

人工智能导论 作业报告*

拼音输入法

BY 韩志磊†

1 原理与分析

1.1 算法

1.1.1 统计算法

为实现拼音序列到汉字的转换，基于预先给定的语料，可以采用概率统计的方法。**假定样本具有足够的代表性**，那么，通过统计样本中各字的出现频率，就可以知晓其大致的实际使用频率。然而，这样只能保证单字的出现概率最大，没有考虑上下文的影响。如果我们不对汉语的语法和语义进行分析（现有的语料也没这个条件），单纯就文本表现的频度而言，就需要一些更为复杂的概率计算。

假定汉字的出现概率可以用古典概型计算，事件 Q_n 代表输入长度为 n 的拼音串 $a_{1,2,3,\dots,n}$ ，事件 ω_i 代表第 i 位拼音代表的字符， ω 代表总的长度为 n 的字符串。我们希望概率 $P(\omega | Q_n)$ 最小，根据贝叶斯公式：

$$P(\omega | Q_n) = \frac{P(Q_n | \omega) \times P(\omega)}{P(Q_n)}$$

在一次试验中， $P(Q_n)$ 是定值；而 $P(Q_n | \omega)$ 是「某个字符串读音为 Q 的概率」，**在不考虑多音字的情况下**，这个概率是1。因此，我们只需求符合给定读音的某个字符串，它本身的出现概率最大。

由乘法公式立即可得：

$$P(\omega) = P(\omega_1 \omega_2 \omega_3 \dots \omega_n) = \sum_{i=1}^n P(\omega_i | \omega_1 \omega_2 \dots \omega_{i-1})$$

精确地计算需要每个 i 元词组出现的频率，这意味着指数级的预处理复杂度（或运行期指数级时间复杂度），对于这个问题是不可接受的。考虑到大部分时候，没有必要考虑已出现的所有前缀对当前字符判定的影响，可以适当对条件进行弱化。前面提到的只考虑单字概率的模型，就是不考虑条件项的结果。由此，我们可以从二元模型开始，即只考虑当前字符的前一字符的影响。

$$P(\omega) = \sum_{i=1}^n P(\omega_i | \omega_{i-1}) = \sum_{i=1}^n \frac{P(\omega_i \omega_{i-1})}{P(\omega_{i-1})} = \sum_{i=1}^n \frac{\# \omega_i \omega_{i-1}}{\# \omega_{i-1}}$$

如此一来，我们只需要统计大约6000*6000个词组的出现频率即可。

1.1.2 识别算法

现在我们已经完成了对原始语料的统计，并得到了一个二维矩阵occurrence用以标识二元词组的出现次数、一个一位数组count记录单个字符的出现次数，并且知道语料库的总大小。输入一个拼音的集合，如何给出概率最大的字符串呢？

在原型程序中，我使用的算法可以大致描述如下：

*. 2019年3月30日

†. 计76， 学号2017011442

算法逐拼音进行处理，令 W_i 为第 i 个拼音对应的可能字符集合， $P(\omega)$ 为 ω 对应的概率。则对于每一个 $\omega \in W_i$ ，考虑每一个 $x \in W_{i-1}$ ，有

$$P(\omega) = \max \left(P(x) \cdot \frac{\text{occurrence}[x][\omega]}{\text{count}[x]} \right), \forall x \in W_{i-1} \wedge \text{occurrence}[x] \neq 0$$

并且假定

$$P(\omega) = \frac{\text{count}[\omega]}{\text{TotalNumber}} \quad \forall \omega \in W_1$$

算法从 W_1 开始，至 W_n 结束。最终的结果就是 $\max(P(W_n))$ 对应的字符串。要回溯这个字符串并不困难，只要在算法进行时用指针记录使得当前 P 值最大的前一字符即可。（详见代码）

易见，此算法的复杂度为 $O(ns^2)$ 。其中， n 是拼音的长度， $s = \max(\text{length}(W_i))$ 。由于汉字集合是固定的， s 是一个常数，也就是说该算法关于输入拼音串是线性的。得益于此，以及 `cpp` 的高效，程序性能相当不错。

该过程的伪代码如下：

```
PROCEDURE FIND_BEST(PINYINS)
BEGIN
  PTABLE <- []

  CHARS <- LETTERSTABLE[PINYINS[0]]
  FTABLE <- []
  FOR CHAR IN CHARS
  DO
    FTABLE.ADD(<COUNT[CHAR]/TOTALNUMBER,NULL,CHAR>)
  DONE
  PTABLE.ADD{FTABLE}

  FOR PINYIN IN PINYINS[1,2,3...]
  DO
    LTABLE <- []
    FOR LCHARS IN LETTERSTABLE[PINYIN]
    DO
      L_PROBABILITY <- 0
      PREV_MAX <- NULL
      PREV_TABLE = PTABLE.BACK()

      FOR PREV IN PREV_TABLE
      DO
        N_PROBABILITY <- PREV.FIRST() * OCCURRENCE[PREV][LCHARS] / COUNT[PREV]
        IF N_PROBABILITY > L_PROBABILITY
        THEN
          L_PROBABILITY <- N_PROBABILITY
          PREV_MAX <- PREV
        DONE
      DONE
      LTABLE.ADD<L_PROBABILITY,PREV_MAX,LCHARS>
    DONE
    PTABLE.ADD<LTABLE>
  DONE
END
```

1.2 实验结果

1.2.1 案例

在实现了上述算法（代码说明见后）之后，得到了较为准确、迅速的原型程序。试选取其中成功的例子为例：

-
- * 清华大学计算机系
 - * 北京市人民政府
 - * 中国特色社会主义
 - * 爱国主义统一战线
 - * 五十六个民族
 - * 我还在上班呢
 - * 你今晚有空吗
 - * 精准扶贫
 - * 反法西斯战争
 - * 我的手机没电了
 - * 菜市场里人很多
-

不成功例子譬如：

-
- 义勇军进行曲
 >>义勇军进行区
 - 人民当家做主
 >>人民党家做主
 - 你能借我你的铅笔吗
 >>你能解我你的铅笔马
 - 大家有什么想说的吗
 >>大家有什么想说的马
 - 火箭成功发射上天
 >>和建成功发射上天
 - 和稀泥
 >>获悉尼
 - 今天回家比较晚
 >>今天回家比较完
 - 真是令人震惊
 >>真实令人震惊
 - 像风一样自由
 >>相逢一样子有
-

1.2.2 分析

由上可以看出，由于语料库的限制，原型程序所能准确识别只有某一类别的词组。譬如，由于给定的语料库绝大部分是新闻，因此时政、经济等方面的词组容易被识别；而相较之下，日常生活用语则较难被识别。当然，目前的算法也可以一部分地选择出有多个词组的最佳的字符串，像「菜市场里人很多」这样的句子就可以被识别，说明存在着概率控制的简单分词功能。

但这毕竟不是真正的分词，「人民当家做主」这种词，原本可以比较简单地分出三个词组，但由于概率模型中「国民党」这个词出现的频率太大，占据了主要的乘积项，导致识别失败。「义勇军进行曲」也是同理。

由此可以看出基于概率模型的输入法本身就有许多问题。一者受语料限制太大，二者受个别辞藻影响过深。此外还有缺失语义分析的问题：上面的例子中，「吗」这个语气词并不能准确地选择，原因在于对于「问句」这种形式无法判定。而且一旦输入的句子语料的偏差越大、长度越长，语义上的问题就越加凸显。

汉语的规则比较复杂，很难用上下文无关的语法来限定候选词的范围。不过，拥有语法分析的模型无疑能更好地处理相关问题（虽然有些时候会是事倍功半，因为汉语的反常现象很多）。

多音字是另一个非常恼人的问题。「火箭」这种词的出现频率没有「和建」高，因此识别失败；但是「和」读huó的概率极小，显然此处是不合理的。而「和稀泥」这类的词，由于缺乏字到多音的对应关系，也没办法识别。因此对于实际的输入法程序而言，某种双向的映射关系及数量关系是必须的。

2 实现

2.1 项目说明

请注意以下所有文件均为utf-8编码，请不要使用gbk编码运行。

按照说明的要求，最终的结构如下：

```

.
├── bin
│   ├── main
│   └── wrapper.py
├── data
│   ├── input.txt
│   └── output.txt
├── src
│   ├── count.cpp
│   ├── data.cpp
│   ├── main.cpp
│   ├── support
│   │   ├── cat.js
│   │   ├── count.js
│   │   ├── letters.txt
│   │   ├── p2l.txt
│   │   ├── pinyin2letter.txt
│   │   ├── process.js
│   │   ├── statistics.js
│   │   ├── translation.js
│   │   └── tr.sh
│   ├── table.cpp
│   └── ver2.cpp
└── todo.txt

```

说明中的「指定输入、输出文件」的功能只能由脚本wrapper.py实现，主程序main只进行计算。确保操作系统为64位Unix，且有libstdc++库，则可以直接使用编译好的程序。要指定输入、输出文件，向封装脚本传入参数即可：

```
./wrapper.py myinput myoutput
```

请确保输入文件是utf8编码、Unix换行符风格。

2.2 编译说明

核心程序以C++编写，预处理与支持脚本以JavaScript编写，包装脚本以Python编写。要完整地完从语料到可执行程序的过程，请确保安装C++11标准及以上的编译器、Node.js解释器，并预留2GB空间、至少4GB内存。要使用包装脚本，还需要Python2解释器。

首先将所有的语料**转换到utf8**，复制到support文件夹下，运行cat.js，会生成whole.txt。然后，依次运行process.js、statictics.js、count.js。**该步大概需要1小时时间完成。**

将生成的occurrence.txt和count.txt复制到src/源码目录，然后用C++11标准编译该目录下**除ver2.cpp外的所有C++源代码**，并链接为可执行文件。**注意：编译需要大概10分钟，4~7GB内存。建议先生成对象文件，再进行链接。**

将可执行文件复制到与Python代码同级的目录，以Python2运行脚本即可。

2.3 优化

2.3.1 朴素的优化法

main.cpp是经典的实现，同时ver2.cpp也提供了「改进版」。但值得说明的是，改进是相对于某种输入而言的，对于其他的输入，这个改进版反而会使识别率下降。考虑到鲁棒性的问题，我没有提供ver2的可执行版本。

ver2所做的优化是很朴素的。在不涉及语法、语义、增加语料、导入预设字典、使用第三方库的情况下，我们能利用的，只有occurrence和count两个数据而已。从某种意义上来说，如果只有这些信息，所做出的拼音输入法的准确率总是有上限的，更遑论人的主观思想的影响了。（我的意思是，像「ta hui lai le」这样的句子，究竟是「他」还是「她」呢？就算是目前顶尖的输入法，也只能通过**候选词**的办法来解决这个问题。）ver2思想是模拟「分词」的效果，通过人为地断开低概率词语间的联系，达到某种程度上的词模型。

ver2是这样实现这个不怎么出色的效果的：如果对于第i个拼音而言，无论选择哪一个汉字，其概率都与上一个字的概率的商小于某一个阈值，则从当前例程递归，将下一个字符视为新的句子开头。这样做的目的也很简单：当两个不怎么常用的词组合时，不会相互影响。

ver2的效果并不好，对于有些字符串，它的确有比较好的效果；不过对于很大部分的普通特有词语，反而会错误地将其断开。这当然是因为这毕竟不是专业的分词器，其内部的原理是概率（而且是古典概型这样的简单模型），而不是查表（以及更高级的人工智能方法）；但是还有一个很重要的原因在于**阈值的选取并不简单**。要区分「人民代表大会」和「香辣冰淇淋」这两种词，所需要的阈值非常微妙，因为概率之间的差异不大。

总的来说这是比较失败的优化尝试，但在不借助外力的情况下，固定语料的概率模型的最大限度大抵便是如此了。

2.3.2 关于「平滑」

据说如果不对概率进行插值，会导致比较严重的后果。当然，在实际的实现中我并没有发现这方面的问题，细细想来，所谓的「平滑化」处理，要解决的主要是首字出现频度的问题。就像「饕餮之欲」这种词一样，「饕」在首字出现的概率基本为0，但这不应该成为该词不匹配的原因。

就我所知有人采用了将句子开头视为一个特殊的字符的方法。这种方法导致的就是原语料「语义」的影响被放大，具体的表现就是「饕餮之欲」这种词永远无法被匹配。

关于这个问题，我想我在第一节推导是没问题的，乘法公式给出的展开式第一项是 $P(\omega_1)$ ，这项概率和字符是否在句首出现没有关系。因此如果有认真看伪代码，会发现第一项是被特殊处理过的。

理论上这样的实现方式是基本不需要平滑的，但是当然，我还是将它实现了。那么下一个问题是，平滑所用到的系数是否影响最终的结果？

平滑改变了所有的概率，并且使得概率之间的差异被缩小了，其结果应该是：概率低的序列被匹配的机率增大了；而概率高的序列则相反。可以预见的是，如果系数越大，对准确率的影响就越大。

实际的测试表明，在系数较少时，其基本不会影响准确性；但如果按照数量级递增（如令 $a=0.5$ 的时候），准确率会下降。

2.3.3 语料

出于时间、精力上的考虑，我只用提供的语料库本身完成了项目，但实际上，这个语料库的偏向性的确非常强，缺少一些流行文化，而且过于正式了。耗费一些寻找和重编译的时间找一个更好的语料库，其代价应该是值得的。要添加新的语料库也很简单，将cat.js和main.cpp里硬编码的文件名、字符数修改一下，然后重新编译一次即可，只是随着语料的增加，编译所需的时间和内存都会上升。据说万维网上有一些统计好的数据，不用再经过预处理，也是不错的选择。

2.3.4 高维模型

三元模型可能并没有看上去那么优秀，可以预见采用三元模型的情况下，其准确性也许在某些时候比现在的二元模型还要差一些。因为二元词组的出现频率，在经验中比三元词组多得多。采用三元词组来进行判定，不仅复杂度更高，往往还会「拆散」原本相连的词语。即：两个二元词组的综合考虑结果也许比一个三元词组的考虑结果更好。

二元与三元结合的模型好一些，即在判定时有限二元，然后以某个衰减系数乘以三元模型的判定结果作为参考。

对于更高维度的模型也是一样的。

2.3.5 分词

分词是很不错的办法（或者找一本字典之类的），但是比较慢。如果不在运行期进行分词，就必需很大的存储空间。同样的，也许更好的办法是词、字模型结合。先对在语料中存在的词语进行搜索（对数级时间？），如果没有，才进行基于字的判定。只是如果拼音和汉字的分词都交给第三方案库来做，那也未免太简单了一点：我们只需要把现有的字改成词就「万事大吉」了，甚至不需要更改算法本身。实际上，我们实际的工作只是生成汉字到拼音的（反向）查找表而已。

2.3.6 多音字

很可惜，这是我最希望完成的优化方案。但是竟尔无法找到任何关于汉字读音频率表一类的资料。简单阐述其原理在此：我们目前的模型之中 $P(Q_i | \omega)$ 恒为1（或者恒为某个常数），这是有违常理的。某字读某音，必然会有一个相对稳定的概率，就比如「和」读「huó」的概率远远低于读「hé」的概率。因此如果能统计出汉字使用中各读音的频率，那就可以对模型进行修正（以频率替换 $P(Q_i | \omega)$ ）。

3 想法

语言是用来沟通的工具，具体的语义总会随着环境的不同而不同。企图用古典概型来模拟真实的语言，只是很粗糙的一种手段而已。但考虑目前真正的输入法，其实大多也就是按照频率（来自于词库，以及来自于用户持续不断的输入，也就是「用户词库」或者「输入历史」）来运行的。按照概率排定候选词，的确在绝大多数情况下为我们带来了便利，这也是概率统计的基本原理。除非将来人机交互的技术发展到足够的程度，否则输入法的存在形式大概也就是如此。

一点感慨：写这篇报告用了一整天，打了不少字，我用的输入法还算是顺手，但也不是十全十美。我几乎可以肯定，这篇报告里会有一些错别字的存在。不过尚算是可以接受，毕竟就算是手写，我的水平也不一定比输入法高，而且它的确提升了我的效率。拼音作为汉字罗马化的历史产物，却在信息时代带来了这么大的便利，实在居功至伟。