分裂属性。而其最终的分裂结果，则取决于选择的分裂度量准则。分裂度量准则的算法有很多，一般使用自顶向下递归分治法，并采用不回溯的贪心策略。本文中使用的信息增益是 ID3 准则，和其对应的是 C4.5 算法。

ID3 算法存在一个问题，就是偏向于多值属性作为分裂属性，这样虽然使得划分充分纯净，但这种划分对于分类几乎是毫无用处的。作为 ID3 的后继算法，C4.5 算法使用增益率 (gain ratio) 作为判断准则，试图克服这个偏倚。

C4.5 算法首先定义了“分裂信息”，其定义可以表示成：

$$\text{SplitInfo}(F_v) = - \sum_{i=1}^v \frac{|D_i|}{|D|} \log\left(\frac{|D_i|}{|D|}\right)$$

其中，

$v$ : 表示属性  $F_v$  下的值的个数。

$|D_i|$ : 表示属性  $F_v$  的值  $i$  对应的样本个数。

$|D|$ : 表示总体样本的个数。

增益率被定义为：

$$\text{gainRatio} = \frac{\text{gain}(F_v)}{\text{SplitInfo}(F_v)}$$

C4.5 算法是将信息增益进行了规范化，这样就可以实现不同数量分裂值的属性都可以在同一个水平上进行比较。

C4.5 作为 ID3 的改进算法，其整体的分裂策略和剪枝策略与 ID3 基本一致，在这里就不再赘述了。

## 3.2 基于逻辑回归算法实现自动摘要

本节要介绍的算法为逻辑回归算法，逻辑回归算法在分类中有较广的应用场景，特别是对于二分类，有很好的应用基础。下面就其在文章摘要中的应用进行详细的介绍。

### 3.2.1 场景

如 3.1 节所述，自动摘要可以看作对句子的一个分类问题：对每一个句子，我们要判断属于/不属于摘要。从这个角度来看，对于自动摘要这个问题的解法，常用的分类算法都可以实现，上面介绍了决策树的分类算法，本节再介绍另一种基本算法：逻辑（logistic）回归算法。

逻辑回归本质上是线性回归，只是在特征到结果的映射中加入了一层函数映射，即先将特征线性求和，然后再使用一个被称为核函数的转换器 $g(z)$ （一般形式为 $g(z) = \frac{1}{1+e^{-z}}$ ），将最终的结果转换到假设函数来预测。核函数 $g(z)$ 可以将线性求和值连续映射到 0 和 1 上。

### 3.2.2 原理

因为逻辑回归的本质是一个线性回归，所以它的基本形式和线性回归的形式是一致的，即 $y=w \cdot x+b$ 。当 $x$ 是一个向量（即多维）的时候， $w$ 也就是一个向量了。如上所述，逻辑回归与一般线性回归的差别在于，它增加了一个转换器，并且这个转换器是作用在线性回归的值上面的。所以其最终的表达式如下：

$$f(x) = g(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}}$$

其中，

$\theta^T$ ：表示各个特征值对应的权重向量的转置。这个向量中包括了常数（即 bias）的权重。根据 $x$ 表达形式的不同，这个值在向量的第一个或者最后一个。

$x$ ：表示特征值向量。一般向量都是用大写字母表示的，这里为了沿袭上面的线性回归表达式，所以就还是用小写字母表示。

$e$ ：是一个数学常量，是自然对数的底数，值约为 2.71828。在一般编程语言中记为 E 或者 exp。

有了上面的公式，我们会发现整个计算很简单，只要知道了各个特征值的权重之后，将特征值和对应的权重相乘再加和，然后经过函数变换就可以了。那么现在唯一要做的就是求解特征值对应的权重。按照国际惯例，下面就要进行一大段非常严谨的数学推导，告诉你如何确立损失函数，然后利用最大似然、求导、梯度下降法求解最优的参数值。但是这不符合我们的风格和初衷，我们是希望用尽可能简单的公式和最浅显的例子，让大家能

用起来这个算法。所以我们这里直接给出最终的结论，因为那一大段的数学推导既不会出现在程序中，也不影响程序运行。所以本着实用简洁的原则，这里就直接给出程序运行所需的结论，公式如下：

$$\theta_j^{k+1} = \theta_j^k + \alpha(y^i - f(x^i))x_j^i$$

其中，

$\theta_j^{k+1}$ ：表示第  $j$  个权重值第  $k+1$  次迭代时的值， $\theta_j^k$  表示其第  $k$  次迭代。

$\alpha$ ：是学习速率，即每一个实用误差校正参数的比例。

$y^i$ ：第  $i$  行数据对应的  $y$  值。 $x^i$  就表示第  $i$  行的变量值。

$f(x^i)$ ：表示第  $i$  变量值对应的预测值。

$x_j^i$ ：表示第  $i$  行变量中第  $j$  列的值。

有了上面这个公式，我们就可以用训练数据、迭代计算求出最优的权重值。当然这个迭代过程也有一个问题要解决：何时停止？因为问题具有共性，所以解决的方法也是算法圈中通用的办法，通常包含下面两种策略：

1. 前后两次迭代差小于阈值，即  $|\theta_j^{k+1} - \theta_j^k| < \varepsilon$ ；
2. 达到最大迭代次数  $N$ 。

如果满足以上两条中的任一条则结束整个迭代过程，将最终得到的权重值作为最终的参数。下面就介绍具体的实现代码。

### 3.2.3 实现

沿用 3.1 节的例子，对于一篇文章中的每一个句子，我们构建如表 3-3 所示的特征数据。

表 3-3

句子编号\ 特征	在段中是 第几句	是否在段尾	包含 TOP10 高频词的 个数	包含总结词的 个数	包含比较词的 个数	结果：是否包含 在摘要中
1	2	0	3	0	2	1
2	1	0	1	2	3	0
...						

因为逻辑回归处理需要输入的变量是数字型的变量，所以对于变量的设计和值的表达要进行转换，比如对于“是/否”类的分类变量，我们需要按照值进行数字枚举，以这里为例，我们将“是”对应为 1，“否”对应为 0。而对于其他的可转换成统计值的变量，则尽可能地转换成数值计数的变量。比如对于“包含 TOP10 高频词”这个特征，在 3.1 节我们使用的是“是否包含 TOP10 高频词”这样的分类变量，而在这一节，我们则使用了“包含 TOP10 高频词的个数”这样的连续型变量。

经过上面的处理之后，我们就得到了基本的训练数据，接下来就是运用训练数据，求出具体的参数，实现逻辑回归特征值求解计算的代码如图 3-4 所示。

```
for (int i = 0; i < max_iter; ++i) { //设置最大的循环次数
    double[] difWeights = new double[weights.length];
    for (int j = 0; j < y.length; j++) { //对所有行进行遍历
        double tmp = 0.0;
        for (int j2 = 0; j2 < weights.length; j2++) {
            tmp += weights[j2]*x[j][j2];
        }
        //计算当前行的误差
        double error = y[j]- function.sigmoid(tmp);
        for (int j2 = 0; j2 < weights.length; j2++) { //更新权重
            weights[j2] += alpha*error*x[j][j2];
            //记录当前循环所调整的权重误差之和
            difWeights[j2] += alpha*error*x[j][j2];
        }
    }
    double max_dif = Double.MIN_VALUE;
    //计算更新权重的最大误差。
    for (int j = 0; j < difWeights.length; j++) {
        double tmp = difWeights[j] / y.length;
        if (tmp > max_dif) {
            max_dif = tmp;
        }
    }
    //如果所有权值的前后两次更新没有太大的变化，则结束循环
    if (max_dif < epsion) {
        break;
    }
}
```

图 3-4

1. 首先对样本数据进行全部遍历，并且在每一步遍历的时候，分别对  $x$  变量和对应的权值求乘积的和、对乘积的和进行 sigmoid 变换、计算变换值与  $y$  值的差值、根据误差值更新权重。如图 3-4 中第 3~10 行代码所示。
2. 在更新权值的同时，记录下当前第  $j$  个权重更新的数量，在整个数据遍历完成以后，对所更新的权重计算平均值，得到此轮循环对最终权值的影响，如果更新的值小于预设的阈值，则结束循环。
3. 这里要特别说明一点的是，我们这里采用的权重更新方式为单步的梯度下降更新，也即每输入一行记录都更新一次权重。这种更新的方式，好处是处理的数据不会很多，所以对于数据量很大的情况，很好用，因为使用的内存很小。但是它有一个缺点就是更新权重的时候，不能保证整个过程一定都是最优的，也即不能保证更新以后的误差一定小于之前的误差。但是如果循环的次数足够多，最终也能达到最优的解。所以针对这个缺点就会有另外一种更新权重的方式，也就是全局的权重更新，这种方法不是在每输入一行记录就更新一次权值，而是在全部数据输入完成以后更新权值，具体的实现代码如图 3-5 所示。

```
for (int i = 0; i < max_iter; ++i) { //设置最大的循环次数
    //进行全局梯度下降的时候需要将每一行的误差存储起来，然后在最后进行计算和处理
    double[] errorArray = new double[y.length];
    double[] difWeights = new double[weights.length];
    for (int j = 0; j < y.length; j++) { //对所有行进行遍历
        double tmp = 0.0;
        for (int j2 = 0; j2 < weights.length; j2++) {
            tmp += weights[j2]*x[j][j2];
        }
        //计算当前行的误差
        double error = y[j]- function.sigmoid(tmp);
        errorArray[j] = error;
    }
    //待所有记录数据的误差都计算完成之后，再更新权重
    for (int j = 0; j < x[0].length; j++) {
        double tmp = 0.0;
        for (int j2 = 0; j2 < x.length; j2++) {
            tmp += x[j2][j]*errorArray[j2];
        }
        weights[j] += alpha*tmp;
    }
}
```

图 3-5

```
        difWeights[j] = alpha*tmp;
    }
    double max_dif = Double.MIN_VALUE;
    //计算更新权重的最大误差。
    for (int j = 0; j < difWeights.length; j++) {
        double tmp = difWeights[j];
        if (tmp > max_dif) {
            max_dif = tmp;
        }
    }
    //如果所有权值的前后两次更新没有太大的变化，则结束循环
    if (max_dif < epsion) {
        break;
    }
}
```

图 3-5 (续)

4. 全局更新的缺点就是需要占用较大的内存，当数据量非常大的时候，就无法进行计算了。当然这时理论上应该有一个折中的办法，那就是 **batch** 梯度下降，每一次取一部分数据，既克服了单个数据的数量太少导致优化路径不稳定的缺点，又可以避免占用过大内存。当然难点也就是如何选取合适的 **batch** 数量。因为这涉及另外一个话题：如何选择最优化方案，所以这里不作过多的说明，有兴趣的读者可以参考其他资料进行深入学习。

### 3.2.4 拓展

逻辑回归的缺点也是很明显的，那就是它是一个线性分类器，所以对于变量中如果存在多个高度相关的变量时，其预测值就会受这些因素的严重影响，导致最终的预测结果不准确。所以可以在数据预处理的过程中利用因子分析或者变量聚类分析等手段来选择代表性的自变量，以减少候选变量之间的相关性。

逻辑回归的另外一个问题就是：预测结果呈“S”型，也就是说，转化器在将预测值向概率转化的过程是非线性的，在两端随着预测值的变化概率变化很小，边际值太小，而中间概率的变化很大，很敏感。导致很多区间的变量变化对目标概率的影响没有区分度，无法确定阈值。针对这种情况，可以通过选择一个随着预测值线性变化的核函数，这样就可以针对两端的值也进行有效的反馈，但是需要注意的是，要排除异常值对模型权重的影响。

同大多数线性回归的方法遇到的问题一样，这里需要有大量的特征输入，当输入的特征数量越大时，最终得到的矩阵就越可能是满秩方阵，对于满秩方阵，我们知道一定有一个最优且唯一的解可以使得最终的误差为 0。那难点也就在于如何选择足够多的特征值的问题。这是特征工程主要解决的问题。当然随着深度学习的使用，这种困难或许不再是困难，因为我们可以将隐含层设置一个够大的扩充向量，将原来的特征空间进行扩充，并且随着网络深度的增加，这个空间也会变得更大。

### 3.3 贝叶斯算法实现自动摘要

贝叶斯算法是基于贝叶斯理论提出的一种分类算法。本节所介绍的算法为朴素贝叶斯算法。下面我们就看看该算法在文本自动摘要的应用中具体实现的细节。

#### 3.3.1 场景

假设我们对 100 篇文章的摘要进行了很多的统计，比如统计了其单词出现在文章不同位置的频率、高频词汇出现在文章不同位置的频率等。有没有一种方法，可以让我们基于这些统计值，就可以实现对语句的判断：判断其是否属于摘要。而不需要做复杂的矩阵、信息熵、信息增益率等运算。这里我们就介绍这样一种方法：朴素贝叶斯算法。或者称其为一个框架也可以，因为在之前的章节中，我们已经使用过其核心公式了。

贝叶斯学派是统计学的一个重要学派，其与经典概率学派的一个最大的区别就是提出了先验概率的概念。在经典概率学派的理论中，同一个事件发生的概率是固定的，比如扔硬币这件事情，你扔和我扔得到正面的概率是一样的。但是贝叶斯学派认为不一样。比如一个有经验的人扔和一个新手扔得到正面的概率就会有很大区别。并且对于先验概率的计算，也给出了一个简单的计算方式：即为这个事件在历史上出现的概率作为先验概率的一个估计值。比如要计算我扔硬币得到正面概率的先验值，就把我历史上投硬币的情况进行统计，出现正面的概率即为我扔硬币得到正面的先验概率值。下面就对其在分类中的应用进行详细的说明。

#### 3.3.2 原理

朴素贝叶斯分类的流程可以用图 3-6 表示：

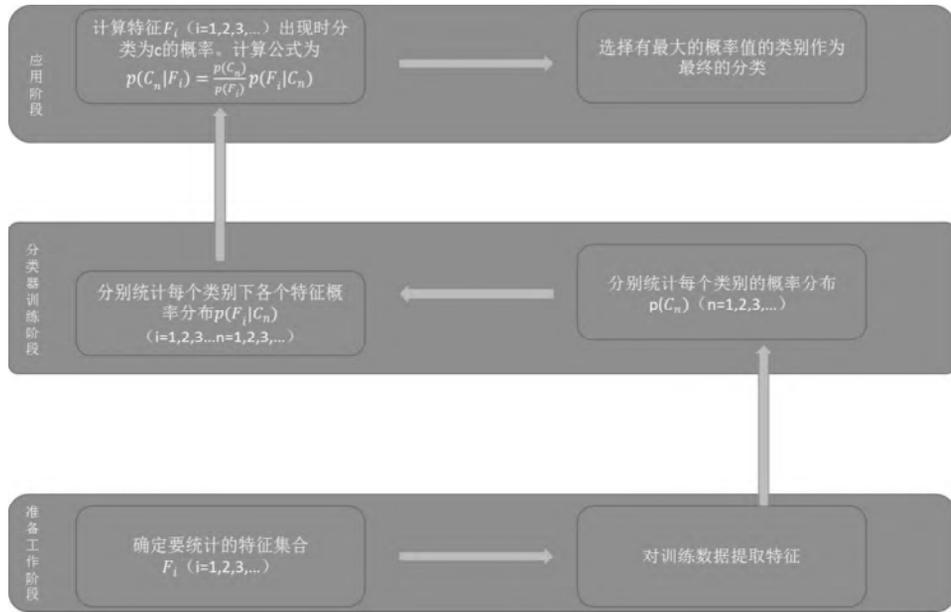


图 3-6

可以看到，整个朴素贝叶斯分类分为三个阶段。

**第一阶段：准备工作阶段。**这个阶段的一项工作是为统计工作准备素材，即确定要统计的指标。另一个是对训练数据按照确定的指标进行统计。这个步骤也是传说中的特征工程部分。对于指标的确定既可以完全由人工确定，也可以确定一些规则、实现半自动化的指标提取。比如确定了一个规则为：统计元素的位置信息。那么对于单词、句子、高频词等只要类型为元素的都要统计其位置，当然对于位置的类型也可以预先定义，这样就可以实现自动的特征提取。

**第二阶段：分类器训练阶段。**这个阶段的任务就是生成分类器，主要工作是计算每个类别在训练样本中出现的频率及每个特征属性划分对每个类别的条件概率估计，并将结果记录下来。这个阶段也可以做一些特征的筛选工作，比如得到每个特征属性划分对每个类别的条件概率之后，可以根据 TF-IDF 公式，对其概率进行进一步的优化调整。具体公式如下：

$$p(F_i|C_n) = p(F_i|C_n) \cdot \log\left(\frac{N + \alpha}{k}\right)$$

其中，

$N$ ：为总的类别个数。

$K$ : 为特征 (feature)  $i$  出现的类别数。

$\alpha$ : 因为有很多词都可能同时出现在所有的类里面, 但是这些词的概率又不能为 0, 所以就加上了一个平滑引子, 保证这类情况的值比较小, 但又不为 0。

第三阶段: 应用阶段。这个阶段就是对待分类的数据, 先根据在训练阶段确定的指标进行统计, 获得其在各个指标下的具体值, 然后根据具体的值计算其属于各个类别的概率。再然后将其归到概率值最大的类别中。

在对特征进行处理的时候, 我们只讲了统计的方法, 这种方法对于连续变量就会显得不是很好用。当特征属性为离散值时, 倒是很方便。对于连续变量, 其概率计算也有相应的处理方式, 具体公式如下:

$$p(F_i|C_n) = \text{Distribution}(F_i)$$

其中,

$\text{Distribution}(F_i)$ : 为特征  $i$  的概率分布密度函数。

假定特征  $i$  的值服从高斯分布 (也称正态分布)。那么有:

$$p(F_i|C_n) = \text{Distribution}(F_i) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(F_i-\mu)^2}{2\sigma^2}}$$

其中,

$\sigma$ : 为特征  $i$  在类别  $n$  下各个值的标准差。

$\mu$ : 为特征  $i$  在类别  $n$  下各个值的均值。

因此只要计算出训练样本的各个类别中此特征项划分的各均值和标准差, 代入上述公式即可得到需要的估计值。均值与标准差的计算在此不再赘述。

### 3.3.3 代码

实现贝叶斯特征计算的代码如图 3-7 所示:

```
for (int i = 1; i < data.length; i++) {
    // data 数据按照 yes 和 no 分类
    if (data[i][attrNames.length - 1].equals(YES)) {
        yClassData.add(data[i]);
    } else {
        nClassData.add(data[i]);
    }
}
//首先按照类别对数据进行分类, 然后分别统计各个类别下不同特征的概率
if (classType.equals(YES)) {
    classData = yClassData;
} else {
    classData = nClassData;
}

// 如果没有指定具体特征的名称, 则统计整个类的概率
if (condition == null) {
    return 1.0 * classData.size() / (data.length - 1);
}

// 寻找指定特征条件的属性列
attrIndex = getConditionAttrName(condition);

for (String[] s : classData) {
    if (s[attrIndex].equals(condition)) {
        count++;
    }
}

return 1.0 * count / classData.size();
```

图 3-7

1. 图 3-7 所展示的是计算样本数据的条件概率 $p(F_i|C_n)$ 的代码。
2. 参数传入特征  $i$  和所属类别的值时, 如果传入的属性值为空, 那么就计算类别的概率:  $p(C_n)$ 。如果传入了具体的特征名称, 那么就计算在指定类别下指定特征的概率:  $p(F_i|C_n)$ 。
3. 从上述代码可以看出, 整个过程只涉及一个计数的统计计算, 所以整个实现过程很简单, 实现对输入数据的分类概率特征计算的代码如图 3-8 所示:

```
for (int i = 0; i < dataFeatures.length; i++) {  
    // 因为朴素贝叶斯算法是类条件独立的，所以可以进行累积的计算  
    xWhenYes *= computeConditionProbablyly(dataFeatures[i], YES);  
    xWhenNo *= computeConditionProbablyly(dataFeatures[i], NO);  
}  
  
pYes = xWhenYes * computeConditionProbablyly(null, YES);  
pNo = xWhenNo * computeConditionProbablyly(null, NO);  
  
return (pYes > pNo ? YES : NO);
```

图 3-8

1. 首先计算测试数据中各个特征在不同类别下的联合条件概率，即图 3-8 中第 3~4 行代码所示。具体实现就是图 3-7 所展示的方法。
2. 然后根据贝叶斯条件概率公式计算测试数据分别属于 YES 类和 NO 类的概率。这里大家仔细看会发现少了一部分，即最终的概率只有贝叶斯条件概率中的分子部分，而没有分母部分。这是因为对于两个类别，分母部分都是一样的，而最终笔者也只是比较一下大小，而不是要看具体的值，所以分母就可以忽略了。

## 第 4 章

# 深度学习——计算任意词距离模型

前面 3 章分别介绍了关键词、关键短语、自动摘要（也可以理解为关键语句）的一些提取技术。接下来我们会介绍对文本基础元素的一些处理技术。本章我们就从词与词之间距离的计算角度来介绍一些相关的处理技术。

## 4.1 FP-Growth 算法实现词距离计算

本节我们将介绍使用频繁模式挖掘的方法来计算词与词之间的距离，本节介绍的是其中一种有代表性的算法，FP-Growth 算法，下面我们就对其具体的实现细节进行说明。

### 4.1.1 场景

首先我们看一下，下面这几句话：

1. 今天我吃了一个苹果。
2. 今天我吃了一个梨。
3. 我今天去打羽毛球了。
4. 我今天去打篮球了。

我们看到前两个句子是一个表达方式：“我吃了 xxx”。后两句是一个表达方式：“我玩了 xxx”。从这个句法上来看，吃的对象间的距离，肯定要小于玩的对象间的距离。

同时上面的句式又有一定的特殊性，即句式非常整齐，有很强的规律性。而对于这种句式的挖掘，有一类算法非常擅长，那就是频繁模式挖掘算法。这里我们就介绍其中的一种算法：FP-Growth 算法。

### 4.1.2 原理

1. 从这类算法的名字，我们就可以看出这一类算法的一个主要功能就是找出“频繁”出现的模式。至于出现多少次才算频繁，这是由模型中的一个参数定义的，这个参数就是：频数。
2. 如果仅仅使用频数，那么就会出现用数量进行统计时的问题：一叶障目不见泰山。我们仅仅知道统计的两者经常会一起出现，但是不知道这两者单独出现或者与其他项共同出现时的情况。所以就要采用一种通用的做法，对每一个组合加上一个比例，看看这种情况的出现，是否就是两个组成项的主要模式。

3. 传统的 FP-Growth 算法有两个步骤，第一步是对所有的数据进行遍历，按照每一行中元素出现的频次由高到低对元素进行排序。第二步是对排好序的元素构建 FP-TREE。由上面的例子构建的树形图如图 4-1 所示。

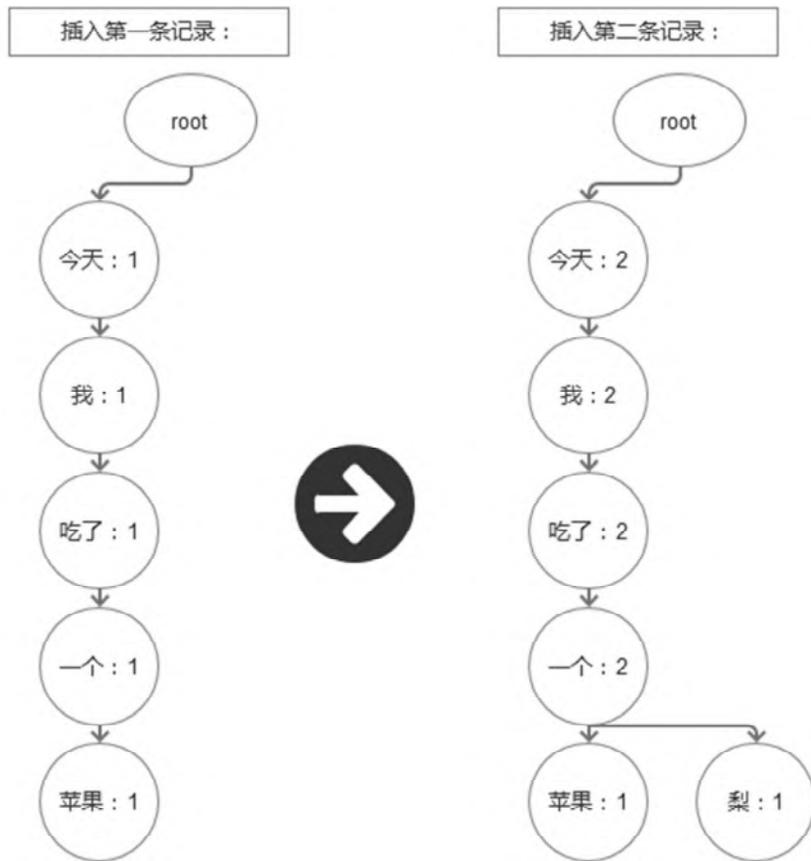


图 4-1

4. 传统的 FP-Growth 算法在获得 FP-TREE 之后，只要按照频数从高到低对各个元素组合进行排序就可以得到频繁项，比如图 4-1 中的[今天, 我, 吃了, 一个]都是频数为 2 的频繁组合项。它们之间相互推出的支持度（置信度）也是 1.0，即出现的情况都是百分百的。但是我们的最终目的不是求出相互频繁组合在一起的项，而是求出词与词之间的距离。所以在算法上，我们要做一些调整。
5. 首先在算法上，传统的 FP-Growth 算法是不关心词与词之间的顺序的，但是我们的场景是需要关心词与词之间顺序的。所以对于传统算法中第一步的统计元素频率，

我们这里就不需要了。

6. 对于每一个断句，我们当作一条记录。然后根据将记录中的元素从左向右构建 FP-TREE。构建完成以后，提取频繁项，得到频繁组合的元素集合。同时根据提取元素的形式将结果分为两大类。一类是直接相连的元素集合，比如{今天，我们}。另一类是句式类型的，比如{我今天去打 xxx 了}。对于这两种的交集的部分，我们采用最大匹配串的方式，即对于“我今天去打羽毛球了、我今天去打篮球了”，我们会选择最长的匹配串“我今天去打 xxx 了”，而{我，今天}这样的频繁项都会被忽略。如果是两种情况都有，则以次数多的情况为主。
7. 接下来就要到我们最终的目标了：确定两个词之间的距离。针对上面两种情况，我们对词的距离也有两种计算方式。针对第一种形式，我们在计算两个词的距离的时候，采用互信息的方式进行。对于第二种形式，我们要计算的是句式中不同元素的词距离，而不是相同元素的距离。对于不同元素的词距离计算，我们首先通过词性过滤，选出其中为名词的元素，然后计算这些元素之间的互信息作为它们的词距离。

### 4.1.3 实现

实现 FP-TREE 构建的代码如图 4-2 所示：

```
// 根据事务记录，一步步构建 FP 树
for (ArrayList<TreeNode> array : transctionList) {
    TreeNode searchedNode;
    pathList = new ArrayList<TreeNode>();
    for (TreeNode node : array) {
        pathList.add(node);
        nodeCounted(node, countNode);
        searchedNode = searchNode(rootNode, pathList);
        childNodes = searchedNode.getChildNodes();

        if (childNodes == null) {
            childNodes = new ArrayList<TreeNode>();
            childNodes.add(node);
            searchedNode.setChildNodes(childNodes);
            node.setParentNode(searchedNode);
            nodeAddToLinkedList(node, linkedNode);
        }
    }
}
```

图 4-2

```
    } else {
        isExist = false;
        for (TreeNode node2 : childNodes) {
            // 如果找到名称相同, 则更新支持度计数
            if (node.getName().equals(node2.getName())) {
                count = node2.getCount() + node.getCount();
                node2.setCount(count);
                // 标识已找到节点位置
                isExist = true;
                break;
            }
        }

        if (!isExist) {
            // 如果没有找到, 需添加子节点
            childNodes.add(node);
            node.setParentNode(searchedNode);
            nodeAddToLinkedList(node, linkedNode);
        }
    }
}
}
```

图 4-2 (续)

1. 图 4-2 为构建 FP-TREE 的实现代码。首先对于新输入的节点, 先确定其在已经存在的树中是否有对应的节点, 如果已经存在, 则在存在的节点后面继续构建新的节点, 如果不存在, 则在 root 的节点后面构建新的节点。
2. 在这个构建过程中, 主要涉及树的查找方法, 基本的有两种策略, 一种是深度优先, 另一种是广度优先, 这里采用深度优先的策略进行搜索。

实现互信息计算的代码如 4-3 所示。

1. 图 4-3 为互信息计算的实现代码。在得到候选待计算的词对之后, 我们通过获得这两个词的左右分布数据来计算候选词对的互信息。

```

//计算互信息
String wordArrayString = entry.getKey();
int wordArrayCount = entry.getValue().getWordCount();
String rightWordString = entry.getValue().getRightWord();
String leftWordString = entry.getValue().getLeftWord();
double leftWordCount = wordCountMap.get(leftWordString);
double rightWordCount = wordCountMap.get(rightWordString);
double MI =
(wordArrayCount/totalWordArrays)/((leftWordCount/totalWords)*(rightWordCo
unt/totalWords));

```

图 4-3

2. 这里的计算方式和之前介绍的章节是一样的，所以就不再赘述了。这里只是对计算的场景进行说明。我们这里计算的词分两种情况，一种是直接相连的词，那么这种情况和第2章定义的互信息计算方式一样，分别统计左词和右词的对方匹配情况就可以了。另外一种情况是非直接相连的词，而且是在同一个句式下面可以替换的词，这个时候它们共同出现的概率为共同出现在一个句子的概率，而不同出现的概率，则为出现在不同句子的概率。
3. 如果两个词不在候选词对中，那么这两个词的距离为无穷大，也即两个词的相似度为0。这是一种处理策略，可以根据自己的情况，选择不同的默认处理策略。

### 4.1.3 拓展

在上面的处理中，我们对于词性的使用并不是很多，比如在对两个直接相连的词进行计算的时候，我们并没有考虑词性的组合，这样得出来的结果会存在不符合语法的情况；所以在具体的应用中，可以根据相似词的词性之间的关系进行限制。对于算法本身的计算，也可以改用其他的距离计算函数，比如对于非直接相连的词语可以通过计算句子结构的余弦距离来进行改进。具体做法如下。

1. 首先将句子的词向量转换成结构向量。比如对于：[我/r, 今天/t, 去/vf, 打/v, 羽毛球/n, 了/ule, 。/w]、[今天/t, 我/r, 去/vf, 打/v, 篮球/n, 了/ule, 。/w]。我们最终得到“羽毛球”和“篮球”对应的句子结构向量： $\langle rr, t, vf, v, ule \rangle$ 和 $\langle rr, t, vf, v, ule \rangle$ ，然后计算这两个结构向量的余弦相似度。
2. 然后，将这两个词的句子结构余弦相似度的最大值作为这两个词的相似度。

这里介绍的方式也是基于统计的方式，或者准确地说是根据两个词共现的频率来作为两个词相似度（或者叫相近度）的度量，虽然能解决一部分情况，但是这对于词的语义距离反应就很不足。对于这种情况可以借助于 word2Vector 的方式来进行处理，实现词语间距离的更好处理。

## 4.2 N-Gram 算法实现词距离计算

4.1 节我们从词的搭配习惯角度介绍了计算词距离的一种方法，下面我们介绍另外一种基于词的搭配习惯来计算词距离的方法：N-Gram 算法。

### 4.2.1 场景

4.1 节我们介绍了一种基于词的共现频率和分布计算词距离的方法，但是对于如下的情况：

1. 蚂蚁搬家。
2. 我看到 5 只蚂蚁搬家。
3. 蚂蚁搬家的故事，我们都读过。

我们得到的会是：“蚂蚁”和“搬家”的词比较相近。而他们只是在搭配上比较相近，而在词义上却相差很多。虽然 4.1 节也提到了可以根据词性对这类的异常情况进行规避，比如限定搭配的词为同词性或者符合业务场景的词性组合。但是对于这种情况，有没有其他的解决办法呢？这里我们介绍一种基于词的向量的词距离计算方法：N-Gram 算法。

使用 N-Gram 计算词距离的方法主要有两种，一种是基于公共子串的方法，另一种是 embedding 的方法，我们这里介绍的是 embedding 的方法。

### 4.2.2 原理

N-Gram（有时也称为 N 元模型）是自然语言处理中一个非常重要的概念。因为对于字或者词的连接关系，我们有这样的一种认知（或者叫假设）：即后一个字/词的出现概率只与它前面的  $n$  个字/词有关系，而与其他元素没有关系。所以我们用一个 N 元窗口去逐个扫

描字/词的前后连接元素，那么就会得到一个字/词的连接向量。基于这个向量我们就可以对不同的字/词计算其距离。如果  $N=2$ ，则为 **bi-gram**，如果  $N=3$ ，则为 **tri-gram**。

图 4-4 为整个处理过程的一个示意图：

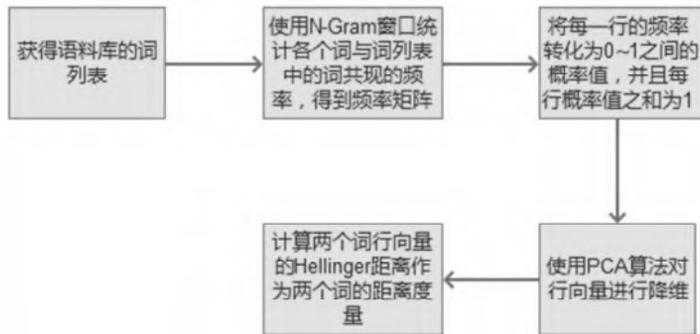


图 4-4

1. 第一步首先对语料库中的词进行统计，获得非重复的独立词向量，作为基础的词表。这个基础词表作为下一步统计共现词分布矩阵的列比较基础。
2. 对语料库中的文本进行逐句扫描，这里选用 **bi-gram** 窗口进行统计，获得各个词的共现分布矩阵。假设语料库中一共有  $N$  个独立的单词，那么共现矩阵的大小则为  $N*N$ 。对于本节开头举的例子，其共现矩阵如图 4-5 所示。

	蚂蚁	搬家	我	看到	……	读过
蚂蚁	0	3	0	0	……	0
……						

图 4-5

3. 得到上面的共现矩阵之后，对每一行的词频进行归一化处理。将每一行的数据除以该行数据之和。对于上面的矩阵经过变换之后，可得到如图 4-6 所示的矩阵。

	蚂蚁	搬家	我	看到	……	读过
蚂蚁	0	1.0	0	0	……	0
……						

图 4-6

4. 由实际情况可以知道，共现矩阵会是一个非常大的矩阵，并且是一个稀疏矩阵。所以每一个单词都对应一个非常大的稀疏向量。接下来就用 **PCA** 算法对于这样一个稀疏矩阵进行降维处理。因为我们最终希望得到一个稳定的表示矩阵，所以这里

使用 SVD 分解。对于降低的维数可以采用信息含量阈值限制（比如要包含 0.95 以上的信息量）也可以采用固定的维数限制。这里采用两种策略共用的方式，我们希望最终的向量维数在 50~150 之间（经验值，可以根据自己的情况进行修改）。所以如果在信息量阈值以上的维数满足上述要求则采用信息量阈值确定的维数，否则取固定的维度 100，如果总维数小于 100，则全部取，不需要做降维。

5. 得到了每一个词的向量表示之后，使用 Hellinger 距离作为两个词向量的距离度量。假设词  $A=(a_1, \dots, a_k)$ ，词  $C=(c_1, \dots, c_k)$ ，则 Hellinger 距离的计算公式如下：

$$H(A, C) = \frac{1}{\sqrt{2}} \sqrt{\sum_{n=1}^k (\sqrt{a_n} - \sqrt{c_n})^2}$$

### 4.2.3 实现

实现共现矩阵构建的代码如图 4-7 所示：

```

for (String sen : sentence) { //获得基础词表
    String[] words = sen.split(" ");
    for (int i = 0; i < words.length; i++) {
        //将每一个单词映射到一个向量数组以便后续构建矩阵的时候使用
        wordsMap[function.changeWord2Index(words[i])] = wordIndex;
        if (i > 0) {
            int pre = function.getWordOriginalIndex(words[i-1]);
            wordCoCurrency[wordsMap[pre]][wordIndex] += 1.0; //构建
共现矩阵
        }
        wordIndex++;
    }
}

```

图 4-7

1. 对已经分词的语料库进行逐句遍历，在遍历的过程首先对每一个词建立一个 map 映射，记录下当前词的行号，以备后续构建共现矩阵和获得词向量使用。如图 4-7 中的第 5 行代码所示。
2. 这里采用的是 bi-gram，所以对于每一句话前后连接的两个词都进行统计，将统计结果存放在一个数组中，以备后续使用。如图 4-7 中第 6~9 行代码所示。

实现 Hellinger 距离计算的代码如图 4-8 所示:

```
//对原始矩阵进行 PCA 降维
wordCoCurrency = PCA.pca(wordCoCurrency);
//计算两个词的距离
int i_A = function.getWordOriginalIndex(A);
int i_C = function.getWordOriginalIndex(C);
double[] a_array = wordCoCurrency[wordsMap[i_A]];
double[] c_array = wordCoCurrency[wordsMap[i_C]];
double sum = 0.0;
for (int i = 0; i < c_array.length; i++) {
    sum += Math.pow((Math.sqrt(a_array[i]) -
Math.sqrt(c_array[i])), 2.0);
}
//得到最终两个词的 Hellinger 距离
sum = Math.sqrt(sum)*(1.0/Math.sqrt(2.0)); }
```

图 4-8

1. 首先对共现矩阵进行降维处理，采用的是 PCA 方法。
2. 接着对待处理的两个词，分别获得其对应的降维后的向量。然后运用 Hellinger 距离计算公式计算两个向量的距离。得到两个词的 Hellinger 距离。这个距离越小，则表示两个词越相似。

## 4.2.4 拓展

embedding 是文本处理中的一个非常重要的步骤，随着深度学习不断深入的应用，这项工作就显得越来越重要。并且很多研究学者的工作显示，词向量能很好地反映词的语义属性，比如词向量之间可以进行加、减操作，并且操作后的向量能得到对应语义的向量。经常用的例子就是：国王-男人+女人=皇后。在国外的一个购物网站上，运用词向量进行服装推荐，用户选择一款衣服，再输入一个单词就可以推荐出对应语义的服装。比如对于一个条纹 T 恤+pregnant，则推荐对应的孕妇装款式，示意图如 4-9 所示。



备注：图片来自于网络

图 4-9

当然对于词的向量化也有很多的方法，谷歌提供了开源的 `Word2vector` 工具。在中文上的应用效果也不错。本节介绍的这种方法是一种比较基础的方法。其中对于稀疏矩阵的处理就是按照最基本的方式。可以按照稀疏矩阵的存储和矩阵分解的方式进行处理，这样就可以节省大量的存储空间，同时提升处理速度。对于词的选择也可以进行处理，比如去除停用词和其他有特殊规则可以过滤的词。

## 4.3 BP 算法实现词距离计算

对于词距离的计算，越来越多的研究和方法都在往 `embedding` 的方向发展，所以在这里我们也介绍一种实现词的 `embedding` 的方法：`skip-gram` 算法。

### 4.3.1 场景

在 4.1 节介绍使用 `FP-Growth` 算法计算词距离的时候，我们提出了要处理两种情况，一种是直接连接的词，另一种是非直接连接的词。在 4.2 节介绍 `embedding` 算法的时候，我们发现其处理的基本只有直接连接的情况。那对于非直接连接的情况，有没有 `embedding`

的处理方法呢？答案是有的。我们这里介绍一种处理这种情况的 embedding 算法：skip-gram 算法。

### 4.3.2 原理

计算流程图和流程示意图如图 4-10 所示。

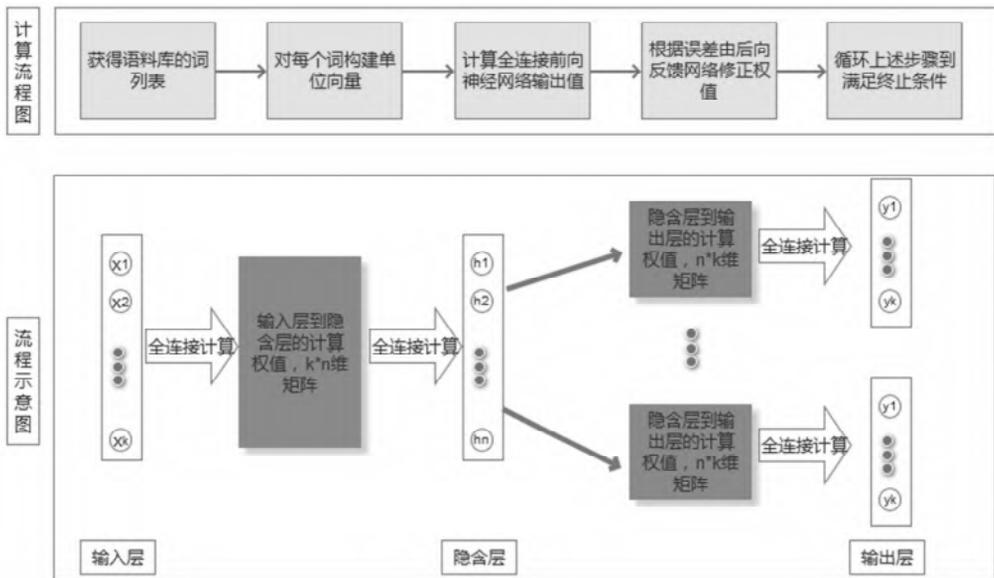


图 4-10

1. 首先对语料进行分词统计，统计语料的独立词个数。注意这里对语料进行了处理，去除了其中的停用词。如果在所处理的场景中，有其他规则也可以加进来去除不必要的单词数量，以减少最终的计算矩阵向量长度。
2. 构建词向量，这里构建的是单位词向量，即当前词所在的单元格元素为 1，其他单元格元素为 0，比如 4.2 节中的例子，所得的词向量如表 4-1 所示：

表 4-1

	蚂蚁	搬家	看到	.....	读过
蚂蚁	1.0	0.0	0	.....	0
搬家	0.0	1.0	0.0	.....	0.0
.....					

3. 接下来就是按照前向网络的路径计算各个层的值，最终得到输出值。并与目标值计算其误差，因为输出的结果为多个，所以需要将多个目标值的误差值加和，然后将加和的误差值作为后向反馈网络中各个权值校正的误差输入。这里要保证所有的输出权值矩阵作为计算的过程是稳定的，并且是相同的。
4. 对隐含层到输出层的权值矩阵  $w_o$  和输入层到隐含层权值矩阵  $w_i$  进行逐层调整。最终调整的终止条件是，前后两次误差值变化小于设定的阈值，或者达到最大的循环上限。
5. 得到稳定的模型之后，对于任意输入的词，都可以得到一个对应的向量，两个词之间的距离也就使用 **Hellinger** 距离来计算。
6. 对于目标值的选择，这里采用前后直接连接词的单位向量。当然这里也可以采用有监督的方案，即人为地给出目标向量，然后进行调整运算。这样就变成了有监督的学习算法。

### 4.3.3 实现

实现前向误差计算的代码如图 4-2 所示：

```
//初始化权重向量值
weightLayer0 = initialWeightMatrix(xDemention, hiddenDemention);
Arrays.fill(hiddenLayerInitial, 0.0);
weightLayer1 = initialWeightMatrix(hiddenDemention, yDemention);
//计算前向反馈网络的误差
overallError = propagateNetWork(x, y, hiddenLayerInput,
    outputLayerDelta, overallError);
/*因为这里采用的是随机梯度下降，所以每一次迭代的结果并不一定比上次好
 * 所以要对迭代结果进行筛选，只保留最优的迭代结果
 * */
if (overallError < minError) {
    minError = overallError;
}else {
    continue;
}
```

图 4-2

1. 在计算前向网络时，首先要对两个权值矩阵进行赋值。这里选择赋值 0~1 之间的

随机值。对于隐含向量的值则赋值为 0。

2. 然后对前向网络进行全连接计算。这里在计算的时候，不涉及激活函数，仅仅是做向量之间的内积计算。
3. 对于每一步的计算，我们都会判断当前所计算的误差是不是小于之前的最优误差，如果小于之前的最优误差，则对最优误差进行更新，同时反向更新权值，否则继续进行循环计算，不更新权值。如图 4-2 中第 6~10 代码行所示。

实现后向权值调整计算的代码如 4-3 所示。

```
first2HiddenLayer =
Arrays.copyOf(hiddenLayerInput.get(hiddenLayerInput.size()-1),
hiddenLayerInput.get(hiddenLayerInput.size()-1).length);
double[] hidden2InputDelta = new double[weightLayerh_update.length];
hidden2InputDelta = backwardNetWork(x, hiddenLayerInput,
outputLayerDelta, hidden2InputDelta, weightLayer0_update,
weightLayerl1_update, weightLayerh_update);
weightLayer0 = matrixAdd(weightLayer0, matrixPlus(weightLayer0_update,
alpha));
weightLayerl1 = matrixAdd(weightLayerl1, matrixPlus(weightLayerl1_update,
alpha));
```

图 4-3

1. 首先对隐含层的输入值进行调整，获得隐含层最新的输入值。即存储隐含层输入值 list 的最后一个元素，如图 4-3 中第 11~12 行代码所示。
2. 然后进行后向反馈网络的权值更新，本步骤也不涉及激活函数的计算，仅仅是向量的内积计算。将计算得到的更新权值存在 update 数组中。
3. 然后分别更新输入层到隐含层的计算权值矩阵及隐含层到输出层的计算权值。更新的方式即为矩阵对应位置的元素加和即可。

### 4.3.3 拓展

不同属性的词有直接连接的处理情形，如因为经常搭配而出现词距离比较近的情况；而对于非直接连接的词语，也会出现同一句式下相反的词距离比较近的情况。比如：“这个人很好”“这个人很坏”。本书 4.1 节、4.2 节介绍的方法都是无监督的，好处就是对于样本

数据的要求比较低，但是随之而来的是对效果的评估比较难。本节介绍的方法既可以转换成有监督的算法也可以转换成无监督的算法。有监督的算法的好处就是对于结果评估更容易，但是缺点就是获得学习样本的成本太高。

借助于新近的对抗神经网络（Generative Adversarial Network, GAN）和微软提出的对偶学习（Dual Learning）理论，可以大大降低学习样本的需求量，同时提高学习精度。GAN 网络是将整个过程分成两步，第一步是生成数据——即由生成器生成用于分类的数据、第二步是识别数据——即由分类器对于混合了生成数据和真实数据的待分类样本进行分类。最终的目标是让分类器无法识别由生成器生成的数据。对偶学习是对于当前的任务构建一个它的对偶任务。比如上文的 `skip-gram`，我们给定一个输入值，进而计算它的目标值，那么其对偶问题为：将目标值设定为输入，而将输入值设定为输出值。这样就可以将学习样本减半，而对整个网络进行同样程度的学习。

## 第 5 章

# 拼音汉字混合识别模型

在词距离的计算中，我们介绍使用 embedding 的方式进行词的向量化，然后通过计算向量之间的距离来实现对词距离的计算。其中实现词的向量化，对于文本处理具有非常大的作用，因为一旦实现了词的向量化，我们就可以利用深度学习的方式对文本进行处理。

既然有了强大的工具，我们对于文本的处理就会有更多的希望，但是这里面也会因为基础数据的异常导致处理起来比较困难，比如文本中夹杂着拼音，而本章就是介绍一些实现拼音汉字混合识别的模型。

## 5.1 贝叶斯模型实现拼音汉字混合识别

这里仍然采用朴素贝叶斯方法，对于这个方法我们在前面已经有所了解，本节就介绍其在混合输入识别中的一个应用场景。

### 5.1.1 场景

在日常使用拼音输入的时候，我们会因为输入过快而出现拼音和汉字混合输入的情况。比如我原本想输入“英雄救美”，但是却输成了“英雄 jium”。对于这种情况，首先就需要把其中的拼音转换成汉字，才能进行后续的挖掘和识别操作。针对这个问题，这里介绍一种基本的方法：贝叶斯模型，来实现对拼音汉字混合体的识别。

对于这种情况，我们要做的一件事就是把句子中的拼音转换成汉字。并且转换的上下文条件已经给定。所以很自然地我们会联想到条件概率，所以这个转换其实就是求所要转换拼音对应的汉字集合中，哪个字在当前上下文环境中拥有最大的条件概率。

### 5.1.2 原理

整个计算流程示意图如图 5-1 所示。

1. 首先对于待识别语句提取其中的拼音信息。并根据声母、韵母组合的情况，将不同的拼音进行分割。由于我们对于声母、韵母的识别使用的是字典的形式，所以我们将已知的声母、韵母列表整理好，得到输入之后，直接根据字典查询进行分割。也可以理解为做了一个拼音的“分词”，得到待识别的拼音列表。
2. 在获得拼音的列表之后，就对拼音对应的汉字进行查找，获得当前拼音所对应的汉字列表，作为候选集合。这里的处理方式也可以对调过来：先由当前上下文确定可能的字列表，再用这些字逐个和当前的拼音进行匹配，取概率最大的字作为结果。但是因为字与字之间的组合情况远远大于字的个数（常用的汉字大概在 3000 个左右），所以需要考虑遍历的数量，这里采用的是先确定候选字，再计算各个字的条件概率。

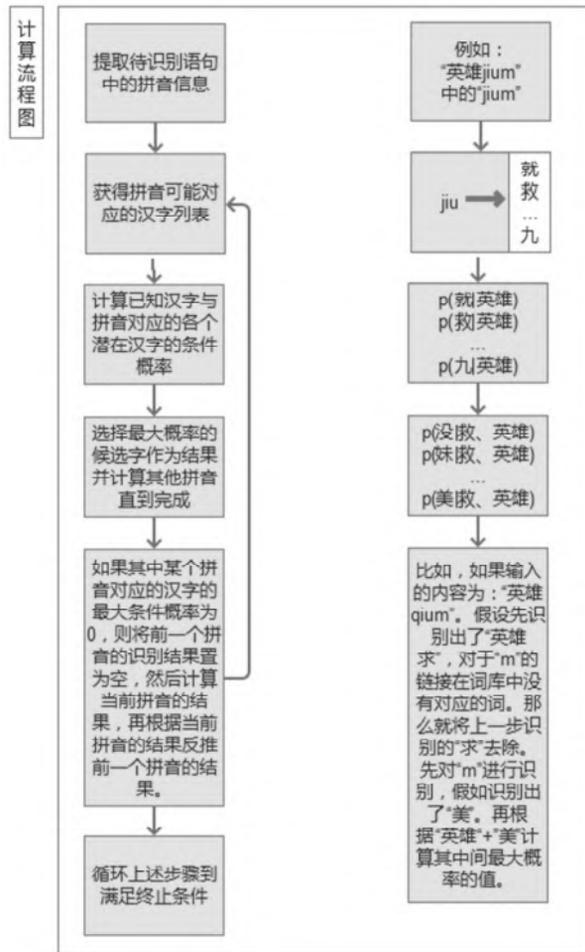


图 5-1

- 根据上下文环境，计算各个候选字的条件概率，然后选择其中概率最大的值作为结果。之前已经介绍过贝叶斯方法的原理了，这里就只把贝叶斯公式简单列出来做一个回顾：

$$p(w_i|w_{i-1}) = p(w_i|w_{i-1}) \cdot p(w_{i-1})/p(w_i)$$

- 对于待识别的拼音列表中的元素进行逐个识别，如果计算过程中没有出现找不到（即最大概率为 0）的情况，则结束整个计算过程，否则则按照调整策略，逐个往前进行调整，直到每个拼音都能找到对应的可能汉字为止。

### 5.1.3 实现

实现拼音对应的候选字计算的代码如图 5-2 所示：

```

for (String sentence : sentenceList) {
    List<PinyinItem> pinyinItems = PinyinParser.parse(sentence);
    int i = 0;
    for (PinyinItem pinyinItem : pinyinItems) {
        //如果当前的元素是拼音字母，则进行识别，否则不识别
        if (pinyinItem.getType() == 0) {
            String pinyin = pinyinItem.getPinyinString();
            List<String> candidateWords = PinyinParser.getWordList(pinyin);
            //获得当前拼音前一个汉字在分词后的列表中的位置
            int prePos = pinyinItem.getPrePos();
            //获得当前拼音紧相连的后一个汉字在分词后的列表中的位置
            int proPos = pinyinItem.getProPos();
            if (prePos < 0) { //如果当前拼音是第一个字
                if (proPos >= pinyinItems.size()) { //如果当前拼音是最后一个字
                    pinyinWordMap.put(pinyin, ""); //当前拼音为独立拼音，则无法
                    识别
                    break;
                } else {
                    //根据当前拼音紧连的后一个中文和相间隔的距离，以及上下文的类型来计算最大概率的
                    候选字
                    String contextString =
                    pinyinItems.get(proPos).getWordString();
                    int distance = proPos - i;
                    int type = 1; //1 表示后缀，0 表示前缀
                    String candidateString = getWord(contextString,
                    candidateWords, distance, type);
                    pinyinWordMap.put(pinyin, candidateString); } } else {
                    String contextString =
                    pinyinItems.get(prePos).getWordString();
                    //当前拼音的前一个字的位为拼音，则找其对应识别的汉字
                    if (contextString == null) {
                        contextString =
                    pinyinWordMap.get(pinyinItems.get(prePos).getPinyinString());
                    //如果前一个拼音对应识别的汉字为空，也即当前是一个独立的拼音串，则不
                    进行识别
                    if (contextString == null || contextString.equals("")) {

```

图 5-2

```
        break;} }
        int distance = i-prePos;
        int type = 0;//1表示后缀,0表示前缀
        String candidateString = getWord(contextString, candidateWords,
distance,type);
        pinyinWordMap.put(pinyin, candidateString); }}
        i++;
    }
}
```

图 5-2 (续)

1. 首先对于输入的句子进行拼音分词, 获得拼音分词的列表。得到的分词结果是一个对象, 这个对象里面记录了: 拼音串、拼音的类型 (字母: 0; 汉字: 1)、拼音前面元素在分词后列表中的位置、拼音后面紧相连的一个汉字在分词后列表中的位置等信息。
2. 接着就遍历分词的列表, 对其中的字母进行转换, 获得其对应的最有可能的汉字。这里分两种情况, 如果待识别的字符串全部都是字母, 则不进行处理。这里只处理汉字和拼音混合出现的情况。如图 5-2 中第 6~34 行代码所示。
3. 对于条件概率的计算, 这里采用 **bi-gram** 的窗口, 仅对相连的两个字符进行计算。当然可以采用 **tri-gram** 或者其他长度的窗口。对于条件概率的条件也可以添加, 比如可以增加词长、拼音在字符中的位置等。

### 5.1.4 拓展

对于最优字符的选择, 我们这里没有考虑其路径的关系, 比如对于“英雄 **jiu**”这个例子, 我们如果仅仅依赖于“雄”来推断“**jiu**”。那么很有可能推断出“雄安”的引申词“雄救”(假如存在这样的词的话)。对于这个问题就是我们 5-2 节介绍的方法要解决的问题。

对于这个问题, 我们已经知道它的本质其实是把拼音转换为汉字, 所以也可以用分类的思路来解决, 只是这个分类是一个多分类, 所以利用类似于 **softmax** 的多分类器, 计算各个类别的概率, 然后选择最大的概率作为候选概率。而这类方法的一个代表就是我们将在 5.3 节要介绍的一种方法。

## 5.2 HMM 模型实现拼音汉字混合识别

HMM（隐马尔科夫模型）因为其在序列数据处理中的良好表现，被广泛应用在自然语言处理的各个方面。这里我们就其在混合输入识别中的应用进行说明和介绍。

### 5.2.1 场景

在 5.1 节，我们讨论一种场景，就是希望在对拼音进行汉字转换的时候，能考虑前后路径的影响，而不仅仅是根据固定窗口的共现概率来进行估计。本节介绍一种能够实现这种场景处理的算法：HMM（隐马尔科夫）模型。

HMM 由两个部分组成：马尔可夫链和一般随机过程。其中马尔可夫链用来描述状态的转移，在模型中用转移概率矩阵来表示。而这个链也就是我们上面说的路径，如图 5-3 所示表示了一个“英雄救美”的路径示意图，黑色线所代表的即为最大可能路径。一般随机过程用来描述状态与观察序列之间的关系，如果这个状态是未知的，那么也就是隐含状态，在模型中用隐含状态到观察值的转换概率来表示。

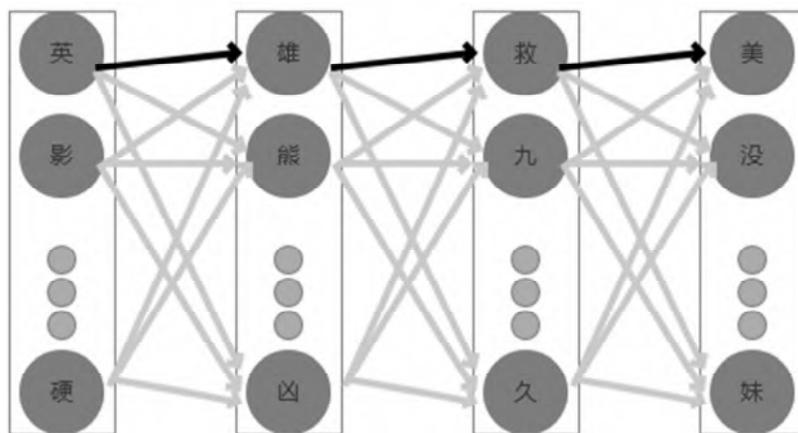


图 5-3

### 5.2.2 原理

使用 HMM 模型进行拼音转汉字的模型流程如图 5-4 所示。

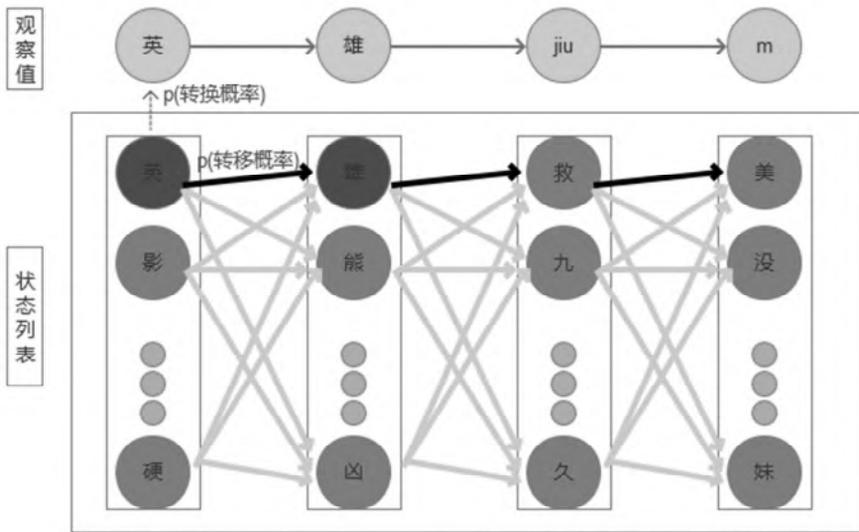


图 5-4

1. 如图 5-4 结构所示，对于 HMM 模型，有两个要素要先确定，一个状态值列表，另一个观察序列。对于我们的情况，状态就是所有汉字的列表，假设我们以常用汉字（3000）为猜测对象，那么就相当于我们的状态值一共有 3000 个。而观察序列，也就是我们得到的输入序列“英雄 jium”。
2. 确定状态值和观察序列之后，就要计算两个概率：状态转移概率和状态到观察值的转换概率。转移概率可以根据语料库中 bi-gram 的频率统计进行计算。转换概率我们这里可以假定一个拼音下面的汉字是等概率转换的，那么每个汉字（状态）到对应拼音（观察值）的转换概率就是： $1.0/N$ 。 $N$  为该拼音对应的汉字个数总和。
3. 有了状态值、观察序列、转移概率矩阵、转换概率矩阵。接下来就要求解黑色连线所表示的最大概率路径了。如果我们把所有的连接路径视为一个图，其中各个汉字（状态）为图的节点，各个汉字之间的转移概率为边，那么最大概率路径的求解也就相当于求解图中连接特定点的最长（最短）路径。下面就介绍一种求解该路径的动态规划算法：维特比算法。

### 维特比算法：

1. 对于  $t$  时刻各个状态的概率计算公式如下：

$$P_t(S_i) = \max_{1 \leq j \leq N} P_{t-1}(S_j) \cdot P(t_{j,i}) \cdot P(T_{i,o_t})$$

其中,

$P_t(S_i)$ : 表示  $t$  时刻状态  $i$  的概率值。

$P(t_{j,i})$ : 表示状态  $j$  到状态  $i$  的转移概率。

$P(T_{i,o_t})$ : 表示状态  $i$  到  $t$  时刻的观测值的转换概率。

也就是  $t$  时刻状态  $i$  的概率等于  $t-1$  时刻各个状态转移到  $i$  状态后, 拥有的最大转移概率。

而  $t$  时刻的最终状态为拥有最大概率的状态值,  $S_i' = \max_{1 \leq i \leq N} P_t(S_i)$ 。

2. 对于本文的示例图, 如图 5-5 所示。

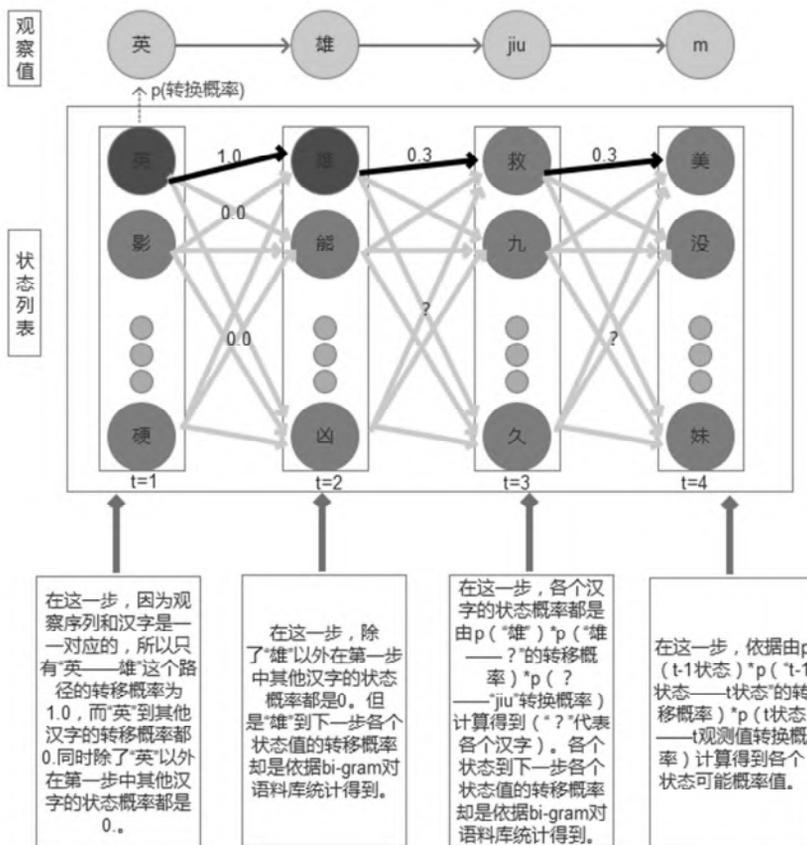


图 5-5

3. 首前两步我们不需要计算，因为这个序列图 5-5 是确定的。所需要计算的只有后面两步。根据规则，我们对于  $t=3$  时刻的计算过程即为如下步骤所示：
  - a)  $t=2$  步向  $t=3$  步传递的状态只有一个，即“雄”。我们把它的概率置为 1（即  $P_{t-1}(S_j)$ ）。
  - b) 下面以  $t=3$  时刻的状态“救”为例，说明它的概率计算过程。
    - i. 首先假设“雄——救”（即  $P(t_{j,i})$ ）的转移概率为 0.3，“救——jiu”（即  $P(T_{i,o_t})$ ）的概率为 0.01。
    - ii.  $P(\text{救}) = P_{t-1}(\text{雄}) \cdot P(t_{\text{雄,救}}) \cdot P(T_{\text{救,jiu}}) = 1.0 \cdot 0.3 \cdot 0.01 = 0.003$ 。
  - c) 同理可以求出其他字的概率，然后选择其中最大的作为  $t=3$  时刻的可能汉字。
4. 再按照同样的方法计算  $t=4$  可能的汉字。这样就完成了整个链路的计算。

### 5.2.3 实现

实现 HMM 计算的代码如图 5-6 所示。

```

for (String sentence : sentenceList) {
    List<PinyinItem> pinyinItems = PinyinParser.parseHMM(sentence);
    for (PinyinItem pinyinItem : pinyinItems) {
        //如果当前的元素是拼音字母，则进行识别，否则不识别
        if (pinyinItem.getType() == 0) {
            String pinyin = pinyinItem.getPinyinString();
            List<String> candidateWords =
PinyinParser.getWordList(pinyin);
            int N = candidateWords.size();
            int type = pinyinItem.getType();
            double max_s = Double.MIN_VALUE;
            int s_index = -1;
            double[] tmp_S = {};
            if (type == 0) { //当前循环为汉字在前，即“英雄 jium”的格式
                getCandidateWords(preTransfer, S, wordStrings,
                    pinyinWordMap, pinyin, N, max_s,
s_index, tmp_S);
            } else { //当前循环为汉字在前，即“ni 妹”的格式

```

图 5-6

```

        getCandidateWords(proTransfer, S, wordStrings,
                           pinyinWordMap, pinyin, N, max_s,
s_index,tmp_S);
    }
    S = tmp_S;
}
}
}
}
}

```

图 5-6 (续)

1. 首先对于待转换的语句进行拼音分词，这里的分词需要注意的是，我们只保留待处理的拼音，以及与其紧相连的前（后）缀词。
2. 根据汉字出现的位置不同，选择不同的转移概率矩阵进行迭代计算，计算过程都一样，所以就封装了一个通用函数，具体函数实现见维特比算法实现代码。

实现维特比计算的代码如图 5-7 所示。

```

private static void getCandidateWords(double[][] preTransfer,
double[] S,
String[] wordStrings, Map<String, String> pinyinWordMap,
String pinyin, int N, double max_s, int s_index,double[] S_) {
for (int j = 0; j < preTransfer.length; j++) {
double max = Double.MIN_VALUE;
for (int k = 0; k < S.length; k++) {
double tmp = S[k]*preTransfer[k][j]*1.0/N;
if (tmp > max) { //获得状态 j 在 t 时间的最大值
max = tmp;
}
}
S_[j] = max; //更新当前时刻状态 j 的值
if (max > max_s) {
max_s = max;
s_index = j;
}
}
if (s_index != -1) {
pinyinWordMap.put(pinyin, wordStrings[s_index]);
}else {

```

图 5-7

```
        pinyinWordMap.put(pinyin, "");  
    }  
}
```

图 5-7 (续)

1. 首先对前一时刻的状态值进行遍历，分别与  $t$  时刻各个状态进行组合计算，获得  $t$  时刻各个状态的值。如图 5-7 中第 4~11 行代码所示。
2. 然后取  $t$  时刻各个状态的最大值对应的作为最终的输出状态。如图 5-7 中第 13~16 行代码所示。

### 5.2.3 拓展

由上述过程我们可以知道，即使对于全部是拼音的输入，HMM 模型也是可以进行计算的，只是我们这里介绍的场景不对这种情况进行处理，但是原理和实现基本都一致。

HMM 模型有一个前提假设： $t$  时刻的变量值只与  $t-1$  时刻的变量值有关。基于这个假设，我们就可以省去很多的计算和处理环节，在语音识别等实际场景中，HMM 模型也有很好的表现。但是这个假设前提在比较小的数据集上是合适的，在大量真实语料中观察序列更多的是以一种多重的交互特征形式表现的，即观察元素之间广泛存在的长距离相关性。在命名实体识别的任务中，由于实体本身结构所具有的复杂性，利用简单的特征函数往往无法涵盖所有的特性，这时 HMM 的假设前提使得它无法使用复杂特征。所以在 HMM 的基础上，有些学者提出了 MEMM（最大熵隐马尔科夫模型）来进行改进，效果也有一定的提升。

## 5.3 RNN 神经网络模型实现拼音汉字混合识别

本节要介绍的 RNN（自回归神经网络）模型，在对序列数据的处理中，也表现得非常抢眼，这主要是由其网络结构所决定的，同时它又继承神经网络的一些优点，所以在自然语言处理领域，越来越受到重视。下面就对其在混合输入中的应用做一个说明和介绍。

### 5.3.1 场景

在 5.1 节中，我们曾说过，将拼音转换为汉字的场景，其实也可以定义为一个分类问题，拿“英雄 jium”的例子中的“jiu”来说就是，我们有如下特征，现在需要将它分类后一个候选的汉字类。

1. 紧相连的前缀字为“雄”。
2. 紧相连的单词为“英雄”。
3. 该词组的长度为 4。
4. 该字处在第 3 个位置。
5. 该字后面一个字的首字母为 m。

在这个场景下，我们输入的数据是一个序列，也就是数据前后之间有较强的关联关系，所以我们需要采用一种能够处理序列问题的算法。这里介绍这类算法中比较基础的一种：Simple RNN 模型。

Simple RNN 是 RNNs 的一个特例，它是一个三层网络，与标准的多层感知机模型相比较，Simple RNN 在隐藏层增加了自身和自身的连接，其连接示意图如图 5-8 所示。

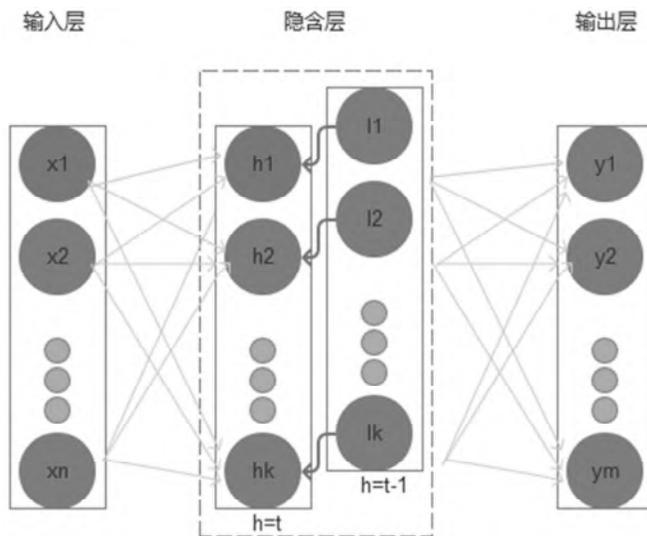


图 5-8

1. 如图 5-8 所示, Simple RNN 在隐含层多了一个操作, 即将  $t$  时刻的隐含层变量和  $t-1$  时刻的隐含层变量进行连接。图中的  $h_t$  代表的是隐含层的计算变量,  $l_t$  是上一时刻隐含层的计算结果。其中  $t-1$  时刻节点和  $t$  时刻节点的连接是固定的, 权值也是固定的, 并且两个向量的长度是一样的, 在位置上也是一一对应的。
2. Simple RNN 在每一步中, 使用标准的前向反馈网络进行传播, 然后使用学习算法进行学习。由于  $t-1$  时刻隐含层计算结果的引入, 即隐藏层的输入是由输入层的输出与上一步的自己的状态所决定的, 使得 RNN 网络能够保存之前的计算结果, 也就是记录了之前序列的信息。因此 Simple RNN 能够解决标准的多层感知机无法解决的对序列数据进行预测的任务。

### 5.3.2 原理

整个计算流程如图 5-9 所示。

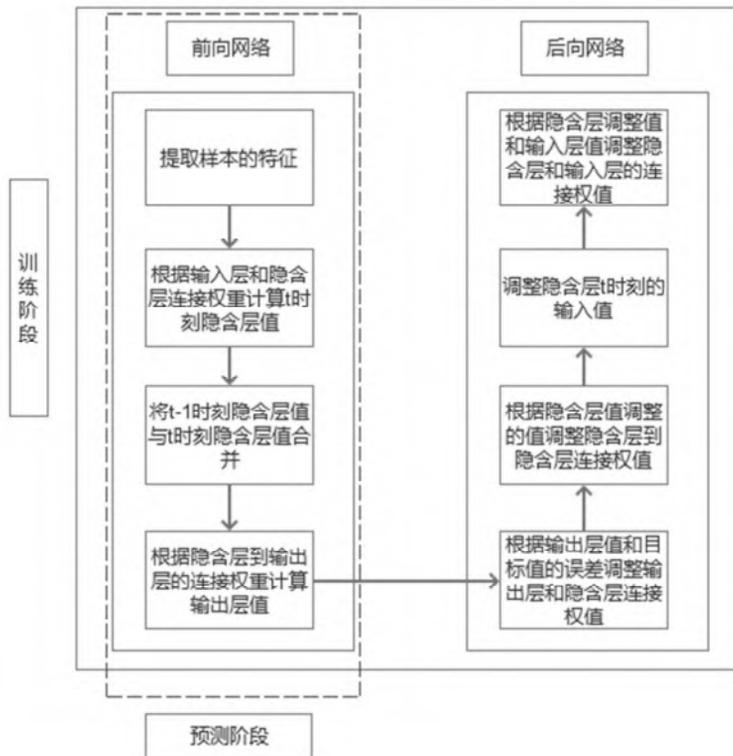


图 5-9

1. 如图 5-9 所示，在训练阶段分为两个阶段，一个阶段是前向反馈网络，另一个阶段是后向反馈网络。前向网络的作用是由输入值根据三层网络的权值计算其输出值。后向网络的作用是根据预测值和目标值之间的误差，逐步调整三个连接权重矩阵和隐含层的输入值。
2. 而预测阶段只是运用前向网络计算其预测值。这个过程中运用的权重是经过训练后更新的权值矩阵。
3. 这里采用的是在每一层的计算完成以后，都要做一个映射，采用的映射的激活函数是 sigmoid 函数。但是这个函数在经过多层调整之后，会存在对于调整梯度为 0 的情况，也即对于整个权值的调整不再起作用。如果在实际的使用中出现了这种情况，则要根据自己的情况调整激活函数的形式，比如可以采用 tanh 函数。
4. 整个计算过程的连接基本都采用全连接，这个在实际情况中也可以根据自己的需求进行调整，当然这个过程会非常的麻烦，并且在通常情况下我们的隐含层向量都会大于输入层向量，以起到对特征进行扩展的目的。所以对于特征的连接权重可以交由优化函数完成即可，我们只需要给出足够多的训练样本即可。
5. 从理论上来说 RNN 能记录无限长的因素，但是如果周期过长会导致其无法计算，所以要根据实际情况对循环网络的长度进行设定，以实现效率和效果的平衡。

### 5.3.3 实现

实现特征训练计算的代码如图 5-10 所示：

```
public double train(List<double[]> x, List<double[]> y) {
    alreadyTrain = true;
    double minError = Double.MAX_VALUE;
    for (int i = 0; i < totalTrain; i++) {
        //定义更新数组
        double[][] weightLayer0_update = new
double[weightLayer0.length][weightLayer0[0].length];
        double[][] weightLayer1_update = new
double[weightLayer1.length][weightLayer1[0].length];
        double[][] weightLayerh_update = new
double[weightLayerh.length][weightLayerh[0].length];
```

图 5-10

```
List<double[]> hiddenLayerInput = new ArrayList<double[]>();
List<double[]> outputLayerDelta = new ArrayList<double[]>();
double[] hiddenLayerInitial = new double[hiddenLayers];
//对于初始的隐含层变量赋值为0
Arrays.fill(hiddenLayerInitial, 0.0);
hiddenLayerInput.add(hiddenLayerInitial);
double overallError = 0.0;
//前向网络计算预测误差
overallError = propagateNetWork(x, y, hiddenLayerInput,
    outputLayerDelta, overallError);
if (overallError < minError) {
    minError = overallError;
}else {
    continue;
}
    first2HiddenLayer =
Arrays.copyOf(hiddenLayerInput.get(hiddenLayerInput.size()-1),
hiddenLayerInput.get(hiddenLayerInput.size()-1).length);
    double[] hidden2InputDelta = new
double[weightLayerh_update.length];
    //后向网络调整权值矩阵
    hidden2InputDelta = backwardNetWork(x, hiddenLayerInput,
        outputLayerDelta,
hidden2InputDelta,weightLayer0_update, weightLayer1_update,
weightLayerh_update);
        weightLayer0 = matrixAdd(weightLayer0,
matrixPlus(weightLayer0_update, alpha));
        weightLayer1 = matrixAdd(weightLayer1,
matrixPlus(weightLayer1_update, alpha));
        weightLayerh = matrixAdd(weightLayerh,
matrixPlus(weightLayerh_update, alpha));
    }
    return -1.0;
}
```

图 5-10 (续)

1. 首先对待调整的变量进行初始化，在完成这一步之后，就开始运用前向网络对输入值进行预测，完成预测之后，则运用后向网络根据预测误差对各个权值向量进行调整。

2. 这个过程需要注意的是每次权值的更新不是全量更新，而是根据学习速率  $\alpha$  来进行更新的。其他的步骤基本和之前描述的一样。

实现预测计算的代码如下：

```
public double[] predict(double[] x) {
    if (!alreadyTrain) {
        new IllegalAccessException("model has not been trained, so can not
to be predicted!!!");
    }
    double[] x2FirstLayer = matrixDot(x, weightLayer0);
    double[] firstLayer2Hidden = matrixDot(first2HiddenLayer,
weightLayerh);
    if (x2FirstLayer.length != firstLayer2Hidden.length) {
        new IllegalArgumentException("the x2FirstLayer length is not
equal with firstLayer2Hidden length!");
    }
    for (int i = 0; i < x2FirstLayer.length; i++) {
        firstLayer2Hidden[i] += x2FirstLayer[i];
    }
    firstLayer2Hidden = sigmoid(firstLayer2Hidden);
    double[] hiddenLayer2Out = matrixDot(firstLayer2Hidden,
weightLayer1);
    hiddenLayer2Out = sigmoid(hiddenLayer2Out);
    return hiddenLayer2Out;
}
```

图 5-11

1. 在预测之前首先要确定模型是不是已经完成训练，如果没有完成训练，则要先进行训练得到预测模型。当然这个是一个跟具体业务逻辑有关的校验，这主要是针对预测和训练分开的情况，如果训练和预测是在一个流程内的，那么可以不用校验。
2. 在得到训练模型之后，就根据前向网络的流程逐步计算，最终得到预测值。因为我们这里是一个分类问题，所以最终是选择具有最大概率的字作为最终的输出。

### 5.3.4 拓展

本节介绍的是一种基本的 RNN 模型，它的好处就是容易实现，当然缺点就是对于模型处理得过于简单，对于元素和元素之间的影响因子记录的信息仍然比较少。

LSTM 是 RNN 的一种，是一种能够实现长短记忆的非线性自回归网络。这种网络不仅能够记录长期的影响，还能记录短期的影响，也就是对不同周期的影响因素赋予不同的权重，以决定不同影响周期内不同因素的参与度。

对于 RNN 网络，训练样本也是一个痛点，模型本身需要输入大量的训练样本。而训练样本的获得又是一个高成本的问题。所以针对这类网络，可以结合 GAN 和 Dual Learning 的理论框架进行扩展，一方面提升模型的准确度，另一方面降低训练样本的获得成本。

## 第 6 章

# 文本自动生成模型

在自然语言处理中，另外一个重要的应用领域，就是文本的自动撰写。关键词、关键短语、自动摘要提取都属于这个领域中的一种应用。不过这些应用，都是由多到少的生成。这里我们介绍其另外一种应用：由少到多的生成，包括句子的复写，由关键词、主题生成文章或者段落等。

## 6.1 基于关键词的文本自动生成模型

本章第一节就介绍基于关键词生成一段文本的一些处理技术。其主要是应用关键词提取、同义词识别等技术来实现的。下面就对实现过程进行说明和介绍。

### 6.1.1 场景

在进行搜索引擎广告投放的时候，我们需要给广告撰写一句话描述。一般情况下模型的输入就是一些关键词。比如我们要投放的广告为鲜花广告，假设广告的关键词为：“鲜花”、“便宜”。对于这个输入我们希望产生一定数量的候选一句话广告描述。

对于这种场景，也可能输入的是一句话，比如之前人工撰写了一个例子：“这个周末，小白鲜花只要 99 元，并且还包邮哦，还包邮哦！”。需要根据这句话复写出一定数量在表达上不同，但是意思相近的语句。这里我们就介绍一种基于关键词的文本（一句话）自动生成模型。

### 6.1.2 原理

模型处理流程如图 6-1 所示。

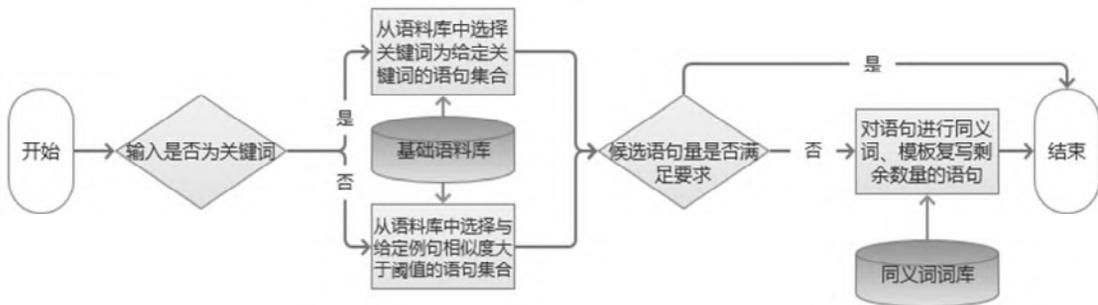


图 6-1

1. 首先根据输入的数据类型不同，进行不同的处理。如果输入的是关键词，则在语料库中选择和输入关键词相同的语句。如果输入的是一个句子，那么就在语料库中选

择和输入语句相似度大于指定阈值的句子。

- 对于语料库的中句子的关键词提取的算法，则使用之前章节介绍的方法进行。对于具体的算法选择可以根据自己的语料库的形式自由选择。
- 语句相似度计算，这里按照图 6-2 左边虚线框中的流程进行计算：

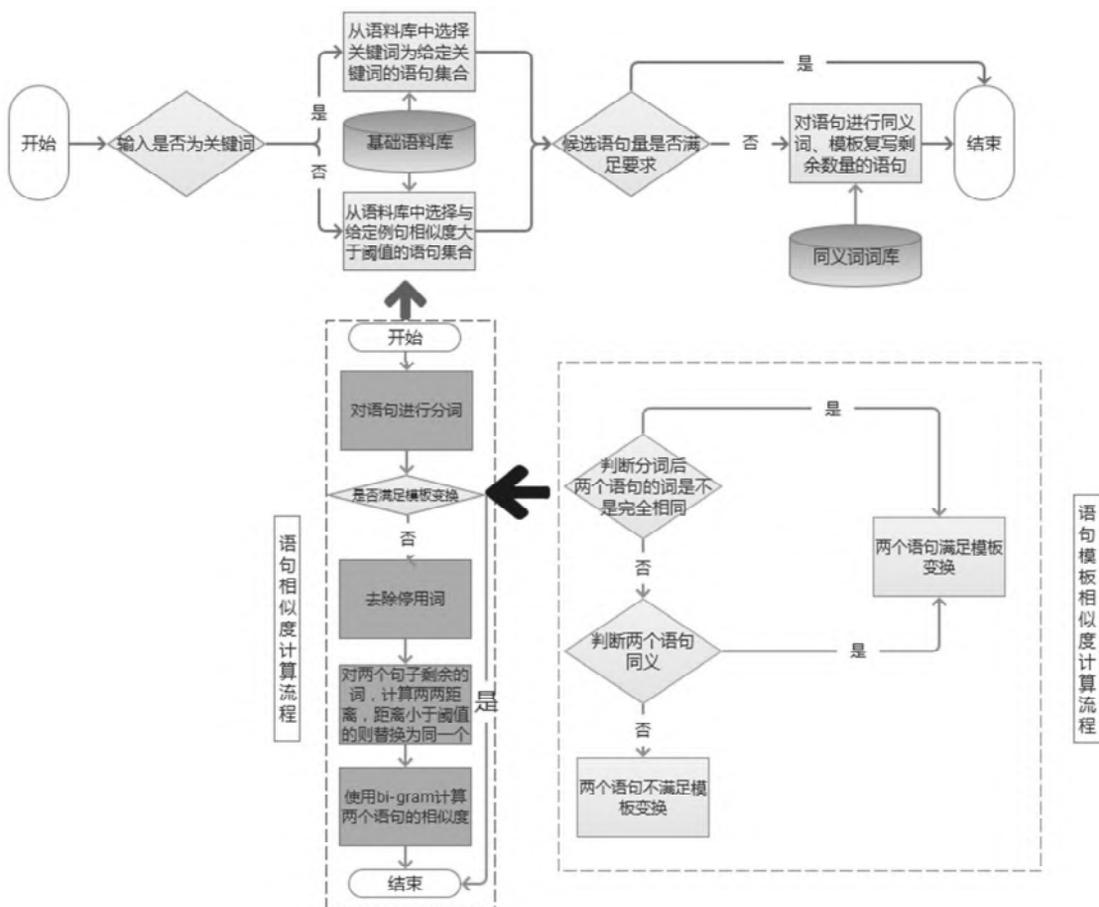


图 6-2

- 首先对待计算的两个语句进行分词处理，对于分词后的语句判断其是否满足模板变换，如果满足则直接将语句放入候选集，并且设置相似度为 0。如果不满足则进入到 c) 步进行计算。
- 判断两个语句是否满足模板变换的流程图，如图 6-2 中右边虚线框所标记的流

程所示：

- i. 首先判断分词后，两个句子的词是不是完全一样，而只是位置不同，如果是则满足模板变换的条件。
  - ii. 如果词不完全相同，就看看对不同的词之间是否可以同义词变换，如果能够进行同义词变换，并且变换后的语句两个句子去公共词的集合，该集合若为某一句话的全部词集合，则也满足模板变换条件。
  - iii. 如果上述两个步骤都不满足，则两个句子之间不满足模板变换。
- c) 对两个句子剩余的词分别两两计算其词距离。假如两个句子分别剩余的词为，句 1：“鲜花”、“多少钱”、“包邮”。句 2：“鲜花”、“便宜”、“免运费”。那么其距离矩阵如表 6-1 所示：

表 6-1

	鲜花	多少钱	包 邮
便宜	0.001	0.01	0.0001
鲜花	1.0	0.01	0.01
免运费	0.011	0.001	0.9

- d) 得到相似矩阵以后，就把两个句子中相似的词替换为一个，假设我们这里用“包邮”替换掉“免运费”。那么两个句子的词向量就变为：句 1：<鲜花、多少钱、包邮>，句 2：<鲜花、便宜、包邮>。
- e) 对于两个句子分别构建 bi-gram 统计向量，则有：
- i. 句 1，<begin,鲜花>、<鲜花,多少钱>、<多少钱,包邮>、<包邮,end>。
  - ii. 句 2，<begin,鲜花>、<鲜花,便宜>、<便宜,包邮>、<包邮,end>。
- f) 这两个句子的相似度由如下公式计算：

$$P(pr_i, pr_j) = \frac{G(pr_i) + G(pr_j) - 2 \cdot U(pr_i, pr_j)}{G(pr_i) + G(pr_j)} = 1.0 - \frac{2 \cdot U(pr_i, pr_j)}{G(pr_i) + G(pr_j)}$$

其中，

- i.  $G(pr_i)$ ：表示句子（phrase） $i$  的 bi-gram 统计向量长度。
- ii.  $U(pr_i, pr_j)$ ：表示句子  $i$  和  $j$  共有的 bi-gram 统计向量长度。

g) 所以上面的例子的相似度为： $1.0 - 2.0 \cdot 2/8 = 0.5$ 。

4. 完成候选语句的提取之后，就要根据候选语句的数量来判断后续操作了。如果筛选的候选语句大于等于要求的数量，则按照句子相似度由低到高选取指定数量的句子。否则要进行句子的复写。这里采用同义词替换和根据指定模板进行改写的方案。

### 6.1.3 实现

实现候选语句计算的代码如图 6-3 所示：

```
Map<String, Double> result = new HashMap<String, Double>();
    if (type == 0) { // 输入为关键词
        result = getKeyWordsSentence(keyWordsList);
    } else {
        result = getWordSimSentence(sentence);
    }
    // 得到候选集数量大于等于要求的数量则对结果进行裁剪
    if (result.size() >= number) {
        result = sub(result, number);
    } else {
        // 得到候选集数量小于要求的数量则对结果进行添加
        result = add(result, number);
    }
}
```

图 6-3

1. 首先根据输入的内容形式选择不同的生成模式进行语句生成。这一步的关键在于对语料库的处理，对于相似关键词和相似语句的筛选。对于关键词的筛选，我们采用布隆算法进行，当然也可以采用索引查找的方式进行。对于候选语句，我们首先用关键词对于语料库进行一个初步的筛选。确定可能性比较大的语句作为后续计算的语句。
2. 对于得到的候选结果，我们都以 map 的形式保存，其中 key 为后续语句，value 为其与目标的相似度。之后按照相似度从低到高进行筛选。至于 map 的排序我们在之前的章节已经介绍了，这里就不再重复了。

实现语句相似筛选计算的代码如图 6.4 所示。

```
for (String sen : sentenceList) {
    //对待识别语句进行分词处理
    List<Item> wordsList1 = parse(sentence);
    List<Item> wordsList2 = parse(sen);
    //首先判断两个语句是不是满足目标变换
    boolean isPatternSim = isPatternSimSentence(wordsList1,
wordsList2);
    if (!isPatternSim) { //不满足目标变换
        //首先计算两个语句的 bi-gram 相似度
        double tmp = getBigramSim(wordsList1, wordsList2);
        //这里的筛选条件是相似度小于阈值，因为 bi-gram 的相似度越小，代表两者
越相似
        if (threshold > tmp) {
            result.put(sen, tmp);
        }
    } else {
        result.put(sen, 0.0);
    }
}
```

图 6-4

1. 首先对待识别的两个语句进行分词，并对分词后的结果进行模板转换的识别，如果满足模板转换的条件，则将语句作为候选语句，并且赋值一个最小的概率。如果不满足则计算两者的 bi-gram 的相似度。再根据阈值进行筛选。
2. 这里使用的 bi-gram 是有改进的，而常规的 bi-gram 是不需要做比例计算的。这里进行这个计算是为了避免不同长度的字符的影响。对于相似度的度量也可以根据自己的实际情况选择合适的度量方式进行。

### 6.1.4 拓展

本节处理的场景是：由文本到文本的生成。这个场景一般主要涉及：文本摘要、句子压缩、文本复写、句子融合等文本处理技术。其中本节涉及文本摘要和句子复写两个方面的技术。文本摘要如前所述主要涉及：关键词提取、短语提取、句子提取等。句子复写则根据实现手段的不同，大致可以分为如下几种。

1. 基于同义词的改写方法。这也是本节使用的方式，这种方法是词汇级别的，能够在很大程度上保证替换后的文本与原文语义一致。缺点就是会造成句子的通顺度有所

降低，当然可以结合隐马尔科夫模型对于句子搭配进行校正提升整体效果。

2. 基于模板的改写方法。这也是本节使用的方式。该方法的基本思想是，从大量收集的语料中统计归纳出固定的模板，系统根据输入句子与模板的匹配情况，决定如何生成不同的表达形式。假设存在如下的模板。

a) rzv n, a a  $\longrightarrow$  a a, rzv n

b) 那么对于（输入）：

这/rzv, 鲜花/n, 真/a, 便宜/a

就可以转换为（输出）：

真/a, 便宜/a, 这/rzv, 鲜花/n

该方法的特点是易于实现，而且处理速度快，但问题是模板的通用性难以把握，如果模板设计得过于死板，则难以处理复杂的句子结构，而且，能够处理的语言现象将受到一定的约束。如果模板设计得过于灵活，往往产生错误的匹配。

3. 基于统计模型和语义分析生成模型的改写方法。这类方法就是根据语料库中的数据进行统计，获得大量的转换概率分布，然后对于输入的语料根据已知的先验知识进行替换。这类方法的句子是在分析结果的基础上进行生成的，从某种意义上说，生成是在分析的指导下实现的，因此，改写生成的句子有可能具有良好的句子结构。但是其所依赖的语料库是非常大的，这样就需要人工标注很多数据。对于这些问题，新的深度学习技术可以解决部分的问题。同时结合知识图谱的深度学习，能够更好地利用人的知识，最大限度地减少对训练样本的数据需求。

## 6.2 RNN 模型实现文本自动生成

6.1.2 节介绍了基于短文本输入获得长文本的一些处理技术。这里主要使用的是 RNN 网络，利用其对序列数据处理能力，来实现文本序列数据的自动填充。下面就对其实现细节做一个说明和介绍。

## 6.2.1 场景

在广告投放的过程中，我们可能会遇到这种场景：由一句话生成一段描述文本，文本长度在 200~300 字之间。输入也可能是一些主题的关键词。

这个时候我们就需要一种根据少量文本输入产生大量文本的算法了。这里介绍一种算法：RNN 算法。在 5.3 节我们已经介绍了这个算法，用该算法实现由拼音到汉字的转换。其实这两个场景的模式是一样的，都是由给定的文本信息，生成另外一些文本信息。区别是前者是生成当前元素对应的汉字，而这里是生成当前元素对应的下一个汉字。

## 6.2.2 原理

同 5.3 节一样，我们这里使用的还是 Simple RNN 模型。所以整个计算流程图如图 6-5 所示。

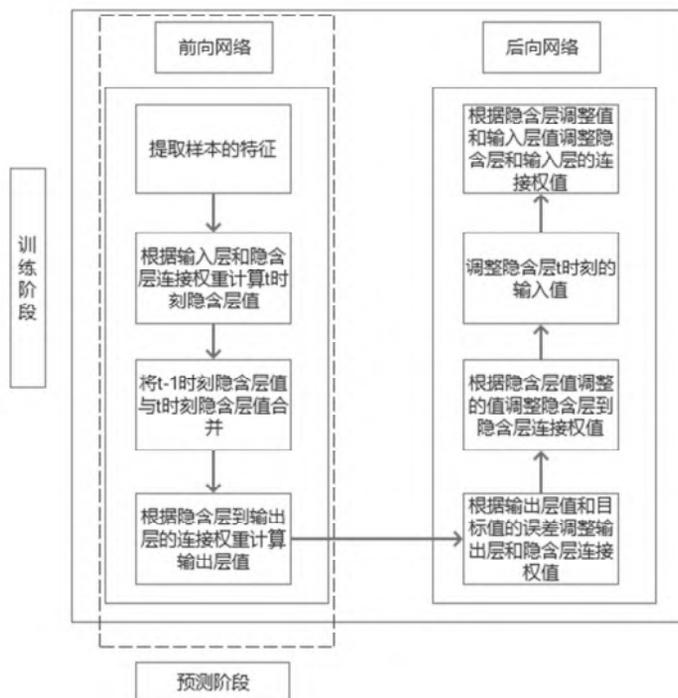


图 6-5

1. 在特征选择的过程中，我们需要更多地考虑上下两段之间的衔接关系、一段文字的

长度、段落中感情变化、措辞变换等描述符提取。这样能更好地实现文章自然的转承启合。

2. 在生成的文章中，对于形容词、副词的使用可以给予更高的比重，因为形容词、副词一般不会影响文章的结构，以及意思的表达，但同时又能增加文章的吸引力。比如最好的，最漂亮的，最便宜的等，基本都是百搭的词，对于不同的名词或者动名词都可以进行组合，同时读者看到以后，一般都有欲望去深度了解。
3. 对于和主题相关的词，可以在多处使用，如果能替换为和主题相关的词都尽量替换为和主题相关的词。因为这个不仅能提升文章的连贯性，还能增加主题的曝光率。
4. 上面这些是对于广告场景提出的一些经验之谈，其他场景的模式或许不太适用，不过可以根据自己的场景确定具体的优化策略。
5. 具体的计算流程和 5.3 节基本一致，在这里就不再赘述了。

### 6.2.3 代码

实现特征训练计算的代码如图 6-6 所示：

```
public double train(List<double[]> x, List<double[]> y) {
    alreadyTrain = true;
    double minError = Double.MAX_VALUE;
    for (int i = 0; i < totalTrain; i++) {
        //定义更新数组
        double[][] weightLayer0_update = new
double[weightLayer0.length][weightLayer0[0].length];
        double[][] weightLayer1_update = new
double[weightLayer1.length][weightLayer1[0].length];
        double[][] weightLayerh_update = new
double[weightLayerh.length][weightLayerh[0].length];
        List<double[]> hiddenLayerInput = new ArrayList<double[]>();
        List<double[]> outputLayerDelta = new ArrayList<double[]>();
        double[] hiddenLayerInitial = new double[hiddenLayers];
        //对于初始的隐含层变量赋值为 0
        Arrays.fill(hiddenLayerInitial, 0.0);
        hiddenLayerInput.add(hiddenLayerInitial);
```

图 6-6

```
double overallError = 0.0;
//前向网络计算预测误差
overallError = propagateNetWork(x, y, hiddenLayerInput,
    outputLayerDelta, overallError);
if (overallError < minError) {
    minError = overallError;
}else {
    continue;
}
first2HiddenLayer =
Arrays.copyOf(hiddenLayerInput.get(hiddenLayerInput.size()-1),
hiddenLayerInput.get(hiddenLayerInput.size()-1).length);
double[] hidden2InputDelta = new
double[weightLayerh_update.length];
//后向网络调整权值矩阵
hidden2InputDelta = backwardNetWork(x, hiddenLayerInput,
    outputLayerDelta,
hidden2InputDelta,weightLayer0_update, weightLayer1_update,
weightLayerh_update);
weightLayer0 = matrixAdd(weightLayer0,
matrixPlus(weightLayer0_update, alpha));
weightLayer1 = matrixAdd(weightLayer1,
matrixPlus(weightLayer1_update, alpha));
weightLayerh = matrixAdd(weightLayerh,
matrixPlus(weightLayerh_update, alpha));
}
return -1.0;
}
```

图 6-6 (续)

1. 首先对待调整的变量进行初始化，在完成这一步之后，就开始运用前向网络对输入值进行预测，完成预测之后，则运用后向网络根据预测误差对各个权值向量进行调整。
2. 这个过程需要注意的是，每次权值的更新不是全量更新，而是根据学习速率 `alpha` 来进行更新的。其他的步骤基本和之前描述的一样。

实现预测计算的代码如图 6-7 所示：

```
public double[] predict(double[] x) {
    if (!alreadyTrain) {
        new IllegalArgumentException("model has not been trained, so can not
to be predicted!!!");
    }
    double[] x2FirstLayer = matrixDot(x, weightLayer0);
    double[] firstLayer2Hidden = matrixDot(first2HiddenLayer,
weightLayerh);
    if (x2FirstLayer.length != firstLayer2Hidden.length) {
        new IllegalArgumentException("the x2FirstLayer length is not
equal with firstLayer2Hidden length!");
    }
    for (int i = 0; i < x2FirstLayer.length; i++) {
        firstLayer2Hidden[i] += x2FirstLayer[i];
    }
    firstLayer2Hidden = sigmoid(firstLayer2Hidden);
    double[] hiddenLayer2Out = matrixDot(firstLayer2Hidden,
weightLayer1);
    hiddenLayer2Out = sigmoid(hiddenLayer2Out);
    return hiddenLayer2Out;
}
```

图 6-7

1. 在预测之前首先要确定模型是不是已经训练完成，如果没有训练完成，则要先进行训练得到预测模型。当然这是一个跟具体业务逻辑有关的校验，这主要是针对预测和训练分开的情况，如果训练和预测是在一个流程内的，则也可以不用校验。
2. 在得到训练模型之后，就根据前向网络的流程逐步计算，最终得到预测值。因为我们这里是一个分类问题，所以最终是选择具有最大概率的字作为最终的输出。

## 6.2.4 拓展

文本的生成，按照输入方式不同，可以分为如下几种：

1. 文本到文本的生成。即输入的是文本，输出的也是文本。
2. 图像到文本。即输入的是图像，输出的是文本。
3. 数据到文本。即输入的是数据，输出的是文本。

4. 其他。即输入的形式为非上面三者，但是输出的也是文本。因为这类的输入比较难归纳，所以就归为其他了。

其中第 2、第 3 种最近发展得非常快，特别是随着深度学习、知识图谱等前沿技术的发展。基于图像生成文本描述的试验成果在不断被刷新。基于 GAN（对抗神经网络）的图像文本生成技术已经实现了非常大的图谱，不仅能够根据图片生成非常好的描述，还能根据文本输入生成对应的图片。

由数据生成文本，目前主要应用在新闻撰写领域。中文和英文的都有很大的进展，英文的以美联社为代表，中文的则以腾讯公司为代表。当然这两家都不是纯粹地以数据为输入，而是综合了上面 4 种情况的新闻撰写。

从技术上来说，现在主流的实现方式有两种：一种是基于符号的，以知识图谱为代表，这类方法更多地使用人的先验知识，对于文本的处理更多地包含语义的成分。另一种是基于统计（联结）的，即根据大量文本学习出不同文本之间的组合规律，进而根据输入推测出可能的组合方式作为输出。随着深度学习和知识图谱的结合，这两者有明显的融合现象，这应该是实现未来技术突破的一个重要节点。

## 第 2 篇

# 自然语言处理系统基础算法

在第 1 篇，我们从应用的角度介绍了一些自然语言处理的场景和使用的一些算法。本篇我们就从一个自然语言处理系统较为基础的实现角度对自然语言处理中用到的一些算法进行介绍。这些基础的实现主要包括：分词、词性标注、句法分析等。



## 第 7 章

# Dijkstra 算法

首先我们介绍一种用于分词的基础算法，该算法是一个最短路径搜索图的算法，算法本身可以使用的场景很多，比如旅行商问题、物流配送问题，等等。这里主要介绍其在分词场景中的应用。

## 7.1 算法应用原理介绍

在自然语言处理中，一般情况下第一步就是对文本进行分词，所以我们这里首先介绍的也是用于分词的算法。本章要介绍的是一种基本的算法，最短路径中的 Dijkstra 算法。

Dijkstra 算法是一个图算法，通过这个算法，可以找到从指定的起点开始，到其所连接的各点的最短距离。应用到分词中来说就以给定的待分词的语句中的某个字为起点，找到一条最短的路径（最佳的分词结果）。假如输入的文本为：“英雄救美”。同时基于词典，我们已经构造了如图 7-1 所示的一个字与字的关联图。

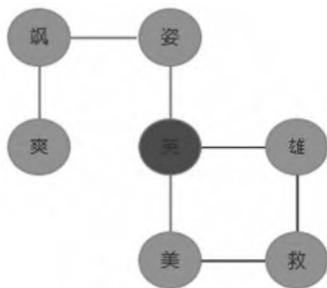


图 7-1

那以“英”字为开头，则会沿着红线所示的路径，走到最后，找到“英雄救美”这个词。这样最终就可以实现对输入文本的分词。这里只是一个简单的示例，实际运用中要根据路径边上的权重进行最优的选择。下面对于算法的具体实现进行详细的介绍。

## 7.2 算法数学原理介绍

首先对 Dijkstra 算法做一个简要的说明：一个最优路径寻找的算法，适用的图形可以是无向图也可以是一个有向图。对于最终的路径有可能存在多条，这个特性会在不同的应用场景中表现为好或者坏的属性，在分词应用中，这个特性不是一个好特性，因为他会涉及语义消歧的问题。但是如果对于方案选择的场景倒可能是一个好特性，因为可以保证不把所有的鸡蛋放在一个篮子里。但是这种情况也不是一定存在的。Dijkstra 算法的实现步骤如下。

1. 找到待识别路径的起点 A 在图中的位置。将图中其他点到 A 点的距离均设置为

infinity。

2. 遍历起点 A 直接连接的所有点，将 A 点与对应点的边权重作为 A 点到该点的距离。并将所有的点按照距离由小到大排序，放到一个 FIFO（first in first out）序列 S 中。
3. 将 A 点到其直接连接的点中距离最小的点 Fo 作为路径中的第二个点。
4. 对 S 中待放出的元素 Fo 遍历其直接连接的所有点，如果对应点经过 Fo 到 A 点的距离小于当前到 A 点的距离，那么就替换当前的距离值。并将发生值替换的点存入到待更新路径的列表中。并将所有的点按照距离由小到大排序，放到序列 S 中。
5. 将 Fo 点到其直接连接的点中距离最小的点 Fn 作为路径中的下一个点。
6. 遍历待更新路径列表中的点，计算经过新路径到当前点的距离，是不是比已保存的路径更短。如果是，则更新当前路径。遍历完成，清空列表。
7. 重复第 4、5、6 步，直到到达终点 E 为止。

整个过程的图例显示如下。

1. 假设由如图 7-2 所示的初始连接图。

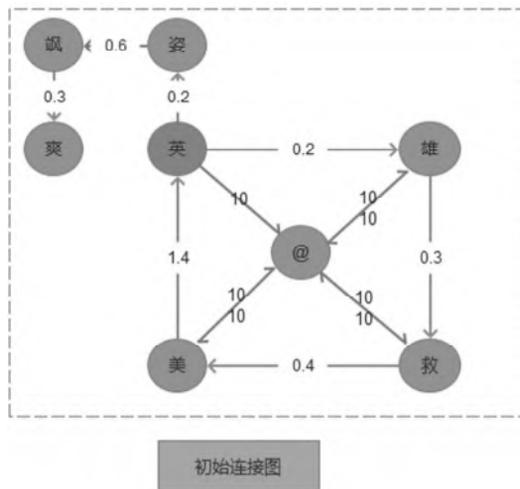


图 7-2

2. 如图 7-2 所示构建了一个有向图，因为字与字之间的连接是有顺序的，所以构建一

个有向图更能合适，当然这里也可以构建一个无向图，只要在处理的时候，进行相应的策略调整即可。首先从起点“英”开始，有两条路可以到达目标“雄”。经过比较发现，直接连接的“英雄”边最短（连接权重和最小）。

a) “英” —— “雄”：0.2

b) “英” —— “@” —— “雄”：20

所以第一个选定的路径为<英，雄>。示意图如图 7-3 所示：

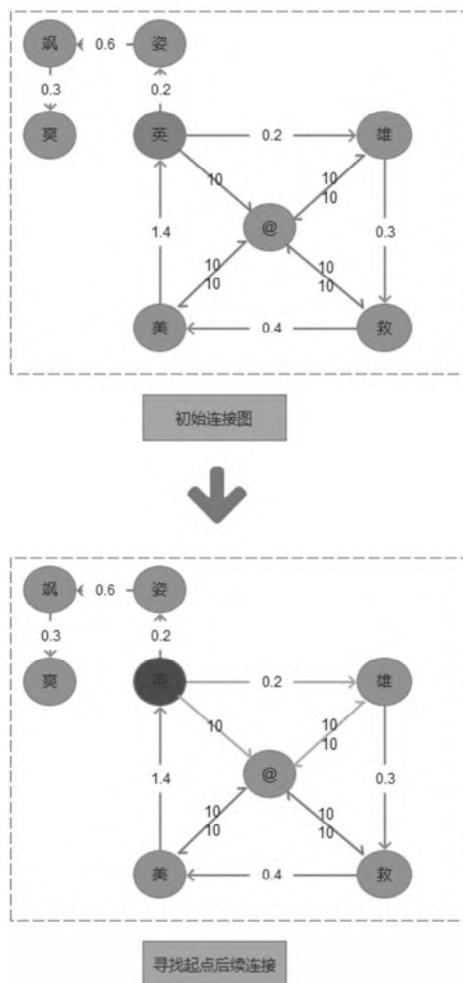


图 7-3

3. 顺次完成后续的“救”、“美”两个字的 가径选择，到达最终节点“美”之后，则不

再进行扩展，也就是不再对“美”字的候选连接节点进行处理，所以不会再循环到“英”字，如图 7-4 所示。

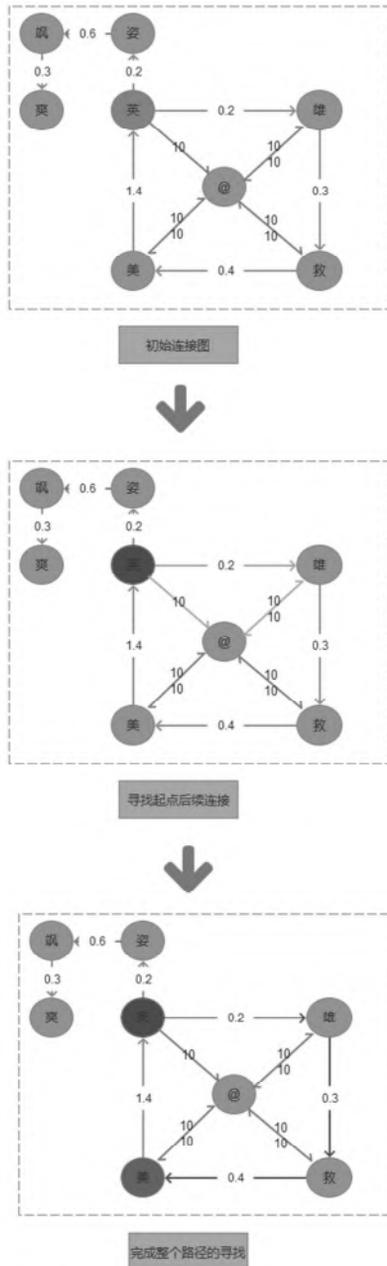


图 7-4

4. 经过上述步骤，我们就完成了整个路径的寻找和选择。这里会发现，对比正常的连接，这里多了一个“@”符，这个符号是表示一个起始符，这个符号可以和任何字进行连接。不同的词之间通过“@”符连接时边的权重和更小。也就是说，如果两个字符串直接通过“@”符连接，路径更短，那么这两个词更加应该独立成词，而不是连接在一起的。
5. 图 7-4 中的边权重是为了说明的方便就给了一个静态的值，而在实际处理中，这个权重是随着前提条件的变化而变化的动态条件概率值。

## 7.3 算法源码说明

实现 Dijkstra 的代码如图 7-5 所示：

```
//为每个节点生成一个存储空间
double dist[] = new double[V];
//记录节点是不是已经添加
Boolean sptSet[] = new Boolean[V];

// 对计算距离进行初始化
for (int i = 0; i < V; i++)
{
    dist[i] = Integer.MAX_VALUE;
    sptSet[i] = false;
}

// 将起点设置为已处理节点
dist[src] = 0;
sptSet[src] = true;

// 寻找最短路径
for (int count = 0; count < V-1; count++)
{
    int u = minDistance(dist, sptSet);
    sptSet[u] = true;
    for (int v = 0; v < V; v++)
        if (!sptSet[v] && graph[u][v]!=0 &&
```

图 7-5

```
        dist[u] != Integer.MAX_VALUE &&  
        dist[u]+graph[u][v] < dist[v])  
        dist[v] = dist[u] + graph[u][v];  
    }
```

图 7-5 (续)

1. 首先对每一个节点声明一个存储单元，以记录其后续的处理结果，同时对于除起点以外的节点均设置为未处理。
2. 根据节点连接权重，更新各个点与起点之间的连接距离，最终得到所要记录的路径。
3. 算法本身的实现是一个广度优先的图搜索算法，即每一次搜索都是首先将离连接点距离最近的点都遍历一遍，再进入到下一步。所以就相当于以起点为圆心，逐层逐层地往外进行扩展，直到到达终点，然后选择整个遍历路径中从起点到终点路径最短的。

## 7.4 算法应用扩展

在分词的应用中，Dijkstra 算法的应用方式也会随着原始图的存储方式不同而不同。比如本节的数据结构中是对单独连接的词之间增加了一个特殊字符，以使得所有的连接都可以在一个图里面表示完成。但是可以将不同的词分开在不同的图中，这样图与图之间的连接就无穷大。这可以根据具体的应用场景，选择适合自己的数据结构进行处理。

Dijkstra 算法所遍历的图只能是正值的边，对于负值的边是不能处理的。相对应的改进算法就是 Bellman-Ford 算法。

上面也说了，在图中有可能存在两条具有相同路径的边连接或者稍微大一点点的连接。而由于本算法只能求出其中的一种，如果想求出次优解那么可以在得到最优解之后，将其中的一个连接点删除，再遍历寻找。因为原始的搜索是由起始点逐层往外扩展的，那么在次优解的求解时就只能从相反的方向进行。

## 第 8 章

# AC-DoubleArrayTrie 算法

第 7 章介绍了最短路径寻找算法中的 Dijkstra 算法。对于分词的实现该算法还是有比较好表现的。本章介绍一种更加高效的分词算法，该算法结合有限状态机和双数组 Trie 树两个数据结构的优势，速度远超过其他的分词算法。

## 8.1 算法应用原理介绍

该算法核心由三个数组组成，分别介绍如下。

1. **index** 数组：用来记录词典中每个字符对应的数字编号。这个编号是自己生成的，生成方式可以自由选择。
2. **base** 数组：用来记录词语前后的关联关系。记录的方式，可以用一个数字记录后接字符为前缀字符的第几个后接字符。对于没有后接字符（即词语的结尾），则可以用一个特定的字符记录。
3. **check** 数组：用来记录，如果遇到词语的结尾，那么要返回到之前已经匹配的那个字符开始继续匹配。

获得了这 3 个数组之后，剩下的就是对输入的字符串逐个扫描，完成所有的匹配即可，最终的结果可以根据自己的策略选择，比如最大匹配字符串。上述过程可以用一个如图 8-1 所示的基础匹配数组构造示意图表示。

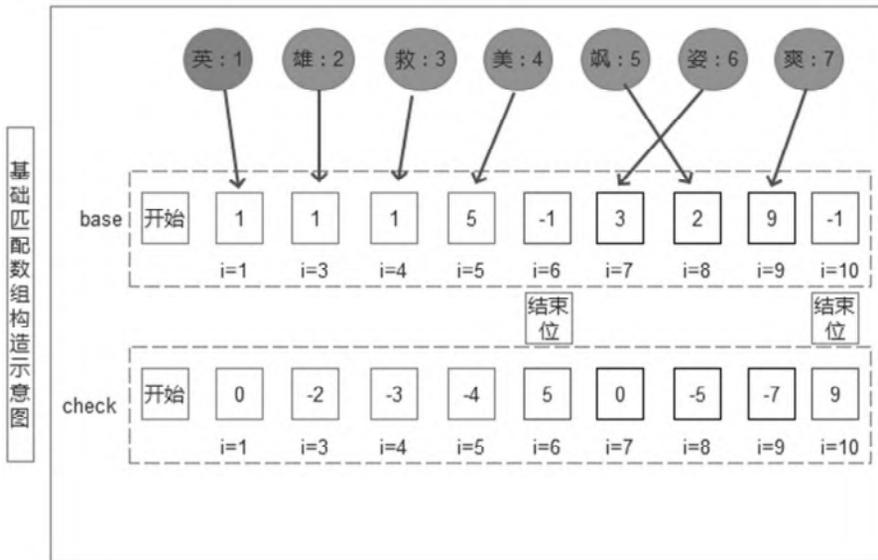


图 8-1

1. 如图 8-1 所示，首先对字典中的字符进行编码。然后将各个词按照先后的顺序逐次放入到 `base` 数组中，`base` 数组的下标代表字符存储的位置，`base` 数组的值代表其下一个字符移动的步长，其默认值为 0。
2. 将一个词结尾字符的下一个连接 `base` 数组的值设置为特定的“结束位符”，比如上文设置为 -1。同时将该结尾字符的值设置为未做位置移动的值，比如上文未做位置移动之前的位置是 5。再将和 `base` 数组“结束位符”相同位置的 `check` 数组的值设置为未做位置移动的值，即 5。
3. 对所有的词重复第 1 和第 2 两步，直到所有的词都添加完成。如果要添加的位置已经被其他字所占用，则左右移动到空的位置，同时将移动的步长记录在前缀字符所在位置的数组值中。
4. 这里需要注意一点，对于同一个词中的叠字要用两个不同的位符来表示。比如“走来走去”中的前后两个“走”字要用两个不同的位符表示。
5. 这里为了说明的方便，就没有把词典中的字符以 `map` 的形式展示，所以不是 `index` 数组的形式。当然这两种方式都可以，只是数组形式表示构建会更快捷一些。

输入数据匹配的示意图如图 8-2 所示。

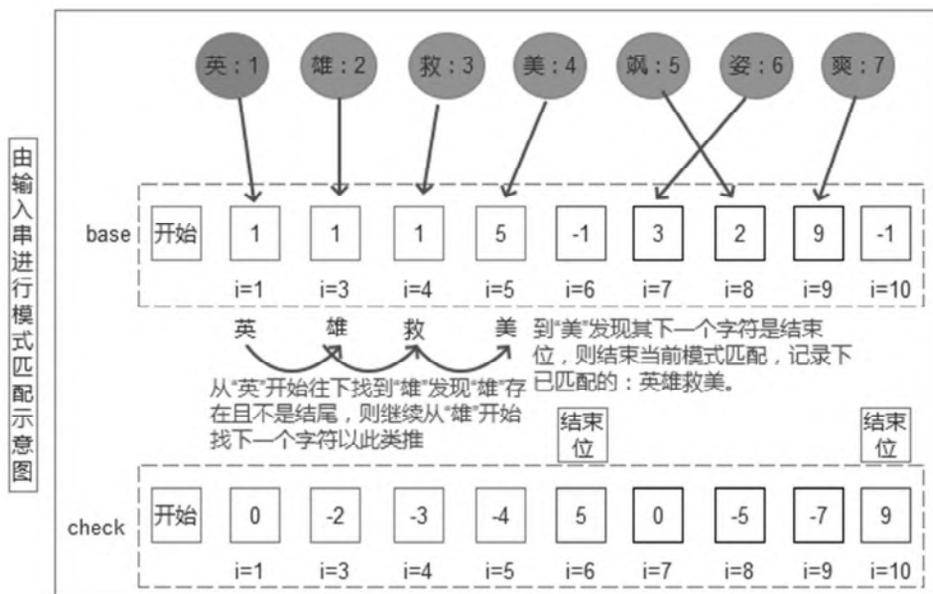


图 8-2

1. 如图 8-2 所示, 按照输入字符的顺序, 逐次寻找其在 `base` 数组中是否存在, 这个存在是以前缀字符为前提条件的。所以它的判断条件是: 以前缀字符对应的 `base` 值+当前字符编号为下标对应的 `base` 数组值不为 0, 用公式表示即为:

$$\text{base}[\text{base}[\text{前缀字符的 base 数组下标}] + \text{当前字符编号}] \neq 0$$

如果是第一个字符, 则它的“前缀字符的 `base` 数组下标”为: “开始”对应的下标 0。

2. 如果当前字符存在, 则看其是不是一个模式串的结尾, 如果是, 则将前边已经连续识别的内容作为一个匹配模式存储下来。判断结尾的条件是否是: 当前字符的下一个连接符的 `base` 数组值为-1, 同时其下一个连接符下标对应的 `check` 数组值等于字符的 `base` 数组值。比如图 8-2 中: “英雄救美”中的“美”字:

- a) “美”字的 `base` 数组下标为 5。
- b) “美”字下一个连接符的 `base` 数组下标为 6。
- c) “美”字下一个连接符的 `base` 数组值为-1, `base[6]=-1`。
- d) “美”字下一个连接符下标对应的 `check` 数组值等于“美”字 `base` 数组值。  
 $\text{check}[6] = \text{base}[5] = 5$ 。

3. 如果输入的字符串已经全部匹配完, 则结束这个过程。否则以结束字符为起点继续往后识别接下来的字符是不是可以连接到当前已经识别的字符串, 如果可以则继续添加, 如果不可以则从以新字符为开头重新开始识别。比如, 如果输入的字符串为“英雄救美人”, 那么在识别出“英雄救美”之后, 接着识别“人”。发现可以连接到已识别的“英雄救美”之后, 则得到一个新的模式串“英雄救美人”。所以结果就是[“英雄救美”、“英雄救美人”]两个模式串。至于最终的输出结果, 则根据预先设定的策略进行选择。比如, 我们选择最大匹配字符串策略, 那么最终保留下来的就是“英雄救美人”。

## 8.2 算法数学原理介绍

其实这个算法是由两个算法组合而成的, 一个是 Aho-Corasick 自动机算法, 另一个是双数组 Trie 树(DoubleArrayTrie)。将 DoubleArrayTrie 称为一种数据结构也可以, 因为它的

实质就是用一个双数组（base、check）来表示一棵 Trie 树。下面对两个算法的数学原理进行说明。

### Aho-Corasick 自动机算法

1. 让我们先考虑一种情况，假设我们的单词已经有 {英雄, 英雄救美, 雄安}。现在要对单词串“英雄安好”进行匹配。如果把已知的词作为一种存在的模式，那么这个匹配过程也就是说要寻找“英雄安好”中已存在的模式有哪些？对于模式识别，我们常用的数据结构为树形结构，那么这里我们也采用这种方式，对已知的词构建一棵树。如图 8-3 所示。

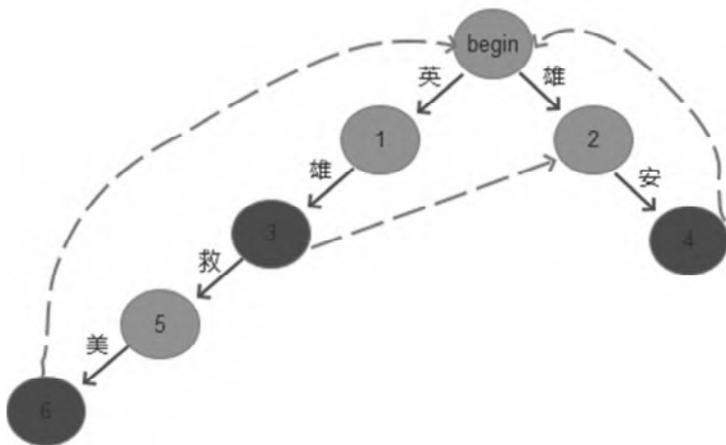


图 8-3

2. 首先树的构建是按照广度优先的顺序进行构建的，也就是说“雄安”中的“雄”后接状态值要小于“英雄”中“雄”的后接状态值。有了这个规则，我们在决定转移状态的时候，就按照由大编号状态向小编号状态转移的策略进行。
3. 对于单个词的结尾，要对其进行标记，如图 8-3 中红色圆圈所示。对于这些“结尾”，即叶子节点，首先在树中寻找其兄弟节点，即有相同前缀字符的状态节点，比如图 8-3 中的“状态 2”和“状态 3”节点。按照原则添加的连接是由大状态节点到小状态节点，所以是从 <3,2> 的连接。如果找不到兄弟节点，如图 8-3 中“状态 6”和“状态 4”节点，那么对于这类叶子节点的转移状态则设置为 begin 节点。
4. 完成了树的构建之后，就只需要对输入串按图索骥，所以对于“英雄安好”的输入，首先从“英”开始，走到“英雄”这个模式，这个时候“雄”后面的字符，按照左

树走不下去了，就按照转移路径跳到右树，然后又匹配出“雄安”。所以最终得到“英雄”和“雄安”两个模式。

5. 所以总结一下 AC 算法的步骤如下：
  - a) 建立模式 Trie 树。
  - b) 给 Trie 添加失败路径。
  - c) 根据 Trie 树，搜索待处理文本。

### 下面来介绍 DoubleArrayTrie 算法

1. 由名字可以知道该算法的本质还是一棵 Trie 树。那么什么是 Trie 树呢，既然名字有树，那就肯定是树的一种，所以其表现形式还是一棵树。只是这个树有一个特点，对于共同的前缀进行压缩存储，即拥有相同前缀的节点都公用一个父节点。这不是树本来应该的样子吗，其实不是，下面我们就通过一个例子来看看什么叫前缀树，如图 8-4 所示。

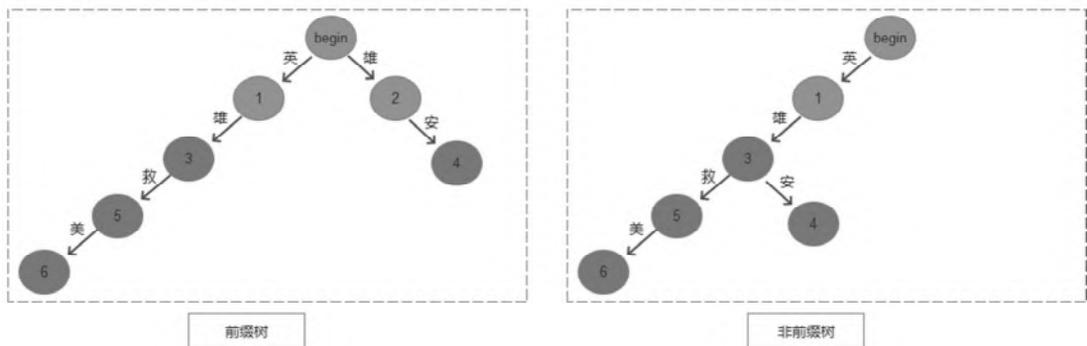


图 8-4

2. 图 8-4 的左图为前缀树，图 8-4 的右图为非前缀树。通过对比可以发现，前缀树是以前缀为分支的依据，虽然节点 A 在树中已经存在，但是如果新串的前缀与已经存在串的前缀不一致，则会新建分支，而不是在已有的分支下面进行添加。与之对应的还有后缀树，这里就不做过多说明了。
3. 对于前缀树有了了解，那么就来说说这棵树的优缺点。优点就是对于任何一个待匹配的串，只要从根节点往下进行遍历，比对前缀是否匹配就可以了，所以查询速度非常快。但是缺点也很明显，会重复存储大量的元素，这样就会造成存储空间过大。

为此学者提出了 Four-Array Trie、Triple-Array Trie 和 Double-Array Trie 结构，其得名源于内部采用的数组的个数，来利用这种结构优点的同时，克服它的缺点。这里我们介绍其中的 Double-Array Trie 结构。

4. Double-Array Trie 包含 base 和 check 两个数组。base 数组的每个元素表示一个 Trie 节点，即一个状态；check 数组表示某个状态的前驱状态。以“英雄”为例来说：

a) “英”字对应的状态为 1；

b) 假设“雄”的字符编号为 2；

c) 那么“雄”对应的 base 数组值（“雄”的状态值）等于“英”字对应的状态加上“雄”字对应的字符编号，所以就是 3。而 check 数组在 3 的值为其前驱（即来源）状态的值 1。即：

$$\text{i. state}[\text{雄}] = \text{base}[\text{state}(\text{英})] + \text{int}(\text{雄}) = 3$$

$$\text{ii. check}[\text{state}[\text{雄}]] = \text{state}(\text{英}) = 1$$

d) 至于  $\text{base}[\text{state}(\text{雄})]$  即“雄”字对应的状态的值是多少，这是由雄后面连接的字的位移所决定的。计算方式由如下公式计算：

$$\text{base}[\text{state}[\text{雄}]] = \text{int}[\text{next}(\text{雄})] - \text{int}(\text{雄})$$

其中，

i.  $\text{base}[\text{state}[\text{雄}]]$ ：表示“雄”对应的下一个字符的位移。

ii.  $\text{int}[\text{next}(\text{雄})]$ ：表示“雄”后面连接的字在词典中的编号，如果当前编号对应的 base 数组位置已经被占用，那么就向后移动指定的步长，直到到达一个空的可用的位置。并把这个位置对应的 base 数组下标作为“雄”字对应的下一个字符的编号。

iii.  $\text{int}(\text{雄})$ ：“雄”字对应的词典编号。

iv. 对于 root 节点， $\text{base}[\text{root}]=1$ 。

5. 有了这两个数组，在查找的时候，就很方便了，只要按照待匹配的字符串顺次完成匹配就可以了。具体判断过程如下：

a) 由公式计算  $\text{state}[x_t]=\text{base}[\text{state}[x_{t-1}]]+\text{int}[x_t]$ ， $x_t$ （即 t 个待匹配的字符）的状态值。

- b) 如果  $\text{base}[\text{state}[x_i]] \neq 0$ , 表示当前位置存在  $x_i$  的转移状态, 也就是以  $x_1 \dots x_i$  为前缀的模式存在已知的模式集合中。如果等于 0, 则表示该模式不存在, 所以匹配的模式就是  $x_1 \dots x_{i-1}$  的模式。
- c) 根据预先定义的匹配策略, 来判断最终的匹配是成功还是失败。如果设置的是全文本匹配, 那么就要求已匹配的模式串最后一个字符对应的状态是结束状态, 否则查询失败。至于状态的值可以自己定义。

上面分别介绍了两种算法, 由上可知, AC-DoubleArrayTrie 算法本质上还是一棵 Trie 树, 只是其表达方式用了双数组的形式。根据 AC 算法的转移状态, 就可以对待匹配字符串中存在的所有模式均找到。

实现 AC-DoubleArrayTrie 算法的代码如图 8-5 所示:

```
//首先计算词典中各个字对应的唯一编码
    Map<String, Integer> map = getUniqueWords(words);
//将词典中各个字对应的唯一编码转换为一维数组
int[] index = getWordIndex(map);
//遍历词典中的各个词, 构建其在 base 数组中的状态值
computeBaseSate(map, index, base);
//遍历词典中的各个词, 构建其在 base 数组中的移动步长
computeBaseValue(map, index, base);
//遍历词典中的各个词, 构建其在 check 数组中的记录值
updateCheckValue(map, index, base, pre);
//遍历词典中的各个词, 构建其转移状态关系网络, 也即记录其转移状态节点
constructFailureStates();
//对待搜索的字符串进行模式发现
List<String> result = findPattern(wordString);
//根据预先设定的结果筛选策略, 确定最终的结果集
result = refinePattern(result, REFINE_TYPE);
```

图 8-5

1. 对于树的构建, 按照首先借助于一个 map 结构实现对唯一字符的寻找。获得了这个结果集就可以实现对唯一字符的编码, 根据这个编码可以确定 base、check 数组的大小, 以及后续 rallocate 数组的大小。
2. 接着就对各个词进行遍历, 构建 base 和 check 数组, 完成这两个数组的构建, 基本上就完成了大部分的工作, 更新的步骤上面已经说得比较多了, 这里就不做过多的叙述了。

3. 在对候选字符串进行模式查找的时候，首先是记录下查找所得的所有模式串，之后再按照之前设定的筛选策略对结果集进行筛选。策略的选择根据需要而定，比如分词的场景，则可以选择最大匹配字符串的策略进行。

## 8.3 算法应用扩展

DoubleArrayTrie 树能高速完成单串匹配，时间复杂度为  $O(n)$ ，即只与待匹配串的长度有关，并且内存消耗可控，然而软肋在于多模式匹配，如果要匹配多个模式串，必须先实现前缀查询，然后频繁截取文本后缀才可实现多匹配。这样，一份文本要回退扫描多遍，性能极低。

AC 自动机能高速完成多模式匹配，用 DoubleArrayTrie 树表达 AC 自动机，就能集合两者的优点，得到一种近乎完美的数据结构。

由于其高性能和可控的内存消耗，在模式匹配领域，该算法具有很高的实用价值，在数据排重方面，相较于布隆算法，它能够实现多样的数据去重，因为在不完全匹配的结果集中，可以设置阈值进行过滤。

## 第 9 章

# 最大熵算法

前面两章介绍了两种用于分词的算法，本章介绍一种用于词性标注的算法，最大熵模型算法。该算法是一种对不确定信息最大保留的算法，下面我们就对其为什么能做到这一点进行说明和介绍。

## 9.1 算法应用原理介绍

首先明确一下我们要解决的问题。对于词性标注，我们要做的事情就是对输入的一个词确定其具体的词性。也就是一个分类问题，将给定的词分到不同的词性类别中。

对于这个问题我们之前已经介绍的很多算法都可以完成，比如朴素贝叶斯分算法、随机森林算法等。那为什么不直接用之前的算法来解决这个问题呢？首先我们回忆一下之前介绍的一些分类算法，它们都是针对历史上已经出现的情况给定一个概率，而对于没有出现过的情况，则认定其概率为 0。对于这个认定，有没有一种改进，有没有一种算法可以实现对未知信息也给出一个不为零的概率，最好还是未知信息能获得的最大的概率。而这个改进就是最大熵模型实现的功能。

从名字我们可以看出，该模型是要使熵最大。根据前面我们已经介绍过熵的概念，熵是对信息纯度的一个度量，熵越小，表示信息纯度越高。那么相反的，如果熵越大，则说明信息的混乱程度越大。具体到分类问题来说，熵越大则说明，目标数据分到不同类的概率相等，对各个类都保持相同的偏好性。这样对于出现未知情形的时候，目标词会被等概率地分配到各个词性，而不是默认指定为 0，这个正是我们想要的。下面就举一个例子对其具体的应用进行说明。

1. 待识别的词：英雄、美人。
2. 词性类别集合：{名词、动词、形容词}。
3. 限定条件：
  - a) 给定词被分到不同类别的概率和为 1。
  - b) “美人”历史上被分为名词的概率为 0.6，“英雄”没有历史记录。
4. 按照最大熵模型求出的结果如 9-1 表所示。

表 9-1

	“英雄”被标记为该词性的概率	“美人”被标记为该词性的概率
名词	1/3	0.6 (已知信息)
动词	1/3	0.2
形容词	1/3	0.2
总和	1.0	1.0

从上面的例子我们可以看到，最大熵模型在保证已知信息被满足的情况下，对于未知信息给予了同等程度的偏好，而这也是其为“不要把鸡蛋放在一个篮子里”最佳实现说法的由来。下面就对其具体实现原理进行说明和介绍。

## 9.2 算法数学原理介绍

1. 对于分类问题，我们首先要确定一个分类函数，以便对于输入数据可以实现分类。注意我们这里是要保证最终的分对于已知信息满足后，要实现对未知信息等概率地分配，所以我们的目标函数是验证这个分配方案是不是得到实现。这样我们就要验证最终分类的结果中，各个类的概率值熵是不是最大。所以我们就选择信息熵作为目标函数，最终优化目标是使得这个目标函数取得最大值。具体函数形式如下：

$$H(Y) = - \sum_{y \in Y} p(y) \cdot \log p(y) \quad (1)$$

其中，

$H(Y)$ : 表示对于随机变量  $Y$  (可能取值为  $Y = \{y_1, y_2, \dots, y_k\}$ ) 的熵。

$p(y)$ : 表示当  $Y=y_i$  时，其概率。

2. 上面介绍了对于最终分类  $y_i$  的信息熵计算公式，但是因为我们最初的输入是  $x$ ，即待分类的词，而不是已经识别的分类，所以就要在式 (1) 中引入  $x$ ，将  $p(y)$  变成在指定  $x$  下取得  $y$  的概率，即  $p(y|x)$ 。这个形式我们之前也见过，就是条件概率的形式，所以最终的熵也变成了条件熵，具体形式如下：

$$H(Y|X) = - \sum_{y \in Y, x \in X} p(y|x) \cdot \log p(y|x) \quad (2)$$

3. 第 2. 中的条件熵只是一个定性的说明，实际的条件熵的计算是由联合熵减去条件值的熵得到的，比如  $H(X,Y)$  为  $x$  和  $y$  的联合熵，条件值  $X$  的熵为  $H(X)$ 。那么当  $X=x$  的条件下， $y$  的条件熵计算公式如下：

$$H(Y|X) = H(X,Y) - H(X)$$

在这里我们就不从公式的角度来进行最终形式的推导，而是从模型应用的角度进行定性的说明。在最大熵模型中，我们在计算由  $x$  到  $y$  的映射时，一般都使用一个泛

化的特征函数，比如现有一个如下形式的特征函数：

$$f(x, y) = \begin{cases} 1, & \text{如果 } x \text{ 的前面 = 数词, 并且 } x \text{ 被分类为量词} \\ 0, & \text{其他} \end{cases}$$

则对于“一打”“三个人”都满足上述的情形。那这样对于式(2)中我们就还需加上对于  $x$  的概率计算，因为不同的  $x$  出现的概率也会不同，所以就得到如下的条件熵公式：

$$H(Y|X) = - \sum_{y \in Y, x \in X} p(x)p(y|x) \cdot \log p(y|x) \quad (3)$$

4. 得到了目标函数后，接下来就是求解这个函数。因为该模型首先要满足已知条件，那怎么将已知条件加入到上面的目标函数中呢？这里就要引入另外一个概念：拉格朗日乘子。拉格朗日乘子法，是一种求解有约束条件优化问题的常用算法，对于等式约束的优化问题，拉格朗日乘子法具有较强的求解优势。而对于非等式问题，则可以用其泛化的 KKT 条件来进行求解，或者对不等式增加松弛变量，将不等式变成等式，再运用拉格朗日乘子法进行求解。同时，对不等式增加松弛变量还有一个好处就是可以让模型对异常值特别是分界面附近的异常值有更好的鲁棒性。对于“美人”的例子具体形式如下。

a) 已知约束条件：

i. 单词分到各个词性类别的概率和为 1，即  $\sum p_i = 1$ 。

ii. 分到名词的类别为 0.6，即  $p_1 = 0.6$ ，为了表示成变量的形式，引入特征函数：

$$f_i = \begin{cases} 1 & i = 1 \\ 0 & i = \text{其他} \end{cases}$$

则该约束条件可以用下式表示：

$$\sum p_i \cdot f_i = 0.6$$

b) 添加上约束条件的最终目标函数为：

$$L(X, Y, \alpha, \beta) = H(Y|X) + \alpha \left( \sum p_i - 1 \right) + \beta \left( \sum p_i \cdot f_i - 0.6 \right)$$

c) 我们将上述条件再进行泛化可以得到一个更一般的目标函数表达式：

$$L(p, \beta) = -H(p) + \beta_0 \left( \sum p(y|x) - 1 \right) + \sum_{k=1}^K \beta_k \left( \sum_{x,y} \tilde{p}(x) \cdot p(y|x) - \tilde{p}(x, y) \right) f_k(x, y) \quad (4)$$

其中,

- i.  $H(p)$ : 表示给定词分配到不同词性类别概率的信息熵。
- ii.  $\beta_k$ : 表示各个拉格朗日乘子的乘数, 也即权重系数。
- iii.  $\tilde{p}(x)$ : 表示由语料库统计的条件  $x$  的概率。
- iv.  $p(y|x)$ : 最终要预测的条件  $x$  下  $y$  分类的概率。
- v.  $\tilde{p}(x, y)$ : 表示由语料库统计的条件为  $x$  时, 分类为  $y$  的概率。所以  $\tilde{p}$  与  $p$  的区别就是前者是由样本数据统计得到的已知结果, 而  $p$  则是最终要预测的结果。
- vi.  $f_k(x, y)$ : 表示第  $k$  约束条件对应的特征函数。

5. 得到了如式 (4) 所示最终的目标函数, 那么接下来就要求解这个公式中各个参数, 使得  $H(p)$  的值最大。这里面涉及两部分参数, 一部分是由约束条件所决定的  $\beta$ , 另一部分是由最终分类概率所决定的  $p$ 。为了在形式上符合惯例 (这个是最优化一个约定俗成的习惯, 将最大极值目标转化为最小极值目标), 对于公式 (4) 左右两边同乘以一个负号, 所以最终的函数可以简化为:

$$L(p, \beta) = H(p) - \sum_{k=0}^K \beta_k \left( \sum_{x,y} \tilde{p}(x) \cdot p(y|x) - \tilde{p}(x, y) \right) f_k(x, y) \quad (5)$$

- a) 要使得  $H(p)$  的值最大, 那就先对  $L$  中的  $\beta$  进行求解使得整个式子最大化 (也即式 (5) 中右边减数部分的值最大化, 这个时候可以假定被减数  $H(p)$  部分是恒定的), 然后再对  $p$  进行求解, 使得整个式子最小化, 用公式表示即为:

$$\min_p \max_{\beta} L(p, \beta)$$

- b) 因为  $\beta$  对应的是约束条件, 而由于约束条件一般很多, 并且由约束条件所决定的决策空间不一定有最终的解, 比如图 9-1 中的 (a) 有一个解即  $C1$  和  $C2$  的交叉点; 而图 (b) 则无解。

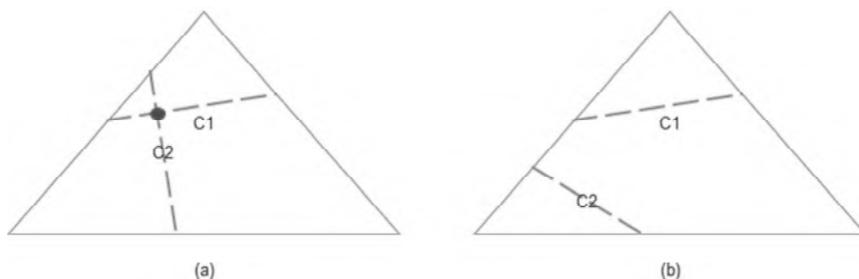


图 9-1

- c) 所以，如果按照上述的顺序求解，一方面求解非常复杂，另一方面最终不一定能求出解。那接下来应该怎么办？经过一系列大牛的努力，对于动态规划的求解，得到一个定理：对于凸优化，其和对偶问题解等价。所以我们就可以通过求解上述问题的对偶问题来得到最终答案。即先对  $p$  求解使得整个式子最小化，再对  $\beta$  求解，使得整个式子最大化。所以最终求解的问题可以转换为下式：

$$\max_{\beta} \min_p L(p, \beta)$$

- d) 按照如上步骤首先对式 (5) 中的  $p$  求偏导，当导数等于 0 时取得其极值。因为  $\beta_0$  在对  $p$  求导之后就是一个常数，所以就单独拿出来。

$$\frac{\partial L(p, \beta)}{\partial p(y|x)} = \sum_{x,y} \tilde{p}(x) (\ln p(y|x) + 1) - \sum_y \beta_0 - \sum_{x,y} \tilde{p}(x) \sum_{k=1}^K \beta_k f_k(x, y)$$

- i. 注意到  $\beta_0$  只与  $y$  有关，所以可以将其乘以一个 1，而  $\sum_x \tilde{p}(x) = 1$ 。所以有：

$$\begin{aligned} \frac{\partial L(p, \beta)}{\partial p(y|x)} &= \sum_{x,y} \tilde{p}(x) (\ln p(y|x) + 1) - \sum_y \beta_0 - \sum_{x,y} \tilde{p}(x) \sum_{k=1}^K \beta_k f_k(x, y) = \\ &\sum_{x,y} \tilde{p}(x) (\ln p(y|x) + 1 - \beta_0 - \sum_{k=1}^K \beta_k f_k(x, y)) = 0 \end{aligned}$$

- ii. 因为  $\tilde{p}(x)$  是由语料库统计得到的，所以值大于 0，则有：

$$\ln p(y|x) + 1 - \beta_0 - \sum_{k=1}^K \beta_k f_k(x, y) = 0。$$

$$\text{所以 } p(y|x) = e^{-1+\beta_0+\sum_{k=1}^K \beta_k f_k(x,y)} = \frac{e^{\sum_{k=1}^K \beta_k f_k(x,y)}}{e^{1-\beta_0}} \quad (6)$$

- iii. 因为最终  $x$  分到各个  $y$  的概率和等于 1，而  $e^{1-\beta_0}$  是一个常数，在规范化的过程中也会消失，所以对于式 (6) 进行规范化得

$$P_{\beta}(y|x) = \frac{1}{Z_{\beta}(x)} e^{\sum_{k=1}^K \beta_k f_k(x,y)} \quad (7)$$

$$\text{其中, } Z_{\beta}(x) = \sum_y e^{\sum_{k=1}^K \beta_k f_k(x,y)}$$

e) 有了最终的表达式, 接下来就是对最终的参数进行迭代求解, 这里介绍使用改进的迭代尺度法 (Improved Iterative Scaling) 进行求解。按照惯例我们对于详细的数据证明部分不做过多的说明, 这里只是说明该迭代解决的问题。

- i. 首先得到了待求解的表达式, 即式 (7) 所表示的形式, 那接下来就是对参数进行求解, 但是因为我们的特征函数构建的  $x$  到  $y$  的映射不是一一对应的, 所以解析解很难求, 就要实现一个逐步迭代的参数求解方法, 即由  $\beta^i \leftarrow \beta^i + \delta^i$  进行求解。
- ii. 那怎么实现逐步迭代求解呢? 就想到了迭代求解中常用的一种方法, 最大似然求解: 如果  $\beta^i + \delta^i$ , 能使得模型的对数似然函数增大, 那么就能通过逐步迭代找到对数似然函数的最大值, 即求得最优解。所以这里就通过构造对数似然函数对参数进行求解。
- iii. 但是因为约束条件很多, 所以对应的  $\beta_k$  是非常多的, 同时相互之间也有依赖, 所以没有办法一次性地对  $\beta_k$  这个向量进行全部的更新。因此就需要单个参数的更新, 即:  $\beta_k^i \rightarrow \beta_k^i + \delta_k^i$ 。那怎么实现这个要求呢? IIS 算法引进了一个新的统计量:

$$f^{\#}(x,y) = \sum_k f_k(x,y)$$

这个统计量的使用还是比较复杂的, 简单来说, 就是借助于特征函数  $f_i$  为  $\{0,1\}$  函数, 所以  $f^{\#}$  是对  $f_i$  的一个计数, 那么总和肯定大于任何一个分项, 这样在一个单调减函数里面, 就一定有分项的函数值大于总和的函数值, 这就用总和的函数替换分项的下限, 也即原来式子的下限, 这样最终求得的似然函数解一定大于原来的似然函数解 (正是我们想要的: 找到对数似然函数的最大值)。

iv. 所以最终的更新公式为:

$$\sum_{x,y} \tilde{p}(x)p(y|x)f_k(x,y)e^{\delta_i \sum_{k=1}^K f_k(x,y)} = E_{\tilde{p}}(f_i)$$

其中,

- (1)  $\tilde{p}(x)$ : 表示在语料库中条件  $x$  的出现概率。
  - (2)  $p(y|x)$ : 表示在预测的条件  $x$  下, 样本被分到  $y$  的概率。
  - (3)  $f_k(x, y)$ : 特征函数  $k$  在条件  $x$  和  $y$  下的取值。
  - (4)  $e^{\delta_i \sum_{k=1}^K f_k(x, y)}$ : 表示由特征计数值调整的更新变量值  $\delta_i$  的指数值。
  - (5)  $E_{\tilde{p}}(f_i)$ : 表示在语料库中特征函数  $i$  对应的期望值。
6. 至此我们就完成了整个模型的推导和最终求解函数的介绍和说明。再小结一下最大熵模型的特点, 一个是对于未知的分类会给予最大的期望值, 这样可以使得我们损失的期望值最小。另一个是模型最终可以推导成为一个指数模型, 所以指数模型的凸函数性质都会被继承, 这在模型的求解和使用中具有很好的特性。

### 9.3 算法源码说明

实现最大熵模型算法的部分代码如图 9-2 所示:

```
//创建特征函数 f
    createFeatureFunction();
//计算函数 f 的经验期望值
    caculateFunctionsExpects();
//计算输入值 x 的经验期望值
    caculateXExpects();
//运用 iis 算法对样本数据进行学习建模
public void train(){
    for (int i = 0; i < ITERATIONS; i++) {
        for (int j = 0; j < functions.size(); j++) {
            double delta = iisSolve(empiricalExpects[j],j);
            Error[i] += delta;
        }
    }
}
```

图 9-2

1. 首先对于资料库的数据进行特征统计，包括对特征函数的统计、每个特征函数的期望值、变量  $x$  对应的经验概率。
2. 获得了上述统计值之后，就相当于获得了参数迭代中每一次迭代中所需要的参数，下面就是对参数进行训练学习，直到最终参数稳定下来为止。
3. 得到训练模型之后，就根据式（6）分别计算样本属于不同类的概率，最终选择最大的概率标签作为输出标签。

## 9.4 算法应用扩展

最大熵模型的一个明显的优点就是它的特征函数具有很好的泛化，所以使用者只要专注于特征的选择；而不要考虑如何使用这些特征，以及特征之间是否满足独立性假设。因为模型本身对于特征之间的使用更多的是计数属性，所以只需要对于特征进行统计即可，而不需要保证变量之间相互独立。但是，这样的结果就是计算量巨大。所以在工程上实现方法的好坏就决定了最终模型使用的效果。

最大熵模型除了可以应用于词性标注，也同样可以应用于句法分析、文本分类、问题回答等应用场景。

# 第 10 章

## CRF 算法

第 9 章介绍了使用最大熵模型做词性标注,本章介绍另外一个算法:CRF (conditional random field, 条件随机场) 模型,该算法模型的思想也主要来源于最大熵模型和隐马尔科夫模型,下面就对其进行说明和介绍。

## 10.1 算法应用原理介绍

在词性标注场景下，输入的是一串已知的文本，输出的是这些文本对应的词性。具体流程如图 10-1 所示。

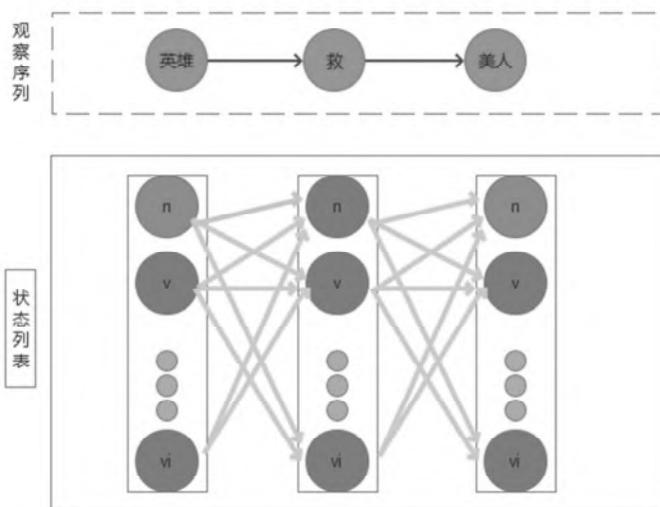


图 10-1

1. 如图 10-1 所示，我们对其过程似曾相识？是的，这个过程和隐马尔科夫过程是一致的，在做拼音混合识别时，已经介绍过这个过程。对于词性标注也是一样的过程，由观察到的序列来推断其最有可能的词性组合，也就是词性路径。这也就是为什么说 CRF 来源于隐马尔科夫模型的原因。
2. 假如我们的信息完备，也就是我们的学习库很全面，那么我们最终可以得到如图 10-2 中红色链条所示的唯一一条词性标注路径。

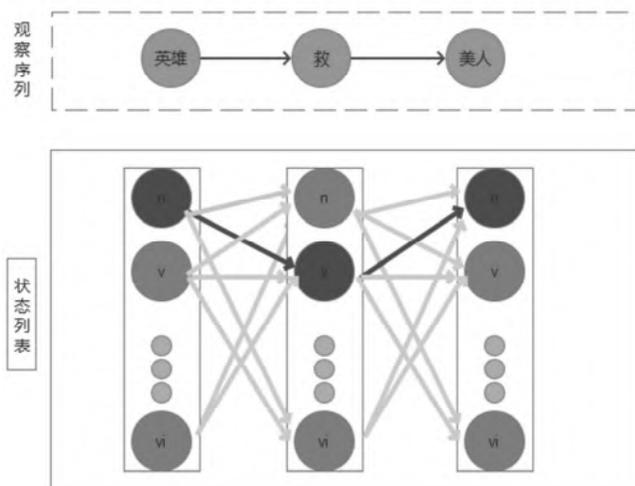


图 10-2

3. 但是在大部分情况下我们的样本数据并不是那么全面的，所以对于部分情况就无法覆盖，这个时候我们又不想让没有样本覆盖的词性概率为 0，那么怎么办？对于这种情况，在第 9 章介绍最大熵模型的时候，我们已经提出过一个处理办法，就是在保证已知信息被满足的情况下，最大程度地保留信息的不确定性，也即分类的均衡性。这也就是为什么说 CRF 来源于最大熵模型的原因。
4. 经过上述步骤，我们就可以得到预测模型所需的参数，然后利用维特比算法进行预测就可以了。具体的计算流程如图 10-3 所示。

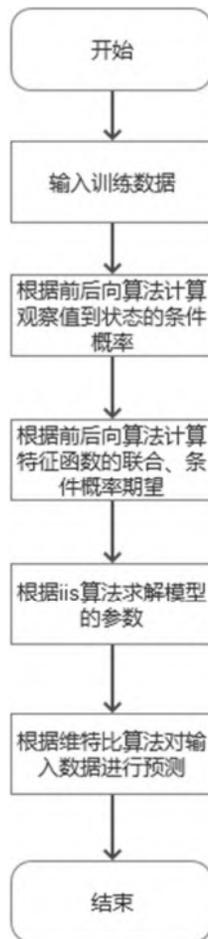


图 10-3

5. 在图 10-3 中有一个利用前后向算法计算观察值到预测状态的条件概率，在上面的步骤中没有说明，现在就来对其进行说明。先让我们来看一个如表 10-1 所示的例子。

表 10-1

句子编号	原 句	对应的词性标注结构
1	英雄救美人	英雄/n 救/v 美人/n
2	英雄美人	英雄美人/n

对于“英雄”和“美人”这两个词在不同的句子中，它们会有不同的词性标注方式。所以对一个句子进行词性标注的时候，最好是能从全句的角度进行考虑。而这个正

是条件随机场（CRF）模型实现的效果。也就是利用前后向算法计算观察值到预测状态的条件概率这一步要做的事情。

下面就对其实现的数学原理进行说明和介绍。

## 10.2 算法数学原理介绍

CRF 模型是概率图模型中的一种，用于解决词性标注问题的模型是 CRF 的线性模型，即线性链条件随机场（linear chain conditional random field）模型。之所以叫做线性链条件随机场，是因为输入的数据是一个链式结构的序列数据，而不是呈图状结构分布的。图 10-4 显示了二者的区别。

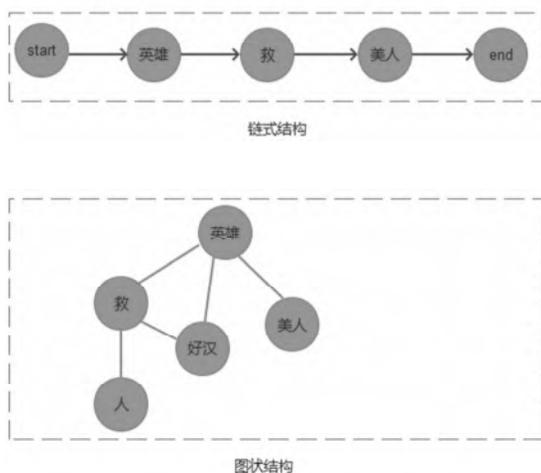


图 10-4

首先我们给出 CRF（条件随机场）模型中所涉及的一些基础定义。

1. CRF 定义：CRF 是给定随机变量  $X$  条件下，随机变量  $Y$  的马尔可夫随机场。
2. 马尔可夫随机场定义：设有联合概率分布  $P(Y)$ ，用无向图  $G=(V,E)$  表示，在图  $G$  中，节点  $V$  表示随机变量，边  $E$  表示随机变量之间的依赖关系。如果联合概率分布  $P(Y)$  满足成对、局部或全局马尔可夫性，就称此联合概率分布为概率无向图模型（probability undirected graphical model），或马尔可夫随机场（Markov random field）。

3. 成对马尔可夫性定义：设  $u$  和  $v$  是无向图  $G$  中任意两个没有边连接的节点，节点  $u$  和  $v$  分别对应随机变量  $Y_u$  和  $Y_v$ 。其他所有的节点为  $O$ （集合），对应的随机变量组是  $Y_o$ 。成对马尔可夫性是指，在给定随机变量组  $Y_o$  的条件下，随机变量  $Y_u$  和  $Y_v$  是条件独立的，即：

$$P(Y_u, Y_v | Y_o) = P(Y_u | Y_o) P(Y_v | Y_o)$$

也就是说：没有直连边的任意两个节点都是独立的。

4. 局部马尔可夫性定义：设  $v \in V$  是无向图  $G$  中的任意一个节点， $W$  是与  $v$  有边连接的所有节点， $O$  是  $v, W$  以外的其他所有节点。 $v$  表示的随机变量是  $Y_v$ ， $W$  表示的随机变量组是  $Y_w$ ， $O$  表示的随机变量组是  $Y_o$ 。局部马尔可夫性是指，在给定随机变量组  $Y_w$  的条件下，随机变量  $Y_v$  与随机变量组  $Y_o$  是独立的，即：

$$P(Y_v, Y_o | Y_w) = P(Y_v | Y_w) P(Y_o | Y_w) \quad (1)$$

由式 (1) 可以得到（以式 (1) 为中心，分别向两边进行扩展）：

$$\begin{aligned} P(Y_v | Y_o, Y_w) P(Y_o, Y_w) &= P(Y_v, Y_o, Y_w) = P(Y_v, Y_o | Y_w) * P(Y_w) = P(Y_v | Y_w) P(Y_o | Y_w) * P(Y_w) \\ &= P(Y_v | Y_w) P(Y_o, Y_w) \end{aligned}$$

所以有：

$$P(Y_v | Y_o, Y_w) = P(Y_v | Y_w) \quad (2)$$

也即：任意节点的条件概率只和与其直接相连的节点有关，用图 10-5 表示如下。

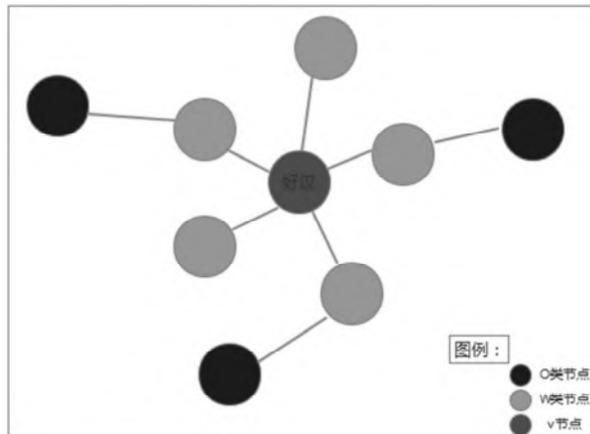


图 10-5

5. 全局马尔可夫性定义：设节点集合  $O$ ,  $V$  是在无向图  $G$  中被节点集合  $W$  分开的任意节点集合，如图 10-5 所示。节点集合  $O$ ,  $V$  和  $W$  所对应的随机变量组分别是  $Y_O$ ,  $Y_V$  和  $Y_W$ 。全局马尔可夫性是指，在给定随机变量组  $Y_W$  条件下随机变量组  $Y_O$  和  $Y_V$  是条件独立的，即：

$$P(Y_O, Y_V | Y_W) = P(Y_O | Y_W) P(Y_V | Y_W)$$

其实成对、局部、全局马尔可夫性在定义上是等价的。如果全局中每一个类的集合只有一个点的时候，就是成对马尔可夫性的场景，如果全局的三个集合中有一个集合只有一个点的时候，就是局部马尔可夫性。

6. 所以再小结一下 CRF 定义：由条件  $X$  所决定的  $Y$  的分布满足成对、局部、全局马尔可夫性。即：

$$P(Y_v | Y_O, Y_W, X) = P(Y_v | Y_W, X)$$

也就是说，任一节点的概率只与其直接相连节点的概率有关。

7. 因为在词性标注中，我们使用的是线性链模型，所以下面给出线性链条件随机场定义：设  $X = (x_1, x_2, \dots, x_n)$ ,  $Y = (y_1, y_2, \dots, y_m)$  均为线性链表示的随机变量序列，若在给定随机变量序列  $X$  的条件下，随机变量序列  $Y$  的条件概率分布  $P(Y|X)$  构成条件随机场，即满足马尔可夫性：

$$P(Y_i | Y_1, \dots, Y_m, X) = P(Y_i | Y_{i-1}, Y_{i+1}, X)$$

其中， $i=1, 2, \dots, m$ （在  $i=1$  或者  $m$  的时候，只取一边的连接）。

8. 线性链条件随机场又根据  $X$  和  $Y$  是否有相同的结构，而分为同结构和非同结构的线性链条件随机场。两者的示意图如图 10-6 所示。
9. 在词性标注这个场景下， $X$  和  $Y$  是满足同结构的，所以对于计算的结构完全可以参照隐马尔可夫模型进行。但是它的观察值又不仅仅只与当前状态值有关，所以在求解参数的过程中又需要调整，即增加了最大熵部分。
10. 在引入最大熵模型之前要给出另外一个定义，无向图中团与最大团的定义：无向图  $G$  中任何两个节点均有边连接的节点子集称为团 (clique)。若  $C$  是无向图  $G$  的一个团，并且不能再加进任何一个  $G$  的节点使其成为一个更大的团，则称此  $C$  为最大团 (maximal clique)。所以由成对马尔可夫性可以知道，只有团内的节点在条件概率上才会有相互影响，非团内的条件概率均是相互独立的。而最大熵模型

在采用 iis 求解时，会用最大似然方法来求解，所以形式上会将待求解的函数转换成连乘积的形式，那么这里只需要将团内的概率转换成连乘积就可以了。这个实现就是通过势函数实现的，具体形式如下：

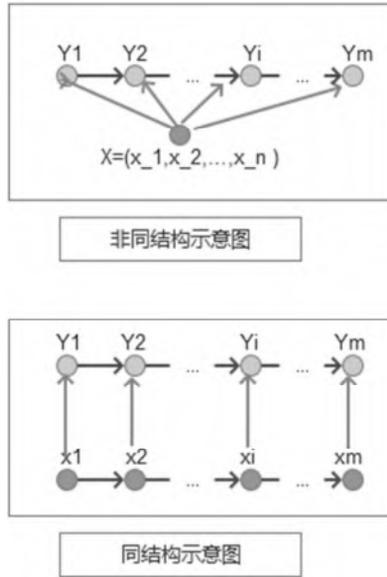


图 10-6

$$P(Y) = \frac{1}{Z} \prod_C \varphi_C(Y_C) \quad \text{式(3)}$$

其中  $Z$  是规范化因子，与最大熵模型中的定义一致，即

$$Z = \sum_y \prod_C \varphi_C(Y_C)$$

11. 上面得到了将无向概率图相连的团节点的联合概率分布变换成了连乘积的形式。对于其在最大团上的该形式表示则是由 Hammersley-Clifford 定理来保证的。形式还是式 (3) 所示，只是集合由团变成了最大团。而概率无向图模型的联合概率分布表示为其最大团上随机变量的函数的乘积形式的操作，称为概率无向图模型的因子分解 (factorization)。同时为保证由势函数累乘后除以规范化因子得到的概率为正，我们规定势函数是严格正实值的函数。为满足势函数是严格正实值的函数，常常选用指数势函数，即：

$$\varphi_C(Y_C) = e^{-H_C(Y_C)}$$

这个形式就和最大熵模型的指数形式完全一致了，所以求解模型的参数就回到了最大熵模型的求解过程。

上面我们给出了 CRF 模型中涉及的一些基础定义，从这些定义我们对于 CRF 模型的马尔科夫属性、最大熵属性有了一些了解。在上一节应用原理部分，我们也介绍了其最大熵属性主要是体现在其分配状态值概率的时候。有了这个策略，就可以保证每一个词的词性能够尽可能地覆盖全部的情况，即使在样本中没有的情况，也会以一定的概率进行体现。

关于隐马尔科夫模型和最大熵模型的数学推导部分，在之前的章节中已经有所介绍，这里就不再赘述了。下面就用图 10-7 对其中各个算法在整个计算流程中的作用进行说明。

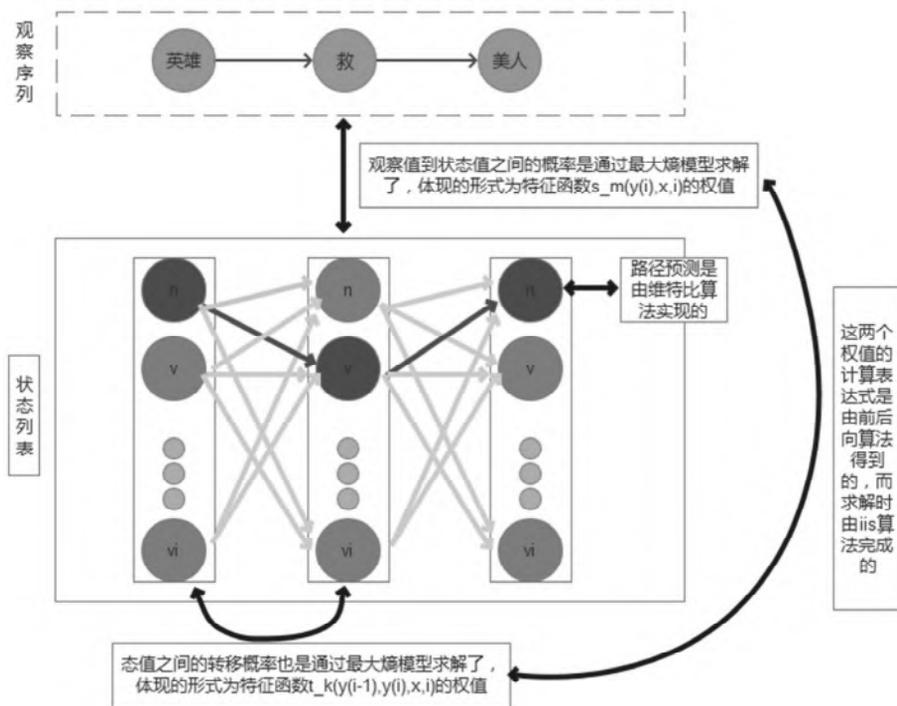


图 10-7

1. 如图 10-7 所示，CRF 整个计算框架是和隐马尔科夫模型一致的，所以模型的 5 个组成元素是一样的：观测值、状态值、观测值到状态值的转换概率 A、状态值之间的转移概率 B、状态值初始概率。同样的，模型最终求解的也是一个最佳路径。
2. 首先，对于 A 和 B 的表达式建立是由前后向算法完成的，而求解是由最大熵模型完成的，使用的求解方式和最大熵模型一致，这里采用的仍然是 iis 算法。

3. 在经过训练样本学习之后就得到了最终的概率模型，使用概率模型进行预测时，使用的是维特比算法。

对于计算的流程我们已经进行说明和介绍，下面就看看其具体的实现代码。

## 10.3 算法源码说明

实现 CRF 模型算法的部分代码如图 10-8 所示。

```
//提取特征值，即将特征函数应用到训练数据上，提取对应的特征
    getFeature();
//初始化特征值权重
initialAlpha();
/*接下来开始训练模型*/
//首先根据输入链建立计算水晶格，也即路径网络
buildNet();
//运用前后向算法计算最大熵指数函数，用于求解具体的权重
forwardBackward();
//计算最大熵模型的特征函数期望值，用于后续求解优化
calculateExpectation();
//运用 iis 算法求解具体的参数
iisSolved();
//运用维特比算法对求解的函数进行预测，看是不是符合最优解路径
viterbi();
//运用 L-BFGS 算法对求解的参数进行优化
lbfgsOptimize();
//保存最终训练的模型
saved();
```

图 10-8

1. 首先对于资料库的数据进行特征统计，包括对特征函数的统计、每个特征函数的期望值等。同时对于特征函数权重值进行初始化。
2. 获得了上述统计值之后，就相当于获得了参数迭代中每一次迭代中所需要的参数，下面就是对参数进行训练学习，直到最终参数稳定下来。训练过程如图 10-7 所展示的那样，首先是利用前后向算法，获得要求解的指数函数，再利用 iis 算法对指数函数与期望值的等式进行求解，获得这些参数之后，再利用维特比算法进行测试

预测，看看预测的结果是不是和真实的情况一致，如果不一致，再利用 L-BFGS 算法对求解的参数进行优化。如此循环，直到达到模型计算终止条件。

3. 得到训练模型之后，就根据维特比算法，计算样本词性标注的最优路径  $Y=(y_1, y_2, \dots, y_k)$ 。这个最优路径就作为最终的词性标注结果。

## 10.4 算法应用扩展

CRF 由 Lafferty 等人于 2001 年提出，CRF 结合了最大熵模型和隐马尔可夫模型的特点，是一种无向图模型，近年来在分词、词性标注和命名实体识别等序列标注任务中取得了很好的效果。在序列标注中还有另外两个常用的算法：隐马尔可夫模型 (Hidden Markov Model, HMM) 和最大熵马尔可夫模型 (Maximum Entropy Markov Model, MEMM)。HMM 首先出现，MEMM 其次，CRF 最后出现。

三个算法主要思想如下：

1. HMM 模型是对状态到状态间的转移概率和状态到观察值间的转换概率直接建模，统计共现概率。
2. MEMM 模型是对状态到状态间的转移概率和状态到观察值间的转换概率建立联合概率，统计时统计的是条件概率，但 MEMM 容易陷入局部最优，是因为 MEMM 只在局部做归一化。
3. CRF 模型统计了全局概率，在做归一化时，考虑了数据在全局的分布，而不是仅仅在局部归一化，这样就解决了 MEMM 中标记偏置 (label bias) 的问题。

与其他两个算法相比较，CRF 的优点如下：

1. CRF 没有 HMM 那样严格的独立性假设条件，因而可以容纳任意的上下文信息。
2. CRF 是在给定需要标记的观察序列的条件下，计算整个标记序列的联合概率分布 (也即我们开头说的，它考虑的是整体的标注，而不是基于部分信息的标注)，而不是在给定当前状态条件下，定义下一个状态的状态分布。

那么相对应，它的缺点也很明显，就是计算量很大，复杂度高，所以实现的模型也非常大，不利于使用。

## 第 11 章

# 马尔可夫逻辑网算法

前面介绍了用于分词、词性标注、句法分析的一些算法，本章将介绍一种可以进行实体名称识别的算法：马尔可夫逻辑网算法。该算法是马尔科夫网络的一种延伸，是在马尔科夫网络的基础之上添加一阶逻辑推理的功能得到的。

## 11.1 算法应用原理介绍

通过前面章节的介绍，我们知道：马尔科夫网络的目标是对网络中有链接的点之间的条件概率进行求解，最终就可以根据点与点之间的条件概率求出最优路径或者是最大可能性连接。马尔科夫逻辑网的基本逻辑也是一样的，只是网络的构建是根据一阶谓词逻辑所确定的模板进行的。具体步骤如下：

1. 首先确定各个一阶谓词逻辑的取值范围，比如我们有如下一个一阶谓词逻辑：

$$(\forall x)\text{包含公司两个字}(x) \rightarrow \text{公司名称}(x) \quad (1)$$

其中，

- a)  $\forall x$ ：表示任意一个词。
- b) 包含公司两个字(x)：表示词 x 中包含“公司”两个字。在下面的说明中用 `containFlag(x)` 表示。
- c) 公司名称(x)：表示 x 为公司名称。在下面的说明中用 `name(x)` 表示。

那么我们首先要确定 x 的范围，从定义上来看，x 应该限定为名词的范围。所以公式 (1) 所适用的取值范围为名词。

2. 接下来是对一阶谓词逻辑中的谓词（命题）进行实例化。即用实例取值对命题进行判断。比如我们假设候选的名词集合中只有一个词 x。那么实例化后的谓词即如图 11-1 所示。



图 11-1

3. 得到了上面的谓词实例，就对同一个谓词逻辑下的实例进行连接。如图 11-2 所示，这样就建立了一个马尔科夫网络。所以一个一阶谓词逻辑推理公式构成了一系列的团。



图 11-2

4. 得到了马尔科夫网络之后，就要对各个节点之间的联合概率进行求解了。咋一想，所有节点之间的概率计算，那不是要疯掉？但是别忘了，马尔科夫网络需要满足：成对、局部、全局马尔科夫性。根据这三个条件可以推出，最终只有直接相连的两个节点之间需要计算联合概率，而非直接相连的节点之间的联合概率等于两个节点概率直接相乘，因为它们相互独立。由于这个性质就把整个计算网络缩小到一个团上了。而上面我们说了，一个一阶谓词逻辑就对应一系列的团，所以一个由一阶谓词逻辑实例组成的团节点之间的联合概率可以由如下公式进行计算。

$$P(Y) = \frac{1}{Z} \prod_C \varphi_C(Y_C) = \frac{1}{Z} e^{\sum_C w_C f_C(Y_C)}$$

5. 对于马尔科夫逻辑网络，上述形式要稍微改变一下。这是因为之所以要对一阶谓词逻辑进行概率化，是为了克服它的一个缺点：知识绝对化。比如对于“公司名称”这个词语，如果使用一阶谓词逻辑进行判断，那么它就是一个公司名称实体，而实际上它不是。但是由一阶谓词逻辑所表达的知识是非常有用的，特别对所要判断的实例加上一定的条件之后，那么这个知识就可能百分百正确。所以就希望对知识加上概率，让知识在不同的条件下表现出不同的准确度。同时对于知识本身，我们也希望有所量化，比如一个很具体的知识“猫爱吃鱼”在绝大部分情况下都是正确的，但如果具体的实例违反了这个条件，那么就应该有较大的可能性不属于猫这个类别。也就是说这个知识的可信度很高。所以我们就对知识（一阶谓词逻辑公式）也加上一个权重。所以最终得到如下的联合概率分布公式：

$$P(Y) = \frac{1}{Z} e^{\sum_i w_i n_i(Y_C)} \quad (2)$$

其中，

- a)  $P(Y)$ : 表示一个团内各个成员之间的联合概率分布。
  - b)  $w_i$ : 表示一阶谓词逻辑函数的权重。
  - c)  $n_i(Y_C)$ : 表示团成员实例  $C$  作用于一阶谓词逻辑函数时，使得函数为真的个数。
6. 对于马尔可夫逻辑网络，上面只是说了其一个特点：弱化知识绝对化的特点。它还有另一个特点：保证知识在信息全面时得到满足。下面我们就看公式（2）是不是能满足这两个条件，如果满足的话，那么我们就可以直接对公式（2）进行求解获得节点之间的联合概率了。

- a) 首先我们以公式（1）为例子进行说明。假设我们现在就只有公式（1）一个一

阶谓词逻辑，同时取值也只有一个  $x$ ，则根据其是否满足 `containFlag` 和 `name` 的条件得到表 11-1 所示的 4 个可能结果。

表 11-1

是否满足 <code>containFlag</code> 判断	是否满足 <code>name</code> 判断	$(c, n)$ 值	公式 (1) 的值是否为真	最终概率值
是	是	(1,1)	是	$P(1,1) = \frac{e^w}{Z}$
是	否	(1,0)	否	$P(1,0) = \frac{1}{Z}$
否	是	(0,1)	是	$P(0,1) = \frac{e^w}{Z}$
否	否	(0,0)	是	$P(0,0) = \frac{e^w}{Z}$

b) 以  $(c,n)$  值为 (1,1) 的情况为例说明计算过程。

- i. 首先因为我们这里只考虑一个一阶谓词逻辑，所以公式 (2) 中  $e$  的指数部分加和的  $i$  就只有一个值，即  $i=\{1\}$ 。所以公式 (2) 对任一个  $(c, n)$  值，就简化为如下的形式：

$$P(c, n) = \frac{1}{Z} e^{w \cdot n(FOL(c,n))} \quad (3)$$

其中： $FOL(c,n)$ 表示在  $(c, n)$  的条件下，一阶谓词逻辑 (First order logic) 的值，如果为真则该值等于 1，否则等于 0。

- ii. 将 (1,1) 代入公式 (3)，则有： $FOL(c,n)=1$ ，所以 $n(FOL(c,n))=1$ 。所以就得到了最终的概率为：

$$P(1,1) = \frac{e^w}{Z}$$

- iii. 至于一阶谓词逻辑的真假取值，用一句话简单地说就是：只要不违背公式的条件，就为真。比如对于公式(1)，它只要求当 `containFlag=1` 时，`name=1`，至于其他情况都没有要求，所以当 `containFlag=0` 时，不管 `name` 取什么值，公式 (1) 的取值都是真。所以表 11-1 中，只有  $(c, n) = (1,0)$  时，公式 (1) 的值为假，其他三种情况都为真。这个属性对于优化推理算法这一点，后面会进行介绍。

c) 既然有了联合概率的值，下面就来验证一下新的公式是不是能满足最初提出的两个条件：弱化了知识的绝对性，信息全面时又能满足知识所限定的条件。

- i. 首先根据  $Z$  的定义有：

$$Z = p(1,1) + p(1,0) + p(0,1) + p(0,0) = 3e^w + 1$$

- ii. 先来看看在违背知识的条件下，它的概率还是不是 0。从表 11-1 知道违背知识的条件只有一个，即  $(c, n) = (1,0)$ ，其概率为

$$P(1,0) = \frac{1}{Z} = \frac{1}{3e^w + 1}$$

很明显，此时的概率不为 0，所以确认弱化了知识的绝对性：对于不满足知识的条件，不是一票否决（将概率设置为 0），而是给予一定的概率，允许其存在。

- iii. 然后再验证一下，当信息全面时，它的概率是不是满足知识所限定的条件。那就是要验证当名称中包含“公司”时，其为“公司名称”这个实体的概率是不是为 1，即： $p(\text{name}(1)|\text{containFlag}(1))$ 的期望值是不是等于 1。

首先求 $p(\text{name}(1)|\text{containFlag}(1))$ 的条件概率，由贝叶斯公式可得：

$$p(\text{name}(1)|\text{containFlag}(1)) = \frac{p(\text{name}(1), \text{containFlag}(1))}{p(\text{containFlag}(1))}$$

其次求 $p(\text{containFlag}(1))$ 的概率，此概率也叫边缘概率，由全概率公式可得：

$$\begin{aligned} p(\text{containFlag}(1)) &= p(\text{containFlag}(1)|\text{name}(1)) \cdot p(\text{name}(1)) \\ &\quad + p(\text{containFlag}(1)|\text{name}(0)) \cdot p(\text{name}(0)) \\ &= p(1,1) + p(1,0) = \frac{1 + e^w}{Z} \end{aligned}$$

所以：

$$p(\text{name}(1)|\text{containFlag}(1)) = \frac{p(1,1)}{p(1)} = \frac{e^w}{1 + e^w}$$

当权重 $w \rightarrow \infty$ 时，概率趋近于 1，满足一阶谓词逻辑的要求。

- iv. 这样我们就验证了公式（3）是满足我们预设的两个条件的，所以下面就是要对公式中的权重进行求解了。

7. 对于公式中权重的求解就和之前介绍的马尔科夫网络的权重求解一样了：

- a) 首先设立一个最大似然函数。

- b) 然后选择一种迭代的策略进行求解。
- c) 设置迭代终止的条件，完成求解过程。
8. 获得最终的模型之后，就可以根据自己的情况来使用了。对于推理的应用，一种是最大可能性问题的求解：根据给定证据变量集  $x$ ，求解其对应的因变量  $y$  最可能处于的状态。正如 CRF 模型中介绍的：求解一组输入文本最可能的词性标注集合即为其一种应用形式。另外一种就是计算边缘概率和条件概率，这样就可以对每一个节点自身的存在概率进行量化。如上面介绍的：计算一个词如果包含“公司”，那么它属于“公司名称”这个实体的概率，也就是本章开头说的用于实体信息抽取。

## 11.2 算法数学原理介绍

马尔科夫逻辑网是在马尔科夫网络的基础上构建的，所以它的主体还是一个马尔科夫网络。对于马尔科夫网络本身的数学原理，之前章节已经介绍，这里就不再赘述了。这里只对模型参数的求解算法进行一个说明介绍。

马尔可夫逻辑网络的模型参数求解主要有两种方法，一种是伪最大似然估计，另一种是判别训练。这里对其中的判别训练进行一个说明和介绍。

我们已经知道，对于马尔科夫网络参数的求解，一般是需要建立一个最大似然的求解过程，这里为啥不用最大似然而用另外两种方法呢？这就要从公式 (2) 说起。假设我们现在还是用最大似然法来进行求解，那么对公式 (2) 中的一个规则  $F_i$  的权重  $w_i$  求对数似然梯度可以得到如下表示公式：

$$\frac{\partial}{\partial w_i} \log P_w(X = x) = n_i(x) - \sum_k P_w(x_k) n_i(x_k) \quad (3)$$

其中，

1.  $P_w(X = x)$ ：表示当自变量取  $x$  集合，也即在  $x$  的世界中， $w$  权重的分布概率。
2.  $n_i(x)$ ：表示规则  $F_i$  在  $x$  集合上取真的次数。
3.  $x_k$ ：表示任一  $x$  的取值集合，不同的集合会形成不同的团，进而形成不同的世界。
4.  $\sum_k P_w(x_k) n_i(x_k)$ ：是在所有可能的世界中，对于规则  $F_i$  的值进行求和。

从公式(3)可以知道,规则 $F_i$ 的权重 $w_i$ 的对数似然函数梯度,等于当前世界 $x$ 中规则 $F_i$ 的真值个数与在所有可能的世界中规则 $F_i$ 的真值个数的数学期望之差。然而,计算一个规则在世界中的闭规则的个数是不明智的,因为假设一个规则对应两个谓词,每个谓词对应两种结果,那么对于 $x$ 集合(包含 $n$ 个实例)而言,其要计算的真值个数为 $2^{2*n}$ 个。如第一节举的例子,只有一个实例时,对应的是 $2^{2*1} = 4$ 个计算量,如果实例变成2个,那么对应的计算量就是 $2^{2*2} = 16$ 个,这样指数级的增长是非常可怕的,所以计算一个规则在世界中的闭规则的个数是不明智的。且直接计算规则的真值个数的数学期望更是不可行的,因此,直接用最大似然方法实际上并不可行。这也就是为啥不直接使用最大似然估计的原因。下面就对判别训练法进行一个介绍。

1. 判别训练法对应的求解公式还是公式(3)所示的形式。它只是在对参数的求解上进行了一些技巧性的处理。
2. 在介绍具体的处理技巧之前,我们先来介绍一种参数求解法,最大后验估计(maximum a posteriori, MAP)。下面以一个例子进行说明。
  - a) 假设有一个集合 $X = \{x_1, x_2, \dots, x_n\}$ 。不同取值是独立同分布的。
  - b)  $X$ 的概率分布是由参数 $\theta$ 决定的。现在希望求出使得 $X$ 发生概率最大的 $\theta$ 值。
  - c) 由联合概率分布可以得到:

$$p(X|\theta) = \prod_{i=1}^n p(x_i|\theta) \quad (4)$$

- d) 对于公式(4)的求解,我们之前已经介绍了一种方法,对数最大似然,通过对公式(4)求对数,就可以把连乘积转换成加和的式子(这样就可以避免因为小数太小而溢出的问题)。然后对加和的式子求偏导,让偏导数等于0求得极值点的 $\theta$ ,也即最终的解。那么怎么由这个方法引出我们要介绍的最大后验估计呢?这就要从极大似然估计的一个缺点说起:极大似然估计是一个点估计,它只能是参数 $\theta$ 的一个估计值,而不能求出 $\theta$ 的分布。同时极大似然估计是完全根据样本数据进行计算的,所以如果样本有偏,那么最终估计的值也会是有偏的,当然对于其他估计也会出现这个问题。但是如果有一定的先验知识就可以部分克服抽样有偏的问题。比如我们抽样一批人进行患癌症概率估计,结果抽取的都是吸烟的人,那么计算的概率会明显增大,如果加上抽烟人只占全部人口的10%的条件,那么就会好一些。

- e) 而最大后验估计正好可以解决上面的问题。但是一下跳到最大后验估计又有一点跳跃，因为最大后验估计是在贝叶斯估计基础上产生的。对于贝叶斯我们已经比较熟了，所以对于求 $\theta$ 的分布，我们有如下公式：

$$p(\theta|X) = \frac{p(X|\theta) \cdot p(\theta)}{p(X)}$$

由全概率公式可以知道 $p(\theta|X)$ 与 $p(X)$ 无关。那么就有要求使得 $p(\theta|X)$ 概率最大的 $\theta$ 就是求 $p(X|\theta) \cdot p(\theta)$ 的极大值。对于这个式子我们很熟悉了，用对数似然函数进行求解就可以了。

3. 有了最大后验估计这个利器，我们就可以对公式（3）中的参数进行求解了。实际上也就是我们并不去计算各个计数值的期望值或者理论值，只是根据样本数据的近似估计来计算最终的权值。采用这个策略的方法主要有两种，一种叫 Voted Perceptron (VP) 算法，另一种叫 Contrastive Divergence (CD) 算法。下面以 VP 算法为例说明其计算步骤：

- a) VP 算法将所有权重初始化为零。  
b) 然后进行  $T$  次梯度下降，最终返回每次迭代的权重之和的平均值，即

$$w_i = \sum_t w_{i,t} / T$$

由于马尔可夫逻辑网络是一个较新的领域，所以它的解法还不是特别完备，会有各种的不足，对于解法的改进更新也在不断提出。这里介绍的方法只是比较初级的方法，对于解的更好更完备的解法，请参考最新的研究结果。下面就对实现的代码进行一个说明和介绍。

## 11.3 算法源码说明

实现 MLN 模型算法的部分代码如图 11-3 所示。

```
//对基础数据按照逻辑函数进行特征提取
DataBean data = getDataBean(F,d);
//根据逻辑函数和实例数据, 构建马尔科夫网络
buildNet(F,data);
//初始化特征权值权重
initialAlpha();
//利用 maxWalkSAT 算法求解最大后验估计状态
MaxWalkSAT();
//在 maxWalkSAT 算法返回的状态上再进行吉布斯采样, 然后对产生的所有样本进行重新评估
Gibbs();
//运用 L-BFGS 算法对求解的参数进行优化
lbfgsOptimize();
//保存最终训练的模型
saved();
```

图 11-3

1. 如图 11-3 所示, 首先根据逻辑谓词的条件对基础数据进行特征提取, 然后利用谓词函数对于各个特征进行计算, 获得其谓词的值。然后根据数据和谓词的对应关系, 构建马尔科夫网络。
2. 获得网络之后, 就根据历史数据进行特征迭代学习, 这里使用 MaxWalkSAT 算法来计算网络的最大后验估计 (MAP) 的状态值, 进而利用这个状态值去求解最终的参数值。但是因为 MaxWalkSAT 算法并不能保证寻找到全局的 MAP 状态。因此, 为了提高估算精度, 通常在 MaxWalkSAT 算法返回的状态上再进行吉布斯采样, 然后对产生的所有样本进行重新评估 (实际上大多取均值)。
3. 也可以将对原子的最大后验估计 (MAP) 状态转换成由算法产生的一部分拟合分布的数据来估计 MAP 状态。常用的产生算法为蒙特卡罗抽样算法。这样就可以产生一些更加接近真实分布的均匀的样本, 并且可以产生更多有用的信息, 所以也更有可能会收敛到真实的期望。这就是 Contrastive Divergence (CD) 算法的主要原理。

## 11.4 算法应用扩展

马尔可夫逻辑网络 (MLNs) 是一种在有限领域将一阶逻辑和概率结合起来的简单方法。一个马尔可夫逻辑网通过一阶逻辑知识库的规则 (或条款) 的权重来获得, 可看作构建一

般马尔科夫网络的模板，知识库中每个可能的基本规则都会产生一个特性。目前马尔可夫逻辑网络已经广泛应用于自然语言处理、地理信息系统和计算机视觉等方面。

但是就像上面说的，马尔可夫逻辑网的理论还不成熟，还有很大的完善空间。对于它的研究有很多学者在不断地推荐，对于参数的求解和模型的表示是阻碍其应用的一个重大瓶颈，所以有兴趣的读者可以关注最新的研究进展，以在自己的项目中使用这个利器帮助自己提升工作效率和研究成果。

# 第 12 章

## DIPRE 算法

信息抽取是一个比较宽泛的领域，既包括实体的识别，还包括关系的提取。第 11 章介绍了使用马尔可夫逻辑网络来进行实体识别，本章介绍另一种方法：DIPRE (Dual Iterative Pattern Relation Extraction)，用它来进行关系的抽取。

## 12.1 算法应用原理介绍

DIPRE 算法的一个主要思想就是：从已知的种子中抽取出模式，再用得到的模式去对未知文本提取信息，然后从提取的信息中抽取新的模式，再运用到新的文本处理中，如此循环往复，最终实现滚雪球式的信息扩增。所以做这件事情最重要的前提是问题域本身有很多各种各样丰富的模式，如果只有单一的模式，那得到的结果也只有一个模式。

首先让我们来看一下整体的流程，运用 DIPRE 算法抽取关系的主要流程如图 12-1 所示。

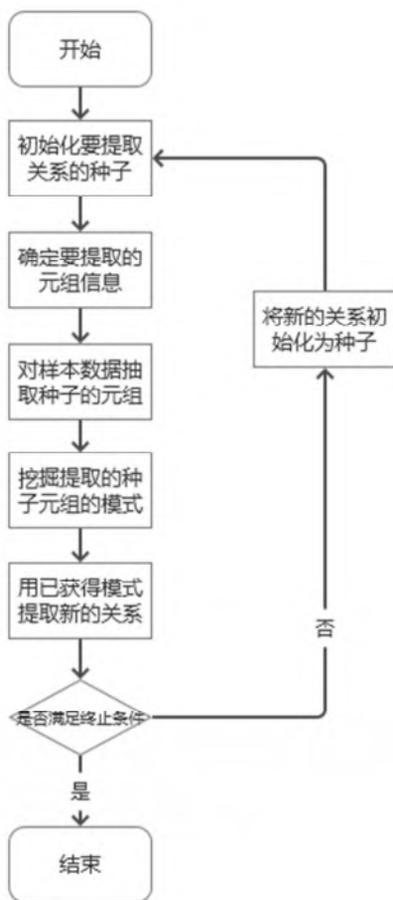


图 12-1

下面对具体流程进行说明和介绍:

1. 对于要提取的关系, 首先给定一些种子。比如我们希望对影响股票上涨的因素进行分析, 就希望从相关新闻资讯中提取股票与其对应的上涨因素关系。对于这个关系, 我们首先给定一组种子: (政策利好, 股票 A)。即政策利好和股票 A 满足所要提取的关系: 政策利好导致股票 A 股价上涨。
2. 在获得种子之后, 就要确定要抽取信息的元组信息。这一步主要是要确定我们要对文本提取什么信息, 能提取出固定的模式, 如果不能提取固定的模式, 那么我们就不能提取到新的关系, 这样信息的自增模式就不能进行了。所以这一步也可谓至关重要。如果事先确定元组信息, 对于模式的处理有一个好处就是能以固定的模式进行处理, 降低了计算的复杂度。当然这里面就要融入一定的专家知识, 要确定这种关系会和哪些特征有关, 并最终把这些特征抽象为一个元组。如果不想事先确定元组, 也可以采用其他的方法进行, 比如频繁模式挖掘, 最大公共子串等算法对包含种子的文本信息进行模式挖掘, 直接获得包含种子信息的模式。这种方法的好处就是不需要专家知识的介入, 但是缺点也很明显, 就是算法复杂度会比较高, 对于最终的结果无法保证一定能得到。里我们使用的是先人工确定元组, 再从文本中提取相应元组信息的方法。
3. 在确定元组信息之后, 就对输入的文本中包含种子的语句提取元组关系。假设我们确定了一个 6 元组信息: [order, entity1, entity2, prefix, suffix, middle]。

其中,

- a) order: 表示 entity1 和 entity2 在语句中的相对关系, 如果 entity1 在前则 order 为 1, 否则为 0。
  - b) entity1: 表示关系种子中的实体 1, 在这里即为: 政策利好。
  - c) entity2: 表示关系种子中的实体 2, 在这里即为: 股票 A。
  - d) prefix: 表示在 entity1 和 entity2 组成关系对前面紧相邻的语素。
  - e) suffix: 表示在 entity1 和 entity2 组成关系对后面紧相邻的语素。
  - f) middle: 表示连接 entity1 和 entity2 的语素。
4. 下面以两个例子对元组的信息抽取进行说明。
    - a) 输入语句 1: 因为出现政策利好导致股票 A 上涨 10%。那么其信息抽取的结果

如下。

6 元组: [order, entity1, entity2, prefix, suffix, middle] =

[1, 政策利好, 股票 A, 因为出现, 上涨 10%, 导致]

- i. 因为在语句 1 中, 政策利好出现在股票 A 的前面, 所以 order=1。
- ii. “因为出现”这个词, 是紧连接在包含“政策利好”、“股票 A”两个实体语句的前面的词, 所以也就是对应的 prefix。同样“上涨 10%”就为 suffix。
- iii. “政策利好”、“股票 A”两个实体之前的连接词为“导致”, 所以也就是对应的 middle。

b) 输入语句 2: 突然出现的政策利好消息导致股票 A 上涨 10%。那么其信息抽取的结果如下。

6 元组: [order, entity1, entity2, prefix, suffix, middle] =

[1, 政策利好, 股票 A, 突然出现的, 上涨 10%, 消息导致]

5. 获得了上述的元组列表之后, 我们就可对这些列表信息进行模式挖掘。具体步骤如下:

- a) 首先由于中文表达的复杂性, 要获得关键语素, 就要去除其中用于连接、停顿等无关紧要的词。因为实体部分是固定的, 所以不需要进行处理, 这一步只针对 prefix、suffix 和 middle 进行。
- b) 针对精简后的 6 元组进行模式挖掘, 这里首先按照 order 顺序对样本进行分类, 不同 order 的样本会放在一个集合中进行模式挖掘。这里采用 FP-Growth 算法进行, 要选择置信度较高的模式, 置信度的阈值可以根据自己情况进行设置。
- c) 针对上面的示例, 我们会得到如下的模式。

pattern1: [出现, entity1, 导致, entity2, 上涨 10%]

因为上述模式的后缀就是我们的关系 (对于数字可以进行忽略处理, 当然也可以将其变成离散值, 然后进行标注, 这里就不做扩展讨论了), 所以这个 pattern 的识别就很丰富了, 加入后缀为上涨, 就并入到当前关系的种子中, 如果为其其他值, 则可以根据具体的值定义一个新的关系, 并把结果当作新关系的种子进行学习。

d) 流程示意图如图 12-2 所示。

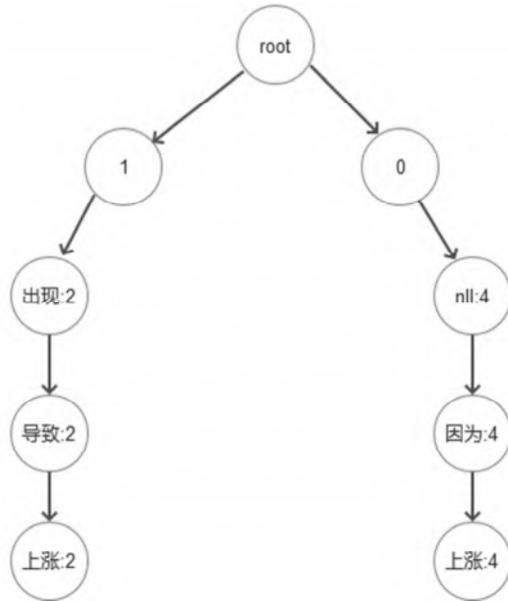


图 12-2

6. 用 `pattern1` 对其他文本进行关系提取，获得满足设定关系的其他种子数据，然后将新的种子数据放入到种子集合中，重复第 3~5 步，获得新种子的 `pattern`，……，逐步自增的扩大关系数据。
7. 当关系的数据达到终止条件，或者其他情况达到终止条件，比如没有待挖掘的数据了，那么就终止整个流程。

下面就对实现的数学原理进行说明。

## 12.2 算法数学原理介绍

DIPRE 算法是半监督关系提取中有代表性的一种方法。在这个基础上又发展出来了 SnowBall 算法。不过大致的流程基本一致，处理的逻辑如图 12-3 所示。

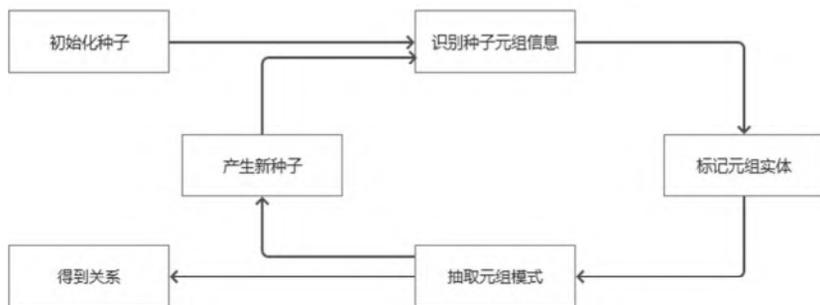


图 12-3

如图 12-3 所示，DIPRE 算法是一个计算框架，它本身并不涉及数学推理或者数学逻辑的证明。它定义了一个计算流程，在这个流程中，每个模块的计算会分别使用不同的算法来实现。对于同一模块也可以有不同的算法实现。

1. 对于元组的信息抽取，一般可以采用正则表达式，这种方法基本上是属于比较快速高效的方法。
2. 对于元组中实体进行识别，可以通过分词>实体标注来完成，而实体的标注既可以通过词性来筛选，也可以通过实体类别来实现。具体可以根据自己的情况来自由决定。
3. 元组模型的抽取，既可以利用本章介绍的模式挖掘算法完成，也可以采用规则的方式，比如对 prefix 取最大后缀公共子串进行，而对 suffix 则取最大前缀公共子串进行。
4. 对于种子的筛选，这里采用的是根据模式的置信度来进行筛选的，当然也可以根据人工或者图关系挖掘的方式来进行改进和更新。

对于关系的抽取会因为场景不同和语料的不同，可用或难或易的方法进行处理，最重要的是选择适合自己的方法，复杂的方法并不一定是最好的!!! 下面就对其计算流程代码进行示例说明。

## 12.3 算法源码说明

实现 DIPRE 模型算法的部分代码如图 12-4 所示：

```
//对基础数据按照正则表达式提取信息
List<String> data = regex(d, regexPattern);
//根据预先定义的元组结构,对实例数据提取元组信息
List<Tuple> t = buildTuple(F, data);
//构建 FP-Tree
FPTree tree = buildFPTrees(t);
//获取满足指定置信度的模式串
List<Pattern> p = getPattern(tree, confidence);
//对基础数据提取新的关系
List<Relation> r = regex(d,p);
//将新获得的实体加入到已有的实体集合中进行新的学习
entityList.add(r.getEntity());
```

图 12-4

1. 如图 12-4 所示,首先根据给定的实体,对样本数据进行信息检查,检查到包含实体的语句之后,就对其提取指定的元组信息。
2. 获得元组信息之后,就对元组信息进行频繁的模式挖掘,最终得到满足已知关系的,新发现的实体对。并把新的实体对加入到已有的实体集合中进行学习。
3. 整个算法的精度主要取决于最开始设定的元组信息,也就是特征的选择。如果特征选择得当,最终得到的效果就比较好。这也是为什么有一部分研究,主要把精力放在对特征的构建上。不过随着深度学习逐渐深入的应用,这种情况有望得到改观,运用深度学习实现特征的自动发现和提取。但是因为深度学习需要大量的标注信息,这又会是一个瓶颈,所以孰优孰劣,要根据具体的情况而定。

## 12.4 算法应用扩展

DIPRE 算法是为了解决在关系发现中需要大量标注样本而提出的一种半监督的学习算法。在此基础之上发展而来的 SnowBall 等半监督算法,在关系抽取的不同细分领域都取得了较好的效果。也都实现了可以在线调用的工具,以方便实验者进行测试和使用。

中文的关系抽取还是一个比较新的领域,或者说是成果不是特别丰富的领域,所以其适用的算法还是有很大研究空间的。在关系抽取中还会使用一些监督学习的算法,比如 SVM 算法,这类算法就和其他分类问题是同一个逻辑:首先对样本数据提取特征,然后根

据样本数据学习出一个分割的超平面，再用该超平面对测试数据进行分类，以获得测试数据中是否包含满足已知关系的实体对。另外还有基于高阶逻辑推理的方法，但是因为高阶逻辑推理在工程实现上难度较大，所以使用的场景有限。

# 第 13 章

## LSTM 算法

第 12 章我们介绍了信息抽取的一些算法，本章我们将介绍用于文本预测的一个算法：LSTM（Long Short Term Memory）算法。LSTM 算法是递归神经网络的一种，它能实现对历史信息的长期记忆。这对于需要利用跨度很长的历史信息来进行预测的场景非常重要，也克服了 N-Gram 等基于临近窗口信息进行预测算法的缺点。

## 13.1 算法应用原理介绍

LSTM 是由 Hochreiter & Schmidhuber (1997) 提出的, 并在近期被 Alex Graves 进行了改良和推广。在很多问题上, LSTM 都取得了相当巨大的成功, 并得到了广泛的使用。下面就对其在文本预测上的应用进行说明和介绍。这里我们以预测下一个词为例进行说明。

图 13-1 为以“英雄”为输入, 用 LSTM 网络预测其下一个单词的流程示意图:

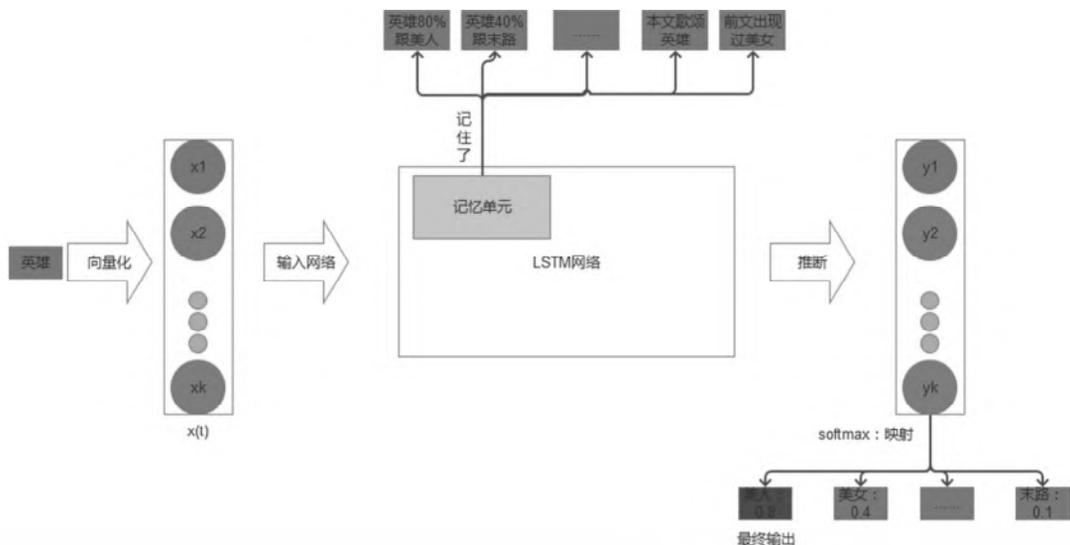


图 13-1

1. 如图 13-1 所示, 对于输入的信息我们首先需要向量化, 以便能够利用神经网络隐含层强大的特征扩充和筛选能力, 自动实现特征筛选。对于词的向量化方法, 之前已经有所介绍, 这里就不再赘述了, 这里采用的是 Word2Vec 的方式进行词的向量化。
2. 在得到输入词的向量之后, 就对输入的文本进行序列化。序列化的过程也就是选择输入的窗口, 对完整的文本按照窗口的长度划分为不同的计算单元。这里选择 1 个词作为一个输入单元。具体长度可以根据自己情况而定。
3. 确定了输入单元, 下面就是利用 LSTM 记忆单元, 将已经学习到的信息来预测下一个单词。比如在图 13-1 记忆单元学习到了:

a) 英雄后面跟不同词的概率，如表 13-1 所示。

表 13-1

“英雄”后面跟的词	概率
美人	0.8
美女	0.4
.....	.....
末路	0.1

b) 距离当前输入信息很久之前的信息，如表 13-2 所示。

表 13-2

Long Term-1	本文歌颂英雄
Long Term-2	前文出现过美女
.....	.....
Long Term-n	被抓

c) 距离当前输入信息较近的信息，如表 13-3 所示。

表 13-3

Short Term-1	美女
Short Term-2	上一个词是名词
.....	.....
Short Term-m	前面是一个陈述词

d) 有了这些记忆的信息，就可以根据当前的输入推断下一个词，但是因为输入的是向量，所以推断出的结果也是一个向量。下面就要将向量转化成最终的结果：单词。

4. 经过上述步骤，我们就可以根据当前输入，预测下一个词向量，得到词向量之后，就可以用 softmax 函数，得到输出结果映射到不同单词的概率，最终选择最大概率的单词作为输出即可。
5. 对于输出的预测，我们也可以借鉴隐马尔科夫模型，对每一个节点都输出多个预测，待整个链路预测完成以后，再按照最大概率的路径作为最终的输出，这样就可以避免因为某一个环节预测错误，而导致整个结果预测错误。流程如图 13-2 所示。

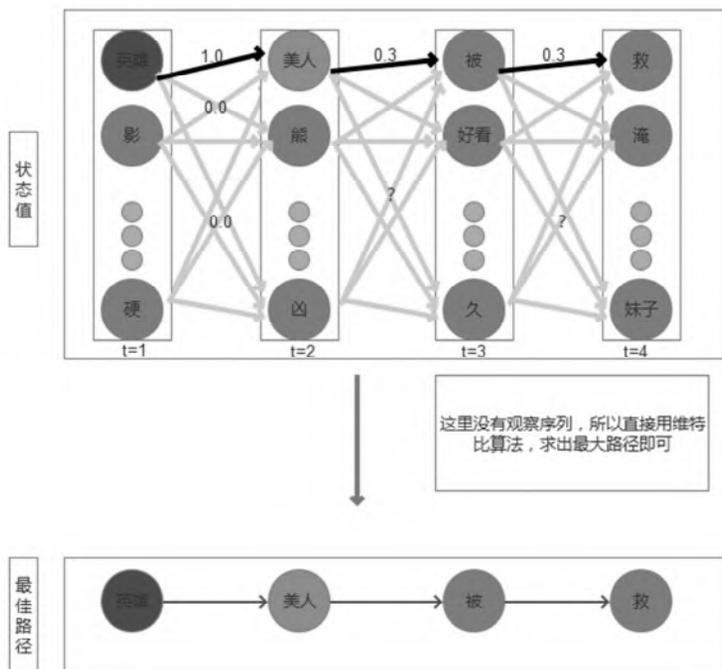


图 13-2

- 首先对于每一步的预测值，不再只是保留概率最大的那个预测单词，而是保留前  $n$  个概率较大的预测单词。
  - 把这些单词根据前后关系组成一个水晶网格，即建立前后连接的两步之间各个词的连接矩阵。
  - 然后用维特比算法，求出最佳路径，也即概率最大的路径，作为最终的输出。
6. 下面就对其数学原理进行说明和介绍。

## 13.2 算法数学原理介绍

LSTM 是 RNN 的一种类型，所以我们还是由 RNN 引出 LSTM。所有 RNN 都具有一种重复神经网络模块的链式的形式。在标准的 RNN 中，这个重复的模块只有一个非常简单的结构，例如只是将  $t-1$  时刻隐含层的变量值合并到  $t$  时刻的隐含层变量中，如图 13-3 所示。

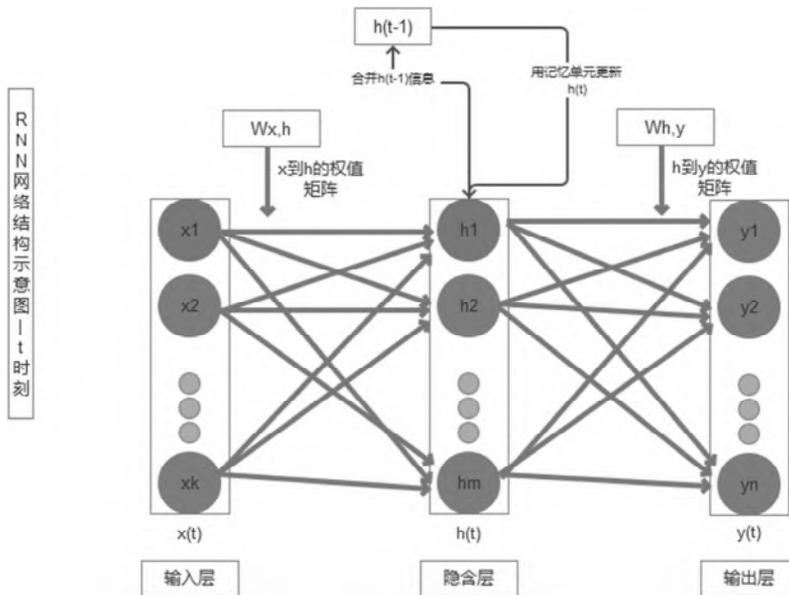


图 13-3

相较于标准的 RNN，LSTM 在重复模块拥有一个不同的结构。同时对于这个模块命名为记忆单元。这个模块根据不同的类型，设计也不一样。

1. 首先我们来看一下单隐含层 LSTM 模型整体的计算流程。如图 13-4 所示，LSTM 本质上还是一个多层感知机的结构，由输入层、隐含层、输出层三层结构所组成。相较于标准的多层感知机，LSTM 多了一个记忆单元的结果。下面就对其实现流程进行说明和介绍。
2. 确定了每一次网络计算的输入之后，就按照前向网络流程计算预测输出值。也就是按照图 13-4 的流程进行处理。
  - a) 首先由输入变量  $X$  乘以输入层到隐含层的特征矩阵，得到  $t$  时刻隐含层的隐含层向量  $h'_t$ 。因为这里不是最终的隐含层向量，所以加了一个“'”加以区别。

$$h'_t = X \cdot W_{x,h}$$

- b) 将  $h'_t$  和上一时刻的隐含层向量进行合并，得到记忆单元的输入向量  $h_c$ 。（这里为了形式的统一，用一个共享的参数  $W_x$  和  $W_h$  来进行遗忘门、输入门、输出门的计算，实际上可以采用不同的权值矩阵来进行。）

$$h_c = W_x h'_t + W_h h_{t-1}$$

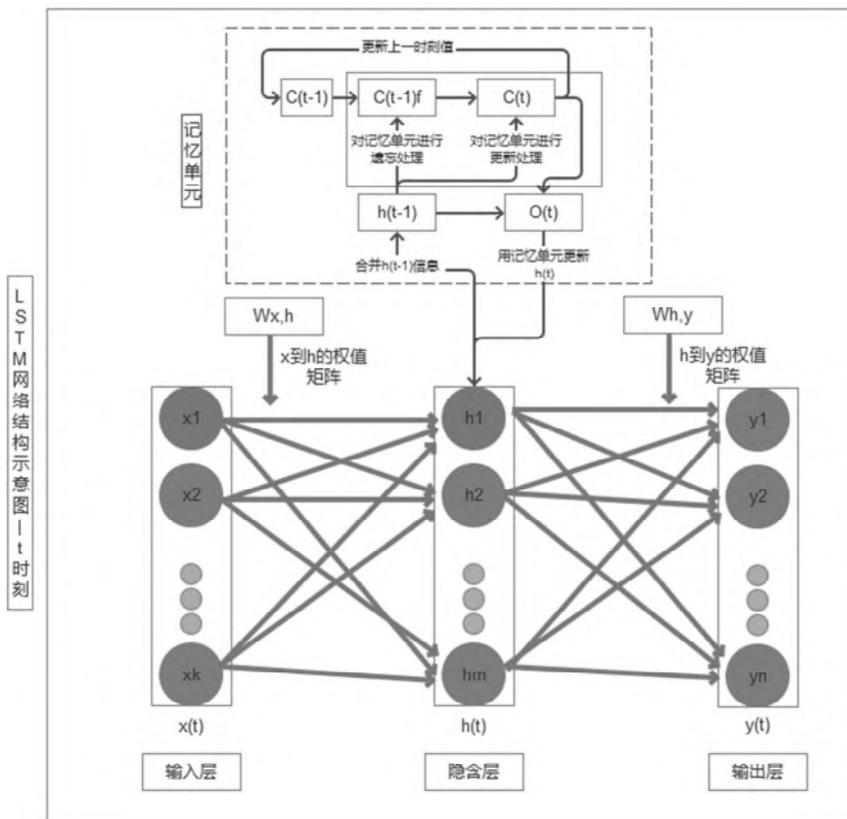


图 13-4

- c) 将  $h_c = (h_{c1}, h_{c2}, \dots, h_{cm})$  用 sigmoid 函数进行激活, 得到遗忘门权值向量  $f_t$ 。再用激活后的向量与  $t-1$  时刻的记忆单元值进行内积操作, 得到经过遗忘门操作的记忆单元向量  $C_{t-1}^f$ 。

$$C_{t-1}^f = C_{t-1} * f_t = C_{t-1} \cdot \text{sigmoid}(h_c)$$

- d) 将  $h_c$  用 sigmoid 函数进行激活, 得到更新门的权值向量  $u_t$ 。再将  $h_c$  用 tanh 函数进行激活, 得到记忆单元准更新向量  $\tilde{C}_t$ 。将两者做内积得到记忆单元更新的值向量  $C_t^u$ 。

$$C_t^u = u_t * \tilde{C}_t = \text{sigmoid}(h_c) \cdot \tanh(h_c)$$

- e) 将  $C_t^u$  和  $C_{t-1}^f$  相加得到  $t$  时刻的记忆单元向量  $C_t$

$$C_t = C_{t-1}^f + C_t^u = C_{t-1} \cdot f_t + u_t \cdot C_t'$$

- f) 将  $h_c$  用 sigmoid 函数进行激活, 得到输出门 (用于更新  $h'_t$ ) 的权值向量  $o_t$ 。再将  $C_t$  用 tanh 函数进行激活, 得到记忆单元的作用向量  $C_t^o$ 。将两者做内积得到新的隐含层向量  $h_t$ 。

$$h_t = o_t \cdot C_t^o = \text{sigmoid}(h_c) \cdot \tanh(C_t)$$

- g) 最后用新的隐含层向量乘以隐含层到输出层的权值矩阵, 得到输出向量。

$$y_t = h_t \cdot W_{h,y}$$

- h) 多说一点, 细心的读者会发现: 上面对于权值向量的激活, 都用 sigmoid 函数, 对于更新值的激活都是用 tanh 函数。这是因为在多次求导之后, sigmoid 会涉及梯度消失, 即对于变量值的更新, 不再起作用了。而 tanh 函数不会出现这个问题, 所以对权值在很久之前的, 可以认为近似等于其临近的值, 而对于计算值则要一直都能更新, 才能体现长期记忆的特性。
3. 以图 13-4 所介绍的结构为例, 有两个变量对应着名称的两个特征: long term 和 short term 记忆。那就是  $h_t$  和  $C_t \cdot C_t$  负责长期记忆,  $h_t$  负责短期记忆。
  4. 如果是用网络进行预测为例, 那么完成前向网络的计算就可以了。但是如果是训练阶段, 完成了预测, 那么还要根据预测结果来调整参数。特别强调一下: 根据误差, 调整参数。所以整个反向网络的核心就是调整参数, 整个计算过程中涉及如下几个参数。
    - a) 输入值:  $x_t$ , 这个调整称为将误差项传递到上一层。
    - b) 上一时刻隐变量:  $h_{t-1}$ , 这个调整称为将误差项传递到上一时刻。
    - c) 输入值的计算权值:  $W_x$ , 隐变量的计算权值:  $W_h$ , 这个调整称为权重的梯度调整。

所以整个反向网络的计算过程就是对上面 4 个输入参数进行更新。
  5. 按照标准的描述, 接下来会有一大波的公式来袭。但这不符合本书的风格, 所以这里会将一大波公式简化为: 用文字描述为啥要这么干。在进行介绍之前, 先给出几个即将用的导数结果。
    - a) 常数  $c$  的导数为 0。
    - b)  $cx$  对  $x$  的一阶导数为  $c$  ( $c$  为常数,  $x$  为变量)。

- c)  $y$  是  $g$  的函数,  $g$  是  $x$  的函数, 那么  $y$  对  $x$  的一阶偏导数等于  $y$  对  $g$  的一阶偏导数乘以  $g$  对  $x$  的一阶偏导数。

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial g} \cdot \frac{\partial g}{\partial x}$$

- d)  $y$  是  $g$  和  $x$  乘积的函数, 且  $g$  与  $x$  无关, 那么  $y$  对  $x$  的一阶偏导数等于将  $g$  视为常数时, 由对  $x$  求一阶偏导数。

$$\frac{\partial y}{\partial x} = \frac{\partial(g \cdot x)}{\partial g} = g \frac{\partial x}{\partial x}$$

- e) sigmoid 函数的导数等于其自身与 1 减去其自身的乘积。

$$\text{sigmoid}(x) = y = \frac{1}{1 + e^{-x}}$$

$$\text{sigmoid}'(x) = y(1 - y)$$

- f) tanh 函数的导数等于 1 减去其自身的平方。

$$\tanh(x) = y = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$\tanh'(x) = 1 - y^2$$

6. 因为误差是由  $y_t$  产生的, 所以反向传播的源头要由它开始, 以对  $h_{t-1}$  更新为例, 首先  $y_t$  不是  $h_{t-1}$  的直接函数, 所以要通过和  $h_{t-1}$  有直接联系的函数入手、发现  $f_t, i_t, o_t, \tilde{C}_t$  是  $h_{t-1}$  的直接函数, 而它们又是通过  $h_t$  与  $y_t$  产生联系的, 所以整个流程如下:

$$\frac{\partial y_t}{\partial h_{t-1}} = \frac{\partial y_t}{\partial h_t} \cdot \left( \frac{\partial h_t}{\partial f_t} \cdot \frac{\partial f_t}{\partial h_{t-1}} + \frac{\partial h_t}{\partial i_t} \cdot \frac{\partial i_t}{\partial h_{t-1}} + \frac{\partial h_t}{\partial o_t} \cdot \frac{\partial o_t}{\partial h_{t-1}} + \frac{\partial h_t}{\partial \tilde{C}_t} \cdot \frac{\partial \tilde{C}_t}{\partial h_{t-1}} \right) \quad (1)$$

- a) 而  $f_t, i_t, o_t, \tilde{C}_t$  与  $h_{t-1}$  的关系都是一样的, 都是由一个常数矩阵乘以  $h_{t-1}$  得到。所以以这四项对  $h_{t-1}$  求导之后, 就剩下常数矩阵  $W_h$  了。

- b)  $f_t, i_t, o_t, \tilde{C}_t$  到  $h_t$  的转化时遵从如下规则: 如果是权值则用 sigmoid 函数激活, 如果是输出值则用 tanh 函数激活。那么只要对它们各自的值用固定的公式计算即可。以  $\frac{\partial h_t}{\partial o_t}$  为例进行说明。

$$\frac{\partial h_t}{\partial o_t} = \frac{\partial(o_t \cdot C_t^o)}{\partial o_t} = C_t^o \frac{\partial o_t}{\partial o_t} = C_t^o o_t (1 - o_t)$$

c) 逐次完成各个步骤的求导就可以得到最终的值为:

$$\begin{aligned}\frac{\partial y_t}{\partial h_{t-1}} &= W_h \cdot (\delta_{o,t}^T + \delta_{f,t}^T + \delta_{i,t}^T + \delta_{\tilde{c},t}^T) \\ \delta_{o,t}^T &= \delta_t^T C_t^o o_t (1 - o_t) \\ \delta_{f,t}^T &= \delta_t^T o_t (1 - \tanh^2(C_t)) C_{t-1} f_t (1 - f_t) \\ \delta_{i,t}^T &= \delta_t^T o_t (1 - \tanh^2(C_t)) \tilde{C}_t i_t (1 - i_t) \\ \delta_{\tilde{c},t}^T &= \delta_t^T o_t (1 - \tanh^2(C_t)) i_t i_t (1 - \tilde{C}_t^2) \\ \delta_t^T &= y_t (1 - y_t) \cdot W_{h,y} \cdot \text{Error}\end{aligned}$$

其中, Error 为预测误差。

7. 按照同样的原理, 完成另外两个参数的更新, 即上一层误差传递和权值矩阵的更新, 就完成了整个学习的过程。
8. 这样整个网络的学习就完成了, 而学习的参数就作为后续预测部分的输入, 只需要用这些参数对待预测的输入信息进行计算, 就可以预测出其序列的相关信息了。

## 13.3 算法源码说明

实现 LSTM 模型算法的部分代码如图 13-5 所示。

```
//定义一个 LSTM 网络
LSTMGenerator gen = new LSTMGenerator();
/*对网络结构进行定义*/
//定义输入层单元格个数
gen.inputLayer(in);
//定义隐含层单元格个数、3 个门的激活函数类型、是否为双向网络
gen.hiddenLayer(hidden, CellType.SIGMOID, CellType.TANH, CellType.TANH,
true);
//定义输出层的单元格个数、激活函数等
gen.outputLayer(out, CellType.SIGMOID, true, -1.0);
//生成网络
Net net = gen.generate();
//设置最大序列长度
net.rebuffer(length);
```

图 13-5

```
//读入样本数据
SampleSet dataSampleSet = readData();
//设置训练数据集的大小
int trainSplit = (int) (dataSampleSet.size()*0.8);
//分割数据集，将数据集分割成训练集和测试集
//获得训练集
SampleSet trainset = dataSampleSet.split(trainSplit, rnd);
dataSampleSet.removeAll(trainset);
//获得测试集
SampleSet testset = dataSampleSet;
//生成训练器
GradientDescent trainer = new GradientDescent();
trainer.setNet(net);
trainer.setRnd(rnd);
trainer.setPermute(true);
trainer.setTrainingSet(trainset);
trainer.setLearningRate(learningrate);
trainer.setMomentum(momentum);
trainer.setEpochs(epochs);
//对数据进行训练
trainer.train();
//对数据进行测试
{
    RegressionValidator v = new RegressionValidator(net, thres);
    for (Sample s : testset) {v.apply(s);
    }
    System.out.println("validation set result: " + (v.ratio() * 100) + "%.");
}
```

图 13-5 (续)

1. 首先定义一个 LSTM 网络，然后对网络的结构进行预先定义，包括：输入层、隐含层、输出层的单元个数、隐含层各个门的激活函数类型、是否采用反向传播策略，等等。
2. 对网络结构定义完成之后，就用训练数据对网络进行学习和测试，这里对于网络的优化，采用的是梯度下降的优化方法，对于数据测试，用的是回归预测的策略，因为我们的输出是一个数值，所以要看具体的数值误差是不是在预先设定的范围内，如果在则都认为正确。

3. 最终是用测试集对于模型的准确度进行测试。这一部分直接用的是看预测值和真实值之间的差距，而没有用其他调整策略。
4. 各个门的权重和计算入参是优化 LSTM 效果的关键，所以对于模型的调优主要集中在这一块，针对模型的优化，可以在这一块下功夫。

## 13.4 算法应用扩展

LSTMs 模型作为 RNN 家族中的一员，以其在序列化数据处理中的良好表现，被广泛地应用于语音识别、错词纠正等自然语言处理领域。

随着研究的不断深入，LSTMs 家族的成员还在不断扩充，比如如果让每一次迭代都可以看到结果自身，那么就形成了 `peephole LSTM`。当然每一种 LSTM 都有自己的长处和不足，对于不同的场景，不同的网络会发挥不同的功效。所以还是那句老话：最好的未必适合自己，而适合自己的那款，永远是最好的。

## 第 14 章

# TransE 算法

在之前的章节，我们分别介绍了使用传统机器学习算法和使用深度学习算法来处理自然语言相关的问题。这两种类型的方法，各有千秋，在不同的场景下也会发挥不同的作用。不过这两类方法有一个共同的特点：都是通过探索数据本身蕴含的规律来指导或者预测未来的情况。即利用信息进行预测推断。

在实际生活中，我们除了利用信息进行判断以外，还会利用知识进行判断。这是非常重要的一个方面。比如我们要预测一条抛物线，如果有抛物线几何学的知识，那么我们只要得到物体运动过程中的 3 个点就可以预测出完整的路径了，但是如果让机器仅仅依赖信息进行描述，那么可能要成千上万的路径分割图片供学习，才能获得最终的路径。

对于知识的工程表示，知识图谱是一种有效的方式。随着知识图谱处理技术的不断更新，开源图谱的逐渐丰富，知识图谱在自然语言处理等领域中有了越来越大的应用价值。本节就介绍一种自动构建知识图谱的算法：TransE 算法。

## 14.1 算法应用原理介绍

在介绍 TransE 算法之前，先简要地介绍一下知识图谱的信息，图 14-1 为知识图谱的一个示意图：

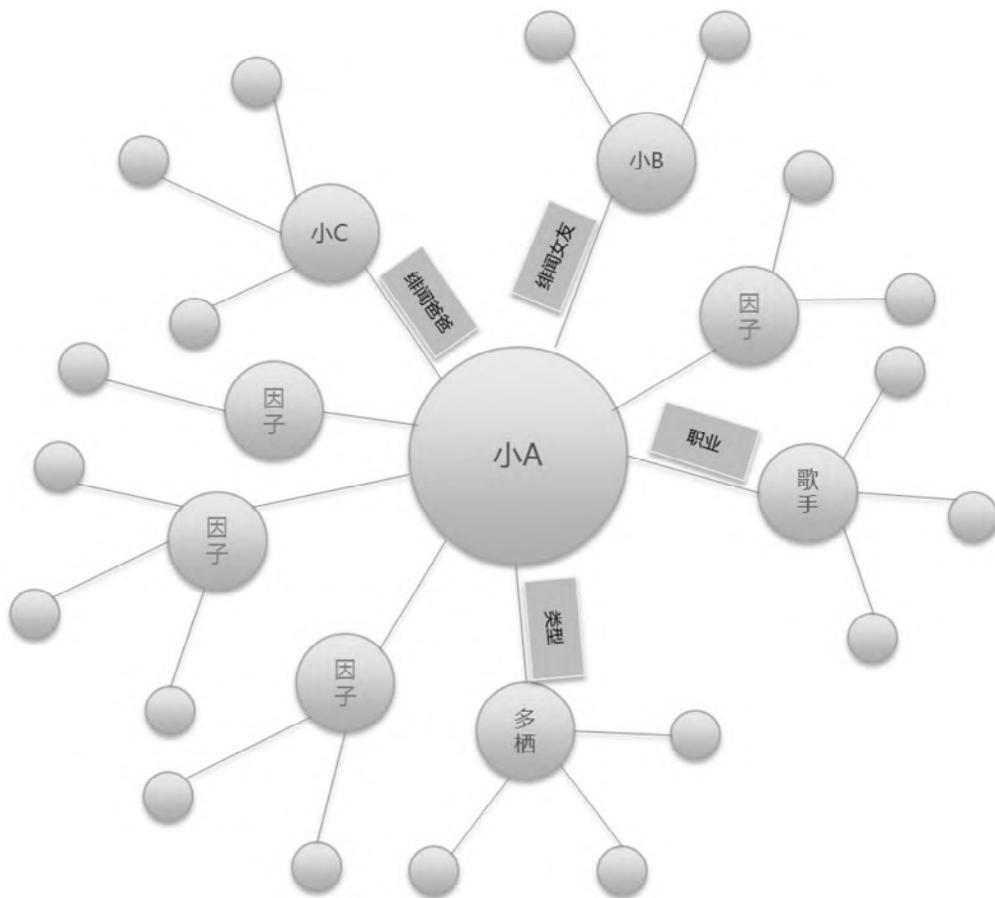


图 14-1

如图 14-1 所示，一条知识图谱可以表示为一个三元组 (subject,relation,object)。举个例子：小明的儿子是小亮，表示成三元组是 (小明, 爸爸, 小亮)。前者是主体，中间是关系，后者是客体。主体和客体统称为实体 (entity)。关系有一个属性，不可逆，也就是说主体和客体不能颠倒。

知识图谱的集合，链接起来成为一个图（graph），每个节点是一个实体，每条边是一个关系，或者说是一个事实（fact）。同时这个图是一个有向图，方向是从主体指向客体。比如，对于（小明，爸爸，小亮）这个关系，是从小明指向小亮，而不能反过来。

有了这样的图谱之后，我们能干什么呢，下面给出一个如图 14-2 所示的简单的示例。

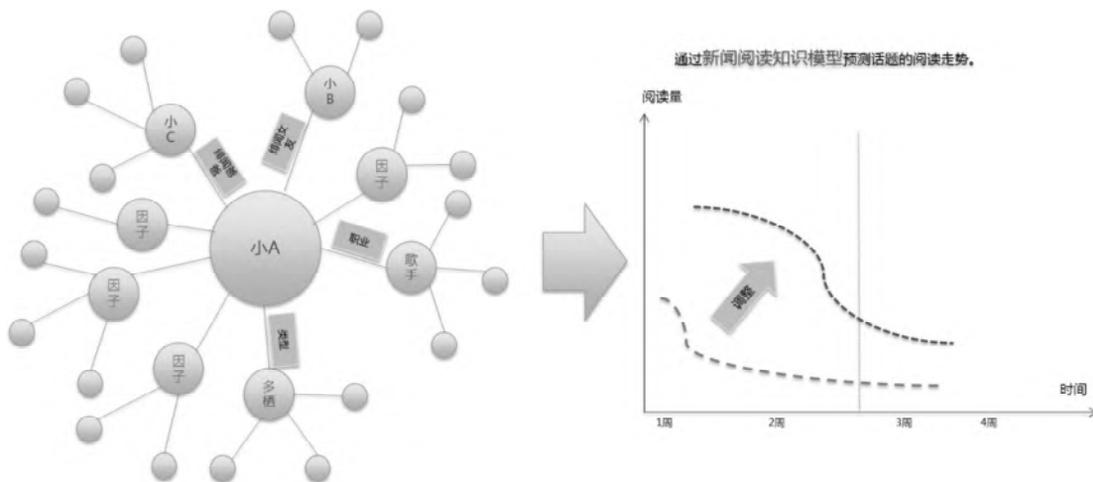


图 14-2

如图 14-2 所示，由左边的知识图谱结合新闻阅读的信息组合而成的新闻阅读知识模型，可以帮助我们找到将新闻阅读趋势由右边图的蓝线变成红线的方法。这样我们就可以实现对内容的智能推广，并且可以实现对内容影响力的有效把控。

这里要特别说明两点：

1. 上文所介绍的知识图谱的信息只是一个引子。完整的知识图谱是非常强大和复杂的。希望不要误导大家。
2. 本文所列举的图谱关系示例，只是为了做场景说明，并不代表真实发生的情况，如有雷同纯属巧合。

上面介绍了知识图谱的基础信息之后，那么接着就要说说图谱的构造了。因为图谱的内容是知识，具有一定的确定性，对于属性为概率的情况，也代表着概率是确定的。所以这就对图谱的构建提出了较高的要求。下面我们就看看怎么用 TransE 的算法进行图谱构建，如图 14-3 所示。

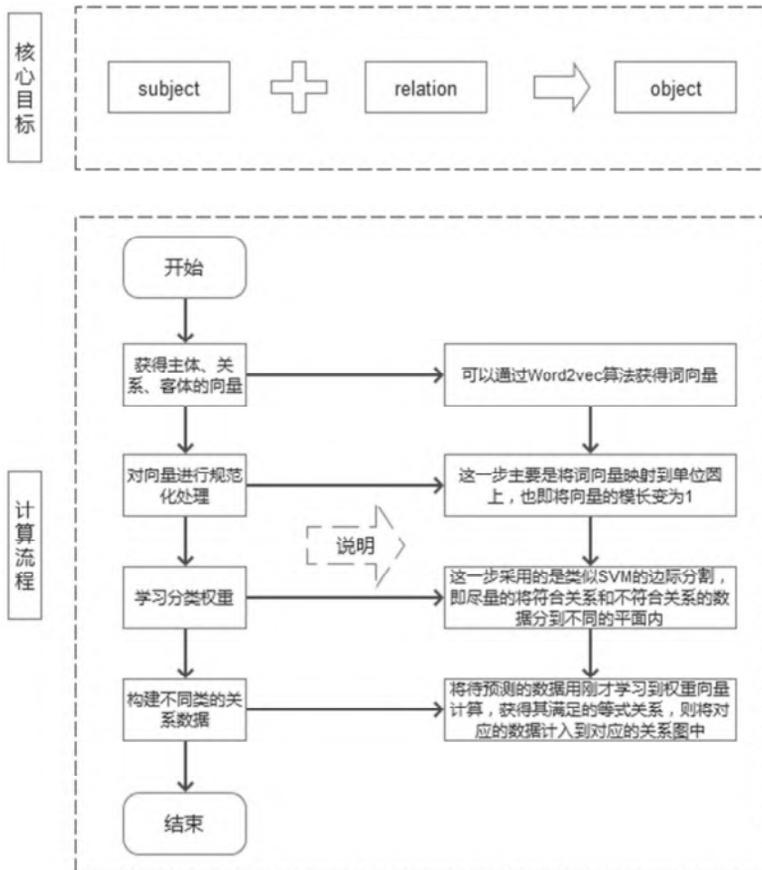


图 14-3

1. 如图 14-3 所示，TransE 算法的核心就是要找到一个向量生成模式，使得由一条知识图谱关系中的三个元素：主体、关系、客体，生成的向量满足图 14-3 中“核心目标”的关系。对于图 14-3 中的推出符号，可以改成等号，等号的信息则更强。本文的学习目标也是按照等式的方式来进行处理的。

$$\text{subject} + \text{relation} = \text{object} \quad (1)$$

2. 明确了算法的目标之后，就按照步骤来朝着目标前进。第一步就是获得词向量，因为在知识图谱中，主体、关系、客体都对应的是自然语言描述，所以这些变量都可以用词的向量化方式，获得对应的向量。这里采用的是 word2vec 的方式，对词进行向量化。当然也可以用之前章节介绍的其他向量化方法。

3. 获得向量之后，首先要做的是对向量进行规范化，即将向量映射到单位圆内，这样可以消除因为量纲不一致导致最终结果受影响。
4. 得到了输入向量之后，就要通过样本数据的学习建立分类模型，即构建公式 (1) 的等式。这里采用的是随机梯度方法进行参数的优化。之所以采用这种方式，一方面是因为参数本身是没有冲突的，也即不同值之间互不影响，另外这种方式还有利于实现分布式计算，提升计算效率。
5. 获得模型参数之后，就要对输入数据进行预测了，其实就是在验证输入的主体和关系条件下，最大可能映射到哪一个客体。当然也可以采用完全分类的特点，就是将三元组关系全部输入，评估最终的分类是否在 positive 的平面上。

下面对其数学原理进行说明和介绍。

## 14.2 算法数学原理介绍

1. 首先介绍一下，为什么要使用  $\text{subject} + \text{relation} = \text{object}$  这个等式来发掘主体和客体之间的关系。

a) 在介绍词向量化算法的时候，我们举过一个例子：

$$V(\text{queen}) - V(\text{woman}) = V(\text{king}) - V(\text{man}) \quad (2)$$

其中， $V()$ 表示各个词的词向量。

- b) 也就是说单词经过向量化之后，可以进行加减这样的线性操作，并且这个线性操作最终的结果是满足相关语义条件的。这就为我们的关系挖掘（知识图谱的核心）开辟了一个新的思路。
- c) 对于公式 (2) 我们进行如图 14-4 所示的一个操作，则可以将其转化为我们构建知识图谱时要整理的一个核心公式： $\text{subject} + \text{relation} = \text{object}$ 。

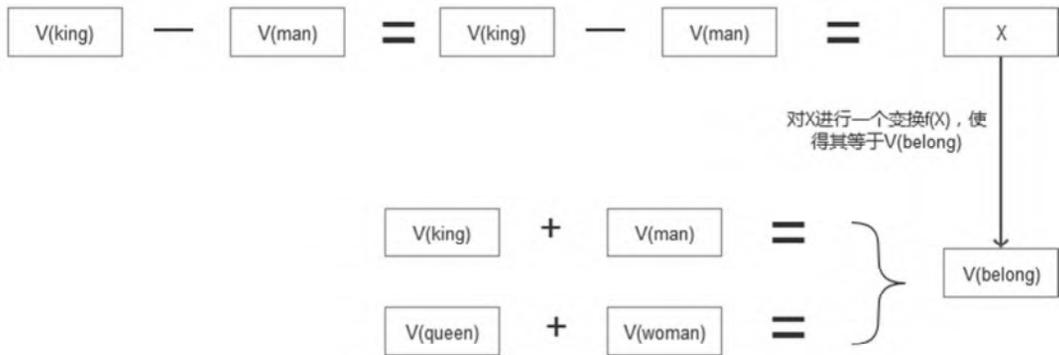


图 14-4

- d) 在图 14-4 中由  $X$  到  $V(\text{belong})$  的转换, 虽然看起来有点故意而为的嫌疑, 但是实际上它是真是存在的。具体步骤如下:
- i. 由词向量之间的加减运算, 将由同一个运算的结果相同的词向量作为一个集合。这里就将  $\langle \text{king}, \text{man} \rangle$  和  $\langle \text{queen}, \text{woman} \rangle$  作为一个集合  $\{\langle \text{king}, \text{man} \rangle, \langle \text{queen}, \text{woman} \rangle, \dots\}$ 。
  - ii. 得到这个集合之后, 首先遍历已知的知识, 判断是否存在对集合中元素关系的定义, 如果存在则将这个集合里面的所有元素都替换为已知的关系。如果集合里面的元素均不存在已知关系, 则结合人工或者是对已知文本内容的挖掘获得其关系。
  - iii. 获得关系词之后, 将关系词进行向量化, 然后建立一个映射关系即可。其实这一步, 在实际工程中也不是必须的。这里只是解释一下图 14-4 的映射并不是纯粹由人造出来的。
- e) 因为词向量之间有这样良好的线性操作性, 所以其对于我们构建知识图谱是一个非常利好的消息, 借助于 word2vec 这样的词向量化算法, 我们能够在大规模的数据集上进行关系挖掘和知识图谱的构建。而对于这样数据的构建, 就是建立

**subject + relation = object**

这样的等式即可。

2. 明确了目标，接下来就是要使得求解目标的方法更加准确，也就是求解目标函数的最优解。

a) 首先我们确立一个损失函数，基于目标我们指定，要使 subject 加上 relation 的值与 object 的距离为最小，用公式表述如下：

$$\text{distance}(\text{subject} + \text{relation} - \text{object})$$

b) 那么接下来我们就要定义一下这个距离的计算公式，这里采用计算其距离平方的方式进行衡量。即：

$$\text{distance}(\text{subject} + \text{relation} - \text{object}) = \|\text{subject} + \text{relation} - \text{object}\|_2^2$$

c) 由于这个模块的计算会存在一个问题，比如向量中某一个分量出现了异常情况，那么如果采用上面的公式进行优化，就会出现结果受异常值影响比较大的情况。借鉴 SVM 求解最优化的方式，我们这里对最终的模板加入两个变量，一个是 margin 值，另一个是松弛变量，松弛变量是通过对 subject 和 object 分别做一个微小的扰动得到的。所以最终的目标函数就变成如下的形式：

$$\begin{aligned} \text{distance}(\text{subject} + \text{relation} - \text{object}) \\ = \alpha + \|\text{subject} + \text{relation} - \text{object}\|_2^2 \\ + \|\text{subject}' + \text{relation} - \text{object}'\|_2^2 \end{aligned}$$

3. 得到了目标函数，接着就是对目标函数求最优解。这里采用的是随机梯度下降算法进行求解。求解的目标函数形式如下：

$$\nabla[\alpha + \|\text{subject} + \text{relation} - \text{object}\|_2^2 + \|\text{subject}' + \text{relation} - \text{object}'\|_2^2]$$

4. 完成了梯度的优化之后，就可以用最终得到的函数对输入值进行预测和求解了，下面我们就对其源码进行说明和介绍。

## 14.3 算法源码说明

实现 TransE 模型算法的部分代码如图 14-5 所示：

```
//首先获得实体 ID 列表
Map<Integer, String> entityList = getEntityId(entityFile);
//获得实体的词向量矩阵
double[][] entityMatrix = getEntityMatrix(entityList, model);
//获得关系词的 id 列表
Map<Integer, String> relationList = getEntityId(relationFile);
//获得关系的词向量矩阵
double[][] relationMatrix = getEntityMatrix(relationList, model);
//获得训练数据集合, 即 subject、relation、object 的关系对
List<String> tripleList = openTrain(trainFile);
//训练 TransE 算法, 并对最终的结果进行保存
TransE transE = new TransE(entityList, relationList, tripleList, margin,
dim);
transE.initialLize();
transE.train(iterations);
transE.save(outfile);
```

图 14-5

1. 首先获得实体词 (subject、object) 和关系词 (relation) 的码表。即文本和数字编号的对应关系。这个数字编号在后面取数和存数时使用。
2. 得到要处理的词列表之后, 就是对具体的词进行向量化处理。处理完成以后, 就根据各个词对应的编号, 将其向量存储到对应的行, 于是就得到了两个向量矩阵, 一个是用来记录实体的, 另一个是用来记录关系的。
3. 对于 TransE 的训练, 可以采用分布式进行, 因为采用的是随机梯度下降法, 即每一次只针对所取的部分数据进行训练。这样只要将所有的数据按照 minibatch 的数量进行分割, 然后便可分别分配到不同的机器上进行处理; 只是要将两个权值矩阵放在 master 的机器上, 一旦每一个 slaver 更新之后, 就可同步更新 master 的数据。
4. 当然对于模型的求解, 也可以根据自己的情况进行优化, 因为随机梯度下降, 虽然能保证最终收敛到最优值, 但是在每一次迭代的过程中, 并不能保证一定比前一次更优。所以可以根据自己的数据, 如果数据量不大, 也完全可以采用梯度下降, 即对所有的数据进行全局求解, 这样就能保证每一次求解一定比上次更优。

## 14.4 算法应用扩展

TranE 是由 Bordes 等人于 2013 年发表在 NIPS 上的一篇文章提出来的算法。它的提出，是为了解决多关系数据（multi-relational data）的处理问题。

TransE 是将  $(h,r,t)$  映射成低维度的向量，最终形成  $h+r \approx t$  的形式。但是对  $1$  到  $N$ ， $N$  到  $N$  的关系表示有缺陷。为了解决这个问题，就提出了 TransH 算法。它将  $r$ （margin 值）考虑成一个关系专用的超平面（hyperplane）的转换操作符，此时它的分割变量就变成了向量，那么映射关系就可以转换成超过 1 个，只要向量的长度大于 1 即可。

同时我们也提到了：由数据得到的关系其实具有一定的不确定性。但是 TransE, TransH 与 TransR 等方法中正例三元组与反例三元组的分割具有相同的边缘（margin），即把这个关系认为是一个确定的关系。针对这个缺点，有学者又提出了 GAUSSIAN EMBEDDING 算法。该算法不同于 TransE 的点映射方式（即将实体和关系映射为向量，通过最小化  $h+r-t$  得到向量表示），提出该算法的那篇文章将实体与关系映射为正态分布，通过 KL-divergence（KL 散度，又称为相对熵）来评价实体间的亲密程度。这样就可以根据一个关系对在样本数据中出现的频率和场景来确定关系的确定性：当一个实体在三元组中出现得越少，确定性越高，反之亦然。一个关系连接的实体越少，确定性越高，反之亦然。

目前对于知识图谱的构建技术还在不断发展中，随着知识图谱的基础设施和应用场景越来越广泛，其价值和重要性也会越来越大，深度学习和知识图谱的融合也成为一种趋势，所以对知识图谱的深入研究，有利于自然语言处理向更高阶的方向发展。

## 第 3 篇

# 系统案例实战

前两篇我们分别从单个场景的应用和基础算法的角度介绍了自然语言处理的相关知识。本篇我们将介绍如何利用上面的技术构建一个较为完成的处理系统，作为本书的结尾。



## 第 15 章

# 搭建舆情分析与挖掘的系统

在做事情之前，我们一般都会首先明确做这件事情的目的是什么，这样才能有的放矢。才知道哪些事我们要做，哪些事我们不要做。所以对于我们即将要搭建的系统，我们也首先给出这个系统的目的，以便确定它的边界范围。

如题所述本系统是一个舆情信息分析和挖掘的系统，所以主要集中在舆情信息的清洗、融合、抽取、分析、研判等环节。而对于基础数据的采集，以及产生结论之后的决策实施是不进行处理的，这些环节的功能和设计也不包含在本文要介绍的范围内。

## 15.1 系统功能设计简述

虽然本文主要介绍的是舆情信息分析和挖掘的系统，但是为了让大家对完整的处理系统有一个大概的了解，首先给出一个完整系统的架构图，如图 15-1 所示。

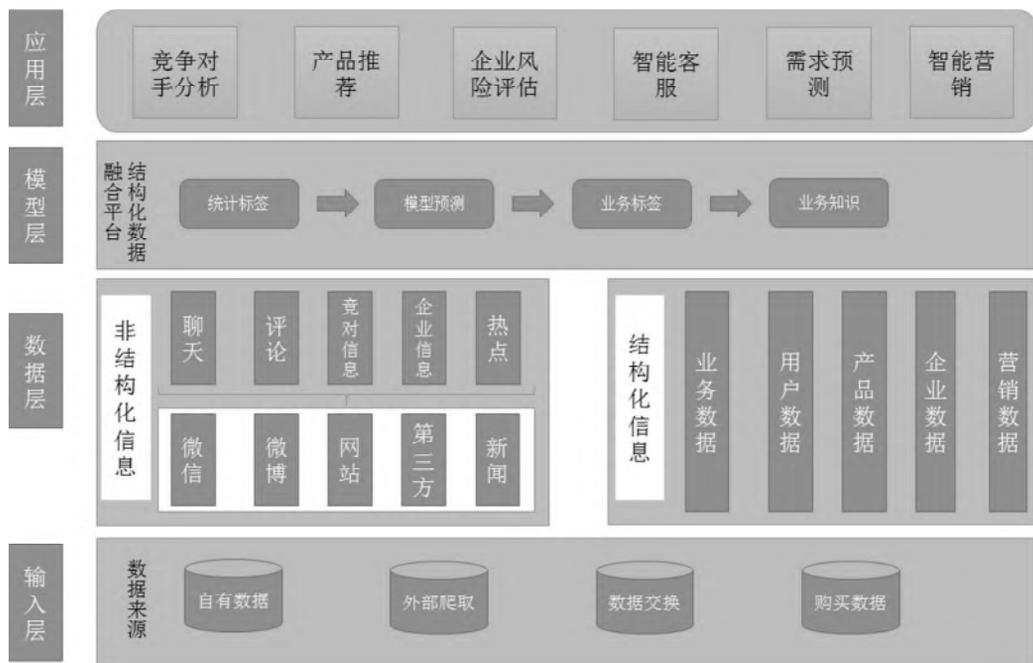


图 15-1

1. 如图 15-1 所示，完整的系统包含：数据输入、数据处理、数据输出等环节。数据输入对于舆情系统是非常重要的，因为信息是否全面、及时，决定了最终信息是否有价值。
2. 同时虽然我们主要是对舆情进行分析和挖掘，但是输入数据也不光是只有文本的信息就可以，还需要借助于其他的数据，比如产品数据、企业数据，才能知道当前说的内容是指向谁的，是否属于危机的管理范围，等等。
3. 最终对于信息的输出也是至关重要的，因为它直接决定了最终信息是否能够触达到相关方，也就是信息的价值是否能被发现和利用。这一步会因为业务方不同而有不同的形式，比如对于预警信息则以短信或者其他 IM 形式通知会比较合适。而对于

业务分析，则以分析报告的形式通知会比较合适。

上面我们介绍了一个舆情分析和挖掘系统大概的全貌，当然这里只是给出其中一些核心环节的架构，完整的系统架构要比这个复杂得多。因为这个不是本文的重点，所以就不做过多的说明。下面就对我们要介绍的分析和挖掘环节的细节进行说明和介绍。

1. 首先给出一个系统的挖掘部分的各个模块的架构图，如图 15-2 所示。

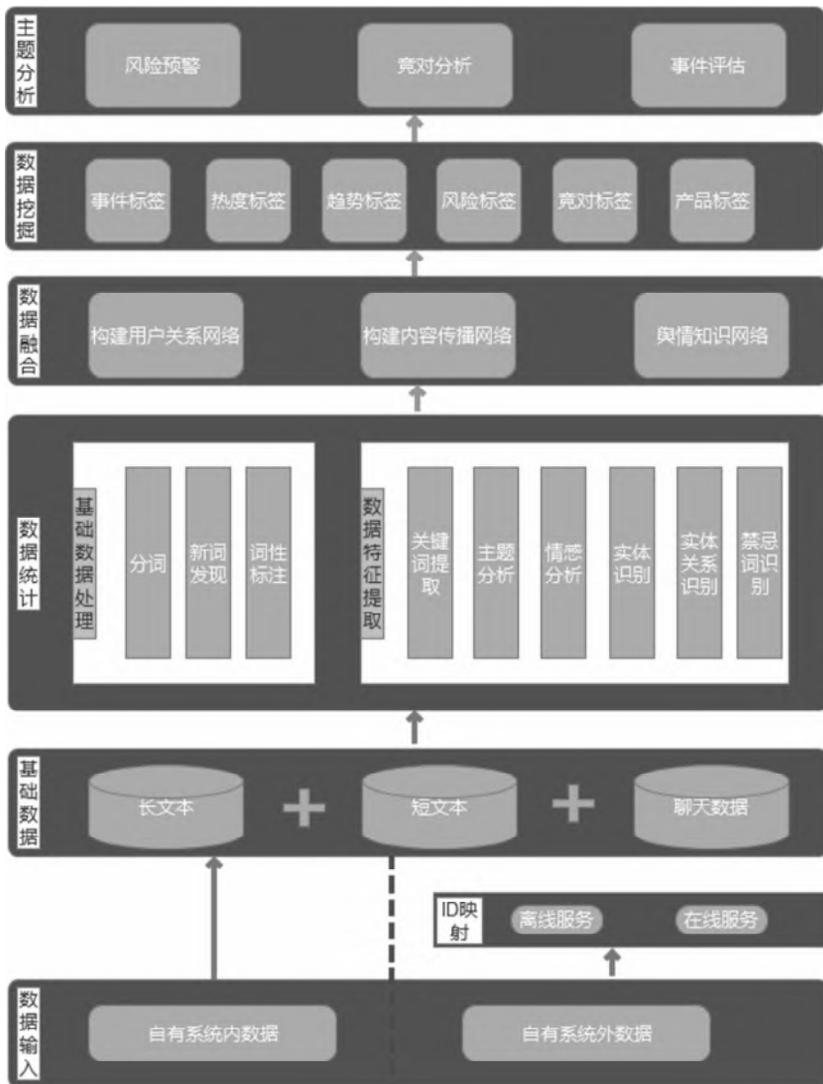


图 15-2

- 其次，因为数据会有多个来源，而不同来源的数据又有可能会指向同一个人、同一个事件，或者来自于同一个人。而在系统内部和外部对同一个人或者事件的标识一般会不一样，所以要对内外部的信息做一个 ID 映射，将同一个主体的数据合并在一起。
- 做完数据的 ID 合并之后，这里按照文本的类型对不同的数据做了一个区分，当然这个区分只是逻辑上的，而不是物理上的。在使用的时候，这三种类型的文本之间并不是完全割裂的，一般都会交叉使用，比如一个人可能在微博上对事件 A 做了一个简单的描述，然后在博客上对事件 A 做了一个详细的描述，这个时候对于事件 A 的分析，就要用到这两种类型文本的数据结合起来分析。短文本会包含更加明确的情感倾向的数据，而长文本会对事件的细节有更多的描述。
- 接下来就是对文本进行基础处理：分词、词性标注等，然后对文本进行聚类和信息抽取，精简主要内容，用于后续的挖掘。这个部分的先后顺序可以根据自己的场景灵活调整，如果所要求的场景只是想找到当前话题。那么只需要进行关键词提取或者是主题发现就可以，其他的都可以省略了。
- 获得了基础的信息之后，就要对数据进行组织了，这里按照内容发生的主体、对象、详情分别构造不同的关系网络，比如通过构造主体网络，我们可以知道在舆情传播的网络中，哪些节点是关键节点，就可以找到对某一类型事件，哪些人有绝对的影响力了。或者要分析的事件，哪些人起到了关键作用。同时对于整个信息的挖掘，还要借助于已知的舆情知识网络，因为这里面会包含很多专家的经验，比如：公司的禁忌、产品信息、公司最近发生的事件以及这些事件的主旨和期望达到的效果等。
- 完成这些网络数据的准备之后，就可以挖掘不同的标签，这些标签类型既有基于细分领域构建的，也有一些通用的标签，这样就组成了基础的标签体系。
- 这里列出了几个在舆情分析和挖掘场景下可能会应用的场景。风险评估，主要是根据先验知识来确定的，即舆情知识网络的内容，因为任何一件事情是否属于风险，要完全依赖于当前主体的特性。比如 A 航空发生了一件恶性事件，那么这个事件对于 B 航空公司来说就是一件利好的事情，而不是一件有风险的事情。

下面就对模型中数据统计和数据挖掘中的部分模块进行说明和介绍。

## 15.2 系统模块实现详解

### 1. 新词发现模块

在数据统计模块，我们的流程相较于标准的处理流程多了一个模块：新词发现模块，这是因为在实际处理中，我们发现很多网络词语的使用会导致挖掘结果误差比较大。但是因为这些网络词语一般都是根据突发事件产生的，比如“蓝瘦香菇”。所以在根据词典完成分词之后，要对文本再做一次新词发现，这样可以避免这种情况发生，当然这个效果的好坏是与新词发现算法有密切关系的。

- (1) 首先给出本文使用的新词发现算法的流程图，如图 15-3 所示。
- (2) 对待处理的文本集进行断句处理。对全部的文本，以句号为分割，获得以句子为单位的语句集合。
- (3) 对上面的语句集合进行编码操作。对每个语句按照字符为单位进行逐个拆分，并记录每个字符所属的语句编号  $i$ ，以及在语句中的位置编号  $j$ 。这样就得到了每个字符在给定的文本集合中的编码。
- (4) 确定最大重复子串。根据编码表，获得字符串左右邻的关系，确定重复的最大子串。并统计该字符串出现的次数。
- (5) 分拆最大重复子串。根据最大重复子串的长度、左右邻字符的信息熵来确定最大子串中各个字符应该包含在该子串中的概率。如果概率大于设定的阈值，则该字符应该包含在子串中，否则就把该字符从子串中剔除。
  - a) 如果字符串字符长度等于 1，则放弃；如果字符串长度等于 2，则直接添加到候选词库中。
  - b) 当字符长度大于等于 3 时，则计算第  $i$ ， $i=2, \dots, k-1$ ，字符左右邻字符的条件概率  $p_{左}$ 、 $p_{右}$ ，通过比较， $p_{左}/p_{右} < r$  或者  $p_{右}/p_{左} > (1/r)$ ， $r$  为设定的阈值。由满足阈值条件的连续字符串组成候选词库。
  - c)  $p_{左}$ =第  $i$  个字符左邻字符的信息熵； $p_{右}$ =第  $i$  个字符右邻字符的信息熵。
  - d) 第  $i$  个字符左（右）邻字符的信息熵 = 第  $i$  个字符的出现的概率 $\times \log_2$ (第  $i$  个字符的出现的概率)。

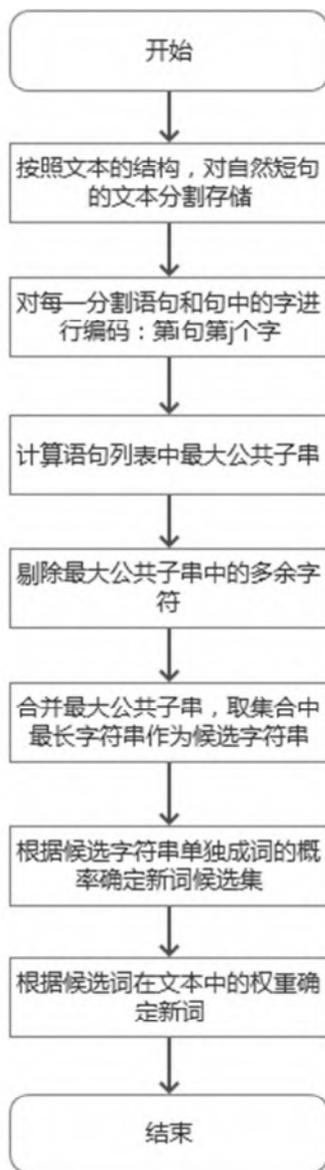


图 15-3

i. 第  $i$  个字符出现的概率 = (左边字符和第  $i$  字符同时出现的次数) / (左边字符单独出现的次数)。

(6) 合并最大重复子串。如果生成的字符串  $a$  被字符串  $b$  包含, 那么对  $b$  中剩余的字符串进行判别, 判断剩余的字符串是不是满足生成独立词的条件, 如果不满足,

则将 a 删除，否则则将 b 分拆。判断条件同第 (4) 步。

- (7) 对拆分的字符串进行新词发现。对 (4) 中剔除和 (5) 中分拆出来的字符集运用条件概率计算该字符串单独成词的概率，如果概率大于设定的阈值，则生成新的词。

$$a) p_i = C_{i-L} \cdot C_{i-R} / C_i \cdot Len_{total} / Len_i$$

- b)  $P_i$ : 表示第  $i$  词单独成词的概率;  $C_{i-L}$ : 表示第  $i$  词左边有字符的次数;  $C_{i-R}$  表示第  $i$  词右边有字符的次数;  $C_i$ : 表示第  $i$  词的次数;  $Len_{total}$ : 表示整个文本的字符个数;  $Len_i$ : 表示第  $i$  个词的字符个数。

- (8) 获得描述话题的主题关键词。计算各个最大重复子串、新词的权重。根据权重确定各个最大重复子串和新词对整个文本集的代表权重。根据权重的阈值和最大代表字符串的个数，选出最具代表性的  $topn$  字符串，作为描述话题的主题关键词。

$$a) p_i = C_i / C_d \cdot \log(D_{total} / D_i) \cdot \log_k \left( \sum_{i=0}^n \cos(i / 4^2 \cdot (Len_i - \theta)) - e^{|Len_i|} \right)$$

- b) 其中,  $p_i$  表示词  $i$  对整个文本集的代表权重;  $C_i$ : 表示词  $i$  在文档  $d$  中的个数;  $C_d$ : 表示文档  $d$  中词的总个数;  $D_{total}$ : 表示整个文档集中文档个数;  $D_i$ : 表示整个文档集中包含词  $i$  的文档个数;  $Len_i$ : 表示词  $i$  的字符长度。

- c)  $k, n, \theta$  为经验值。

- d) 其中词  $i$  表示的是字符长度大于 1 的字符，也即如果只有一个字符是不算作词的。

- (9) 新词获取。将上述结果形成的词集与现有词库进行比较，找出未登录的新词、短语。列入库中作为新词和短语存储。

完成了新词发现之后，就可以根据句法分析对词进行词性标注，获得当前词的词性，以便进行后续的情感判断等分析使用。

## 2. 热度计算模块

这里通过对用户的话题进行聚类，发现不同话题的用户簇，然后根据簇的大小决定话题的热度。所以这里的热度是一个相对的概念，如果某一段时间大家在网络上讨论的积极性都不高，但是我们的模型依然会选出前  $k$  个讨论比较热的话题。这样就可以对当下的舆

情有一个及时的关注和发现。

- (1) 按照惯例，我们首先还是画出一个流程图，让大家对整个计算流程有一个大概的了解。具体流程如图 15-4 所示。

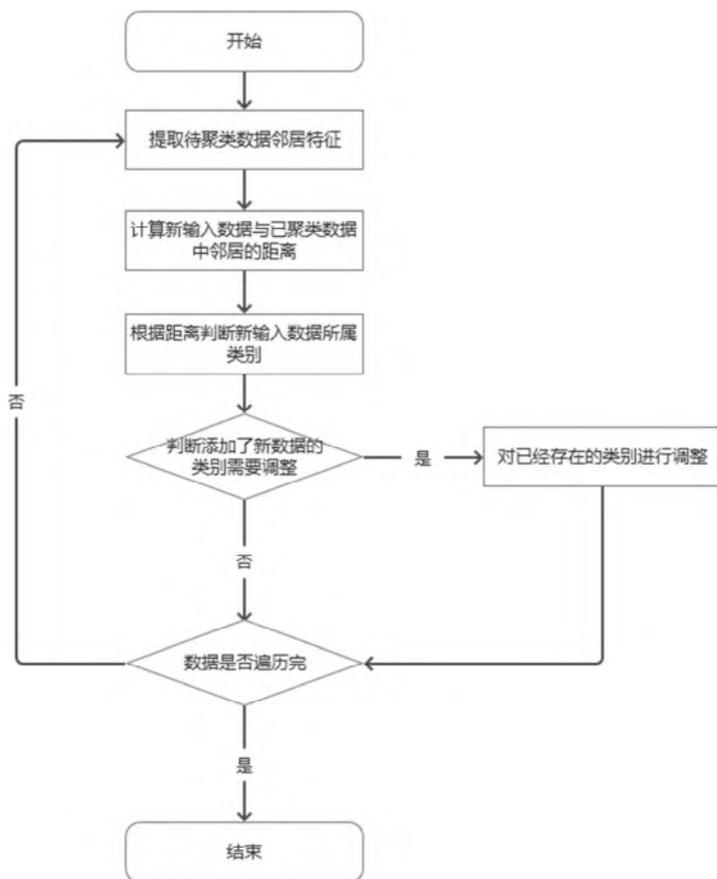


图 15-4

- (2) 提取待聚类数据的特征。对数据进行整体扫描获得每个点在所有数据中的位置、与最近邻点的距离、与次近邻点的距离、最近邻点的位置、次近邻点的位置。由这 5 个指标组成一个点的特征数据。
- (3) 对第 2 步取得的数据进行聚类。将 (2) 中得到的数据按照位置顺序逐个输入，根据输入点与已聚类数据中邻居的距离来判断新输入的点是否归入已有的类别。如果新输入的点与已有聚类中邻居间的距离满足突变的条件，则把新输入的点单独列为新的一个类，否则就把新输入的点归并到已有邻居的类别中，并对类别进行

顺序编号。同时通过计算已有类别之间的距离，记录各个类之间的最大距离  $\max d$ 。

- a) 确定突变判断标准，分为绝对值衡量标准  $D$  和相对值衡量标准  $r$ ，其中， $D/r=R$ 。
    - i.  $R$ ：是可承受的最大距离差，这个值可以根据样本数据确定。
    - ii.  $D$ ：为第一个十分位数的值，如果样本总数小于 10，也即第一个十分位数不存在，那么  $r$  是一个固定的值，为 0.1。
  - b) 计算各个样本之间的最近邻，并根据相邻两个点之间最近距离的关系，确定这两个点之间是不是存在突变。比如点  $a$  的最近邻是  $b$ ， $b$  的最近邻是  $X$ 。则根据如下的规则确定  $a$  和  $b$  之间是不是存在突变：
    - i. 如果  $X$  等于  $a$ ，也即  $b$  的最近邻也是  $a$ ，那么认为  $b$  和  $a$  是一类，从  $a$  开始的这个最近邻链条终止于  $b$ 。
    - ii. 如果  $X$  不等于  $a$ ，那么就要看  $b$  与  $a$  的距离  $R_{a-b}$  和  $b$  与  $X$  的距离  $R_{b-X}$ ，如果， $R_{a-b} > R_{b-X}$ ，并且  $R_{a-b} > D$  或者  $R_{a-b} > a \cdot r$ ，则判定  $a$  和  $b$  之间存在突变，从  $a$  出发的最近邻链条结束于  $a$ ，而  $b$  作为新链条的起始点继续。
  - c) 按照上面的规则遍历完所有的点，得到最初的聚类结果。
- (4) 对第 (3) 步的类别进行调整。按照第 (3) 步得到的各个类的编号，逐步顺序取出相邻两个类  $a$ ， $b$ ，计算两个类之间的类间距离，类内点之间的平均距离、类内点之间的最大距离、类内点之间的最小距离、类中点的个数。通过相邻两个类  $a$ ， $b$  之间的类间距离与最大类间距离  $\max d$  的比较，初步判断相邻的两个类是不是应该合并。如果初步判断  $a$ ， $b$  应该合并则进入第 (5) 步，否则继续，直到遍历完所有的分类为止。

a) 具体判断步骤如下：

- i. 计算相邻两个类簇之间的差值，其中这个差值定义为相邻两个类簇点之间的最近距离  $d$ ，并计算这两个相邻点到原点的距离  $S_a$ 、 $S_b$ 。
- ii. 计算这两个相邻类簇质心距离原点之间的距离  $h_1$ 、 $h_2$ 。
- iii. 计算这两个相邻类簇中质心距离原点近的一类簇的直径  $k$ ，以及另一个类簇各个点之间最近邻距离的均值  $m$ 。

iv. 如果满足如下条件, 则将两个类簇合并:

$$(R_{a-b} / S_a < \theta_1) \&\& ((R_{a-b} / \max(h1, h2)) < \theta_2 \parallel k / m < \theta_3)$$

$\theta_1$ 、 $\theta_2$ 、 $\theta_3$  是根据实验得到的,  $\theta_1$  的值为 10,  $\theta_2$ 、 $\theta_3$  的值分别为 0.8。

- (5) 如果初步判断  $a$ ,  $b$  应该合并, 那么就根据  $a$ ,  $b$  两个类: 类内点之间最大距离、类内点之间最小距离、类中点的个数, 进一步确定这两个类是不是需要进行合并。经过这一步判断确定  $a$ ,  $b$  两个类需要合并, 则把这两个类进行合并, 生成一个新的类。并把两个类别较大的编号赋予新的分类。返回第 (4) 步继续。
- (6) 对合并分类后的数据进行异常点检测。遍历合并后的聚类数据, 根据每个类别中的数据点数、与最近邻类的类间距离、与次近邻类的类间距离、类内距离平均值判断该类是否为异常类别, 如果为异常类别, 则将该类中的数据点识别为异常点, 也即当前话题属于长尾话题。
- (7) 获得最终分类之后, 统计各个类别包含的主体数量作为当前类簇话题的热度, 根据主体数量从高到低排, 提取前  $k$  ( $k$  为事先指定的值) 个热度的话题。

## 15.3 系统实现源码说明

### 1. 新词发现模块

实现新词发现模型算法的部分代码如 15-5 所示。

```
for (int i = 0; i < m - 1; i++) {
//获得词的链接矩阵
    if (res == null || res.size() == 0) {
        addWordLinkMap(res, c, i);
    } else {
        if (!res.containsKey(c[i])) {
            addWordLinkMap(res, c, i);
        } else {
            int step = i;
            while (true) {
                if (step + 1 < m) { //对给定的字符串逐个遍历获得最大公共子串
```

图 15-5

```
        step++;
        String tmpString = "";
        for (int j = i; j <= step; j++) {
            tmpString += c[j];
        }
//计算最终的成词概率
        Base.putKVMap(frequencyMap,
tmpString,Double.valueOf(1.0), true);
            } else {
                break;
            }
        } else {
            System.out.println("the data is not valid, please check!!!");
            break;
        }
    } else {
        break;
    }
}
    addWordLinkMap(res, c, i);
}
}
```

图 15-5 (续)

- (1) 这里以对单个字符串的处理为例来进行说明。我们首先对字符串从左往右进行遍历，找到当前字符串中与已存在的词的最大公共子串。
- (2) 获得最大公共子串，就要确定当前串单独成词的概率。对于这一步是我们这个算法的核心，如果判断为能单独成词，那么就可以作为候选词参与后续的计算，否则将会舍弃当前获得的字符串，而从新的位置开始进行遍历和截取。
- (3) 这里再多介绍一下当前算法的优势。从上面可以看到，我们是在一次循环里面就完成了最长字符串获取，之后只要做短语拆分就可以了，而不是通过逐一列举每个字符可能出现的后缀，这样就不需要按照每个不重复的字符去扫描全部文本，减少了计算耗时。
- (4) 我们这里是根据每个字符出现的概率来计算当前词是不是可以单独成词的，而不是通过凝固度筛选出可能的组合词。这样不用将词的出现范围限定在单个文本中，可增加多个字符组合出现的概率，避免了因为字符串的长度不在考虑的范围内而

无法识别的问题。

- (5) 当然对于算法本身的改进还可以有多个维度，比如这里在判断字符串的链接时用的是信息熵，这样对于小概率出现的词会有所偏重。另外，也可以根据编辑距离来计算，这样就可以避免这个问题。

## 2. 热度计算模块

实现热度计算模型算法的部分代码如图 15-6 所示。

```
for (int i = 1; i < dataList.size() - 1; i++) {
    //计算元素 i 的左距离
    left = dataList.get(i) - dataList.get(i-1);
    //计算元素 i 的右距离
    right = dataList.get(i+1) - dataList.get(i);
    if (left > right && (left > splitRange || left >
dataList.get(i-1)*0.1)) {
//判断元素 i 是否与左边的元素产生突变
        positionList.add(i);
        ratio = Math.abs(dataList.get(i)) -
Math.abs(dataList.get(i-1));
        splitRatio.add(ratio);
    }else { //判断元素 i 是否与右边的元素产生突变
        if (i == dataList.size() - 2) {
            positionList.add(dataList.size() - 1);
            ratio = dataList.get(dataList.size()-1) -
dataList.get(dataList.size()-2);
            splitRatio.add(ratio);
        }
    }
}
//对最后一个元素进行处理
right = dataList.get(dataList.size()-1) - dataList.get(dataList.size()-2);
left = dataList.get(dataList.size()-2) - dataList.get(dataList.size()-3);
if (left > right && (left > splitRange || left >
dataList.get(dataList.size()-2)*0.1) ) {
    positionList.add(dataList.size()-1);
    ratio = Math.abs(dataList.get(dataList.size()-1)) -
```

图 15-6

```
dataList.get(dataList.size()-2);
        splitRatio.add(ratio);
    }
}
//对最后一个元素进行处理
right = dataList.get(dataList.size()-1) - dataList.get(dataList.size()-2);
left = dataList.get(dataList.size()-2) - dataList.get(dataList.size()-3);
if (left > right && (left > splitRange || left >
dataList.get(dataList.size()-2)*0.1) ) {
    positionList.add(dataList.size()-1);
    ratio = Math.abs(dataList.get(dataList.size()-1)) -
Math.abs(dataList.get(dataList.size()-2));
    splitRatio.add(ratio);
}
//对数据之间的突变点分割比例做排序
Collections.sort(splitRatio);
//获得最大的分割比例
double maxDif = splitRatio.get(splitRatio.size() - 1);
//System.out.println(maxDif);
//对当前的分类进行重新调整
modifySplit(dataList, positionList, maxDif);
//System.out.println(positionList.size());
//System.out.println(splitRatio.size());
//获得其中的孤立点
getAlias(dataList, resultList, positionList, aliasList);
```

图 15-6 (续)

- (1) 首先对元素的集合进行逐个遍历，计算其左右邻接点的距离，通过该距离判断当前元素与邻接元素之间是不是存在突变，如果存在突变，则新生成一个类簇，否则将当前元素加入已经存在的类簇中。
- (2) 其中对于最后和开头的元素需要进行单独处理，这里虽然只列出了最后一个元素的单独处理，但是开头的元素也是需要单独处理的，处理方式一致，所以就不再赘述了。
- (3) 待所有元素遍历完成之后，就要对元素形成的类簇进行合并。具体的合并方法就是参照已经形成的类与其他类之间的距离和当前分割间隔的情况来进行，示意图如图 15-7 所示：

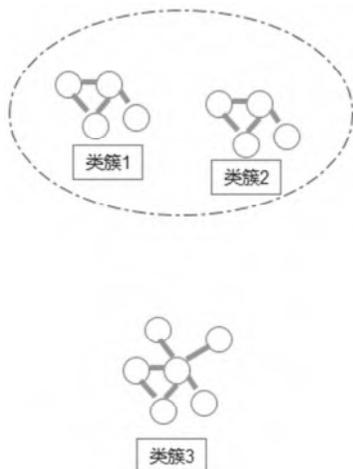


图 15-7

如图 15-7 所示，对于类簇 1 和类簇 2，它们之间的距离很近，并且相较于另外一个类簇 3，它们显得“非常”接近，所以就可以将它们合并成一个类簇。

- (4) 最终是要获得孤立的节点，这里是因为我们发现有一次计算中部分数据太少，所以就会体现为孤立点，但是孤立点并不一定是非重要的信息，如果对于它们不做处理可能会漏掉很多重要的信息，所以需要将它们单独发现出来，进行人工确认是否需要进行关注和处理，并把结果存储起来，作为后续处理的先验知识。

### 小结

至此我们就对舆情分析和挖掘系统的设计和实现进行了简要的说明和介绍。由于应用场景的限制，在某些方面肯定会存在许多不足和缺陷，如有更好的建议和想法，欢迎随时交流。