拼音输入法大作业报告

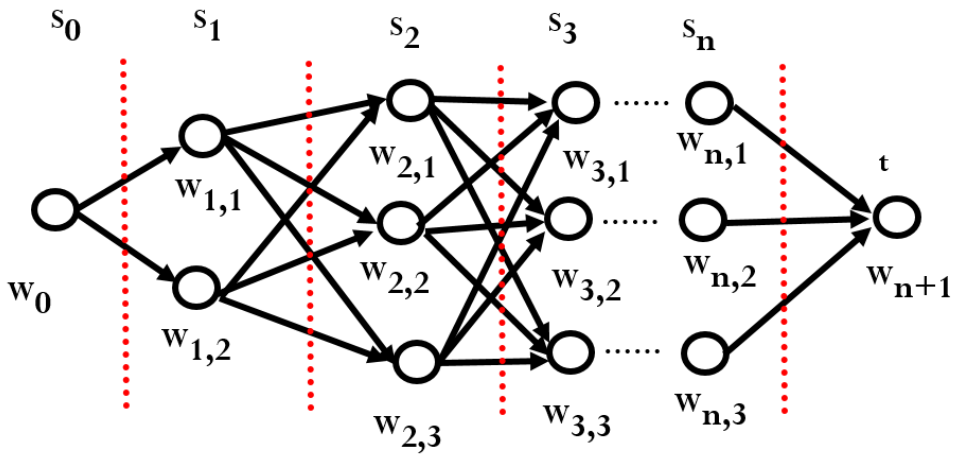
1 基本功能

从指定文本中输入若干行用空格分隔的拼音序列，不含标点以及汉字。输出拼音序列对应的汉字串到指定的文本文件中。

2 算法思路

整个算法基于一个概率模型。用$S = w_1 w_2 \dots w_n$表示某一长度为n的汉字串，$O = o_1 o_2 \dots o_n$表示输入拼音串，同时用w_i简化地表示从事件o_i转换为w_i，则有$P(S) = \prod_{i=1}^{n} P(w_i | w_1 w_2 \dots w_{i-1})$，其中$P(w_i | w_1 w_2 \dots w_{i-1}) = \frac{w_1 w_2 \dots w_i \text{出现的次数}}{w_1 w_2 \dots w_{i-1} \text{出现的次数}}$，基于语料统计得出。但是因为训练语料有限，涉及多个字相连的情况时很可能出现次数为 0 这样计算会引入难以计算上述概率的问题。因而化简为二元字模型，即每个字出现的概率仅和前一个字有关，对整句概率的计算简化为$P(S) = \prod_{i=1}^{n} P(w_i | w_{i-1})$。

对于拼音序列中的每个拼音，它们可能对应多个汉字，我们将每一种可能性都列出来，并用一条单向边连接前一汉字与后一汉字，如此构成一个复杂的网络（下图最终状态t可视为没有）。



如果每条路径(u, v)的长度看成u后出现v的概率，那么我们的目标就是求出上图中的最长路（对数意义下）。

假设我们已经通过语料得出了所有汉字对(w_i, w_j)对应的$P(w_j | w_i)$。可以用动态规划算法求解上图中的最长路径。逐层计算初始结点到当前结点的最长路，并保存路径。因为二元字模型是无后效性的因而可以只用一个列表维护当前层的所有结点，记录到当前结点的最长路径及其长度，并且每个结点选取前驱结点时贪心即$P(W_{i,j}) = \max_k \left(P(W_{i-1,k}) \cdot P(W_{i-1,k} | W_{i,j})\right)$，不断更新列表。如此计算最长路，最后只需要在维护列表中选取最长的路径输出即可。

3 算法实现过程

a) **开发环境：**python 2.7 结合 jupyter notebook

b) 模型训练及算法实现：

全程使用提供的国标一二级汉字表，存储为以单个拼音为键，以此拼音对应的所对应汉字的列表为值的字典，使用 cPickle 库进行字典的持久化存储，写入文件。测试集采用往届学长学姐整理的样例集。

一开始使用提供的 sina news 的汉语语料库进行训练。训练过程中维护两个字典，其中一个记录每个字出现的次数，以字为键，次数为值；另一个记录每一汉字对接连出现的次数，以两个汉字组成的元组为键，次数为值。利用正则匹配所有中文字串，逐字扫描更新字典，同样使用 cPickle 库进行字典的持久化，写入文件。

概率计算如下：$P(w_i)=\frac{w_i\text{出现的总次数}}{\text{出现的总字数}}$，$P(w_j|w_i)=\frac{(w_i,w_j)\text{接连出现的总次数}}{\text{出现总字数}}$，为了防止$(w_i,w_j)$连续出现次数为 0 导致不符合实际的情况，使用$(1-\lambda)P(w_j|w_i)+\lambda P(w_i)$作为真正的$P(w_j|w_i)$参与计算。首字选择概率按$P(w_i)$计算。

初次测试选取$\lambda=0$，在测试集上字的正确率为 76%，整句正确率为 36%。尝试更改了几次λ的值，效果不是很理想，由于是一开始的尝试，和最后选择的模型有些许出入，因而未做详细统计。

做出一个初步结果后思考了如何提高准确率，初步考虑了模型的改进。在测试时发现多音字对结果有比较大的影响，比如“家”字有音“jie”,而这又是个高频字，会导致涉及拼音“jie”时转换准确率较低。因而尝试考虑加入每个字对应拼音的权重，在此过程中参考了 wiki 上 Viterbi 算法的介绍。Viterbi 算法是用来解决隐含马尔科夫模型问题的。在本项目中，拼音串可以看成观察到的状态，要求解的汉字串可以看成隐含状态。Viterbi 算法不仅考虑了隐含状态中的转移概率矩阵，还考虑了隐含状态到观察状态的发射矩阵，这正好对应了从汉字到拼音的概率（发射概率），可以用来解决多音字问题。在上述的网络图中我们只需要将每个结点的发射概率加到所有指向该节点的边上（对数意义下）即可。

接下来的问题是如何从语料中获取发射概率。显然的做法是给每句话注音，显然人工注音费时费力，因而使用 pypinyin 库进行注音，把结果保存为一个字典，以单个汉字为键，值为一个子字典，后者以单个拼音为键，以该汉字注音成该拼音的次数为值。由于 pypinyin 速度比较慢故只选用了一部分语料进行注音处理，因为多音字占少数而且只需要一个大概的概率来修正模型，所以这么做丢失的准确度是可接受的。此外将计算首字选择的概率的分母调整为出现的所有句子数，以此更精确地描述实际情况。

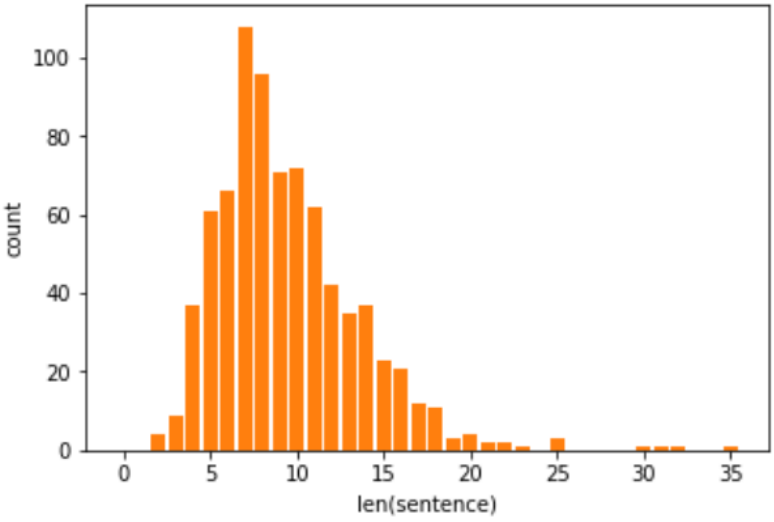
修正模型的同时我考虑了语料的更换，目前采用的新闻语料语言相比日常语言比较正式，有一定局限性，故而选择爬取知乎获取一些贴近日常生活的语料。为了加快速度以及反爬虫，我部署了分布式爬虫和代理池，该过程不赘述，最终获得了大约 4G 的语料，并选取其中 1G 成为注音语料，整合发射概率和转移概率到一个字典中。注音过程大约 1 小时，扫描全部语料大约 20 分钟。此外需要的一些全局信息单独用一个字典保存。至此最终使用的模型统计完毕。

在对比只用新闻语料、只用知乎语料和两者共用的效果之后，我最终选择了只用知乎语料。在最后的测试中取$\lambda=0$时单字的准确率达到了 84%，整句的准确率达到了 42%。可以看到还是有提高的，但仍有改进的空间。对于不同λ取值对准确率的影响将在性能分析中讨论。

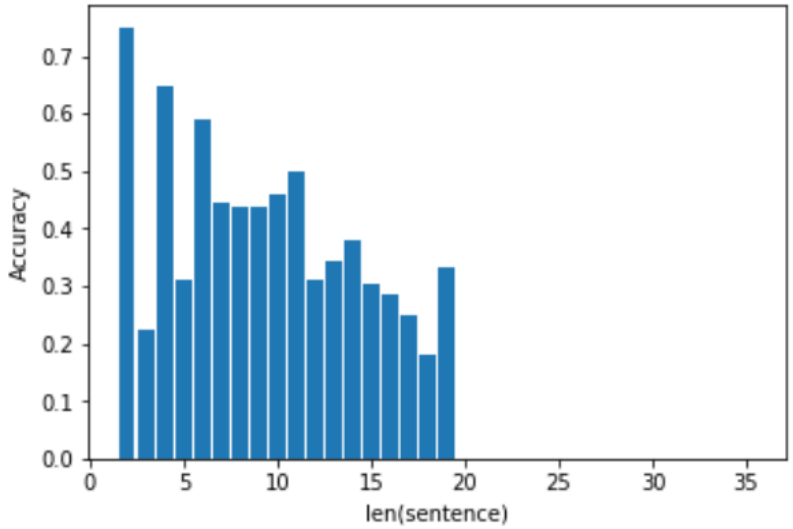
此外尝试了三元字模型，但因为某些问题未能进行下去，具体过程会在改进方案中讨论。

4 实验效果

测试集共包含 786 个句子（长度至少为 2），长度分布如下：



在$\lambda = 0$时统计了各长度句子的准确率如下：



其中最短的句子为 2 个字，最长的句子为 35 个字。

一个比较显著的特征是在句子比较短时准确率会随着句子变长而周期振荡，应该是由于使用了二元字模型，当长度为偶数时，句子可能包含更多的二元词，这样模型发挥的筛选和比较的作用会更加显著，效果也更好。例如“**你好世界**”、“**越过长城走向世界**”就是典型的二元词组合的句子，都转换正确。而这种效应随着句子的进一步变长逐渐减弱。

当字数超过 10 字时句子的准确率总体下降，这符合二元字模型不能很好地考虑上下文的特点。比如“**前除了不能起司会生之外十万能的/钱除了不能起死回生之外是万能的**”，当句子结构趋向复杂，概率模型就趋向模糊，从而表现地并不尽如人意。

此外，测试集中还有一些诗句，它们的句子结构比较松散，二元字并不能很好

地表现实力，但幸运的是，知乎语料比新闻语料对诗句具有更强的抗性，一些常见的诗句如“**满城尽带黄金甲**”，“**何妨吟啸且徐行**”都正确转换了，但其他诗句的表现就差强人意了，例如“**红小巷短又睡莲/红消香断有谁怜**”等就和正解差别很大。

此外，涉及句意的一些输入二元字模型也会出错，会出现“**他是我的妈妈**”这样的前后矛盾的句子。

整体来看二元字模型的效果还是很不错的，在占用较少资源的条件下做到40%+的长句准确率有些出乎我的意料。由此可见隐含马尔科夫模型非常适合于解决中文输入法的问题。

5 性能分析

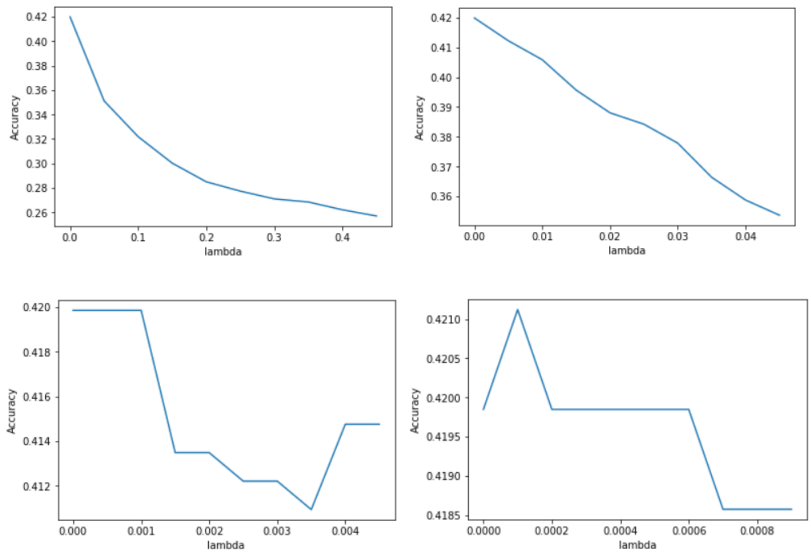
a) **速度：**

从硬盘中读取持久化存储的模型用时 5.07s, 转换 786 个拼音序列用时 20.77s, 平均每个拼音序列用时 0.026s。

b) **准确率：**

以下讨论λ取值对准确率的影响。

我们等间隔（横轴的十分之一）地选取λ，观察整句准确率的变化。逐渐减少间隔缩小最佳取值的范围。



可以看到最佳取值十分接近于 0，考虑到λ的存在是为了平滑概率函数，只在汉字对(w_i, w_j)中w_j不在语料中时才有意义，而我们的语料足够覆盖所有的常见二元组，某些不常见的二元组不出现对转换没有太大影响。因而选取$\lambda = 0$或$\lambda = 1e-4$都是可行的做法，因而之前的一些统计都基于参数$\lambda = 0$。

6 总结收获及改进方案

本次大作业主要运用了隐含马尔可夫模型实现了拼音输入法，尝试了不同语料的选取，不同参数的取值，最终选择知乎语料并且不平滑概率函数，在测试集上达到了 84%的单字正确率和 42%的整句正确率，体会到了统计模型的能力。

但是二元字模型仍有许多不足，不能充分地体现上下文的关联是二元字模型的重要缺陷。我尝试了将其更换为三元字模型，但在训练过程中因为字的三元组分布比较稀疏，字典的内存开销较大，超过了 PC 内存上限，动用了虚拟内存丧失了速度。遇到这一问题后我尝试改变了编码方式，不用字符串作为字典的键值，而改用一个与之一一对应的

数字作为汉字的本地编码代替汉字，并且剔除出现次数太少的三元组，节约了大约 20%的内存，仍然不能达到需求。之后我放弃了字典改用 python 自带的数据库 sqlite3 作为持久化存储的对象，将所有的三元组及其出现次数放到数据库的一个表中，但是数据量较大，相应地增加了查询的时间开销，经测试，转换长度仅为 3 的拼音序列就耗时 10s 以上，这样的速度是不能接受的。对于这样的情况初步有两种改进方向，其一是修改索引结构，将单个表分成多个表，然后建立关于表的索引，将索引读入内存，分两步查找，这样时间会线性地减少；其二是优化算法，在 dp 的过程中加入剪枝，不考虑概率过低的路径来加速求解。

7 参考资料：

Viterbi 算法的 wikipedia:：https://en.wikipedia.org/wiki/Viterbi_algorithm
分布式爬虫部署和代理池：https://github.com/SpiderClub/haipproxy