人工智能导论.拼音输入法大作业

2018011324

计82

尤艺霖

1.算法思路

1.1.需求

编程实现一个简单的汉语拼音输入法，实现从拼音（全拼）到汉字（字串）内容的转换。

1.2.整体思路

先从数据集中统计出单字、二元组、三元组的出现频率存入文件，随后使用基于字的二元/三元的模型，列出算式并使用动态规划取得最优解，并通过记录路径的方式获取对应的字串。在二元模型下我们需要求出$\prod P(x_i|x_{i-1})$的最大值。这时候的动态规划需要知晓最后一位才能工作。在三元模型下我们需要求出$\prod P(x_i|x_{i-1}x_{i-2})$的最大值。这时候的动态规划则需要最后两位都被记录在状态里。

2.文件结构

根目录下存在：**readme.pdf**为说明文档，**data**子目录存储测试集，**src**子目录存储源代码和**json**文件

2.1.data目录下：

case1/case2/case3/case4.in/out/ans —— 测试用的数据集

input.txt/output.txt —— 按pdf要求加入的文件

2.2.src目录下：

data_c_1.py/data_c_2.py/data_c3.py —— 用于统计语料中词频/字频的程序

charlist.py —— 用于预处理字符集的程序

check.py —— 用于计算测试中的正确率的程序

pinyin_2.py/pinyin_3.py —— 二元/三元模型对应的拼音输入法程序

pinyin.py —— 选取出的最优模型，和**pinyin_3.py**完全一致

2_test.bat/3_test.bat —— Windows批处理文件，用于自动化测试

charlist.txt —— 微调后的字符集，和下发的字音对照表基本一致

c1freq.json/c2freq.json/c3freq.json —— 存放统计出来的词频的**json**文件

spelltochar.json/chartospell.json —— 存放拼音汉字映射的**json**文件

3.实现细节

3.1.数据预处理

首先通过**charlist.py**将提供的字符集转换成两个字典，分别负责字符到拼音（列表）的转换、拼音到字符（列表）的转换，并分别以**json**格式存储在**chartospell.json**和**spelltochar.json**中。

随后发现汉字存在多音字的问题，因此字符集不再是六千余个汉字，而是形如**(汉字,拼音)**的元组，这样的元组大约有**7500**个。如果不进行这样的处理，则会出现**地震（de zhen）**，**开车（kai ju）**这样的错误。

具体到实现上我的做法是按照非汉字字符将新闻内容分开，然后在分开的字符串中统计二元组/三元组出现的次数，从直觉上这样比将所有汉字连一起更优。同时使用**pypinyin**给语料注音，但考虑到**pypinyin**本身会出现错误，我的纠错方法是判断注音的结果和对应汉字的搭配是否出现在了给定的字符集中，如果没有出现就选取一个出现在字符集中的搭配。这样可以在大多数多音字上获得正确的注音并保证注音都在字符集中出现过。

3.2.二元模型

考虑二元模型中每一位只和它上一位有关，因此规定动态规划的状态为**dp[i,j]**表示以第**i**位结尾的前缀中、最后一位是**j**的状态的最优解，在具体实现中，**j**取的是在**spelltochar.json**中对应的编号。同时用**fr[i,j]**记录该最优解是从哪里转移过来的，方便最后生成字串。转移的时候枚举前一位的字**k**，然后计算P，尝试从**dp[i-1,k]**转移过来。平滑的方法选择ppt中的方法：采用$\alpha P(x_{i-1})+(1-\alpha)\frac{P(x_ix_{i-1})}{P(x_{i-1})}$，处理好边界情况即可。同时为了避免潜在的精度问题，在计算过程中进行取对数操作。

3.3.三元模型

与二元模型类似，但是状态需要改为**dp[i,j,k]**，其中**k**表示的是第**i-1**位上的字。通过特判来处理最开始的两个字符。其他与二元模型基本一致。转移的方式是枚举**i-2**位置上的字**l**，然后尝试从**dp[i-1,k,l]**转移过来。平滑的方式也与二元模型类似：采用$\alpha P(x_{i-2})+(1-\alpha)\frac{P(x_ix_{i-1}x_{i-2})}{P(x_{i-1}x_{i-2})}$，需要处理的边界条件更多一点，但不改变本质。也需要进行对应的取对数操作。

4.测试结果

4.1.测试环境

操作系统：Windows 10 家庭中文版

Python版本：**3.7.5**

拼音输入法程序**(pinyin.py/pinyin_2.py/pinyin_3.py)**的依赖：

拼音输入法的程序不依赖第三方库，只需要确保**json**，**sys**，**math**可用即可。

数据处理程序**(data_c_1.py/data_c_2.py/data_c_3.py)**的依赖：

依赖**pypinyin**，同时需要在根目录下建立一个名为**traindata**的新目录，并在该目录下放入提供的训练集，命名方式和内容格式必须和网络学堂上云盘链接中下载的内容相同（即为形如**2016-11.txt**的名称）。由于该训练集即使压缩后仍然很大，就没有随之上传。如果只需要测试拼音输入法，则不需要运行这几个程序。

4.2.测试数据

选取了四个不同的测试集，分别命名为**case1**到**case4**放在**data**文件夹下，其中**in**为后缀的为输入文件，**ans**为后缀的为对应的答案，**out**为后缀的是程序输出的结果。

四个数据集具有不同的特性：

case1由10条文言文构成，由于训练用的数据集都是新闻，可以合理推测这个测试集正确率会很低

case2包含30条从网络学堂上向同学征集的数据集中选出来的句子，用来寻找错误的个例进行分析

case3包含大约1000条测例，主要来自于学长提供的测例，用来评估参数的优劣，辅助调参

case4包含大约300条测例，主要来自于人智导群里发出的一个文件，质量偏低，用来评估模型优劣

4.3.测试方法

通过批处理文件在四个测试集上分别调用拼音输入法，将结果输入到对应的**out**文件中

随后通过**check.py**进行**out**文件和**ans**文件的对比，排除换行符的干扰后统计整句正确率和逐字正确率

4.4.测试结果及分析

4.4.1.正确率&参数选择

参数的选择主要是平滑参数α的调整，正确率主要观察**case3/case4**的正确率。

对于二元模型结果如下：

α	case3逐字正确率	case3整句正确率	case4逐字正确率	case4整句正确率
1×10^{-8}	0.7268	0.1344	0.6849	0.1092
1×10^{-9}	0.7837	0.2026	0.7471	0.1737
1×10^{-10}	0.8361	0.3119	0.8096	0.2969
1×10^{-11}	0.8633	0.3821	0.8394	0.3551
1×10^{-12}	0.8693	0.4002	0.8408	0.3669
1×10^{-13}	0.8695	0.4002	0.8422	0.3697
1×10^{-14}	0.8694	0.4002	0.8419	0.3697

对于三元模型结果如下：

α	case3逐字正确率	case3整句正确率	case4逐字正确率	case4整句正确率
1×10^{-9}	0.9084	0.5416	0.8739	0.4706
1×10^{-10}	0.9202	0.5827	0.8867	0.5154
1×10^{-11}	0.9229	0.5927	0.8895	0.5351
1×10^{-12}	0.9209	0.5877	0.8901	0.5322
1×10^{-13}	0.9206	0.5888	0.8894	0.5294
1×10^{-14}	0.9203	0.5897	0.8864	0.5266

首先是发现α必须要很小，才可以让他看起来像一个输入法。对这一现象我的认知是，如果α过大会使得字频占的比重太大，而给定的语料字频不甚合理，因而需要调小α，使词频在决策中占比更大。

综合表格数据和个人考虑，最终选择1×10^{-12}为二元模型的参数，1×10^{-11}为三元模型的参数。

对比两种模型，很明显从正确率角度考虑，三元模型远远优于二元模型，故选择三元模型作为**pinyin.py**的内容。

4.4.2.时间效率

没有进行精确的时间效率测量，对时间效率的评估主要靠感觉。

二元模型的运行速度是非常快的，不需要过多的分析。

三元模型的用时较高，从复杂度分析的角度看，若假设输入长度和每种拼音对应的字的数量是同一数量级，则可以分析出：每次状态转移是线性的，而状态总数是立方级别的，因而总复杂度是四次方级别的。在实践中输入长度往往在10到20，每种拼音对应的字的数量常常在20到30之间，加上**Python**本身效率低于**C++**，最终的结果是大约每一秒可以运行出一条测例。在运行大约一千条测例的**case3**的时候，大约需要**15**分钟来完成。

另一个不可忽视的点是三元模型读入包含统计数据的**json**文件所需要的时间。在机械硬盘上可能会很久。

4.4.3.对个例进行观察

4.4.3.1.case1

二元模型在**case1**上全错，符合预期。

三元模型在**case1**上有出现正确的，是如下几个：

岂因祸福避趋之

苟全性命于乱世

不求闻达于诸侯

铁马冰河入梦来

滕子京谪守巴陵郡

可以发现都是足够著名的文言文句子，并可以合理怀疑这些句子曾在语料中出现过。由此可见，对于语料中没有出现过的文言文，基于字的二元/三元模型是乏力的。

4.4.3.2.case2

二元模型在**case2**上有几个代表性错误：

北京是一个美丽的城市（北京市一个美丽的城市）

每隔四年一次的奥运会（每个四年一次的奥运会）

学会处理人际关系（学会处理人机关系）

数学分析本身就是难学的（术学分析本身就是难学的）

可以发现这几个错误在单个词中，比如**北京市**、**人机**中都是很合理的，但无法兼顾上下文。

在三元模型下，这四个测例都可以输出正确的结果，证明了三元模型对上下文的关注优于二元模型。

4.4.4.其他

在三元模型中，空间开销是不可忽视的，整体空间开销在读取**json**文件的过程结束的时候达到最大，接近**8GB**，随后回落到**4GB**左右，并在处理测例的过程中保持稳定。

测试的成功也证明了可以通过**python pinyin.py ../data/input.txt ../data/output.txt**进行测试（需要在**src**目录下调用**python**）。

最终随作业提交的**out**文件是三元模型运行的结果。

5.总结

本次作业中我成功完成了一个全拼拼音输入法，使用的方法是在基于字的二元/三元模型上使用动态规划。对比ppt中所讲述的，viterbi算法就是一种分层图上的动态规划，最终出来的代码也应当和直接按照动态规划写出的一致。而目前的数据要求下动态规划是可以在空间和时间的合理开销下完成任务的。

对比二元模型和三元模型，后者在正确率上是稳定优于前者的；在时空开销上则劣于前者。这也很符合直觉。三元模型利用了更多的信息和时间，理所应当通过更多的测例。由此也可以合理推测，使用四元模型可以获得更高的效率。使用四元模型主要的障碍一是统计时对内存压力很大，二是最终的**json**文件可能会很大，三是求最优解会很耗时。对应的解决方案应当是：分块统计词频，最后合并；最终只保留一定数量的统计结果；考虑时间复杂度主要由状态数提供，可以在每一层动态规划上都只保留一定数量的状态。四元模型的具体实现细节还有待将来完成。