拼音输入法作业

2018011359 计84 乐阳

1. 算法思路

本次作业中实现了从拼音（全拼）到汉字的转换程序，分别实现了**基于字的二元模型和三元模型**。整体的思路为建立一个简单的语言概率模型，并利用维特比算法（动态规划）给出拼音最有可能对应的中文语句。

基于字的 N 元语言模型假设，下一个出现的字的概率分布与之前的 $N-1$ 个字有关。

$$P(w_1w_2\dots w_n)=\prod(w_i|w_{i-1}\dots w_i)\approx\prod P(w_i|w_{i-1}\dots w_{i-N+1})$$

整个字符串 s 的出现概率则为其中每个字符出现的条件概率的乘积。我们的预测目标就是找出出现概率最大的字串s。

$$s^*=\arg\max_s P(s)$$

二元模型与三元模型的算法细节有所不同，下面分别给出。

1.1 基于字的二元模型

二元模型假设，下一个字的概率分布只依赖于前一个字符，整个字符串的概率为：

$$P(w_1w_2\dots w_n)=\prod_{i=2}^{n}P(w_i|w_{i-1})P(w_1)$$

对当前字符的预测即最大化条件概率：

$$w_i^*=\arg\max_w P(w|w_{i-1})P(w_{i-1}w_{i-2}\dots w_1)$$

在本程序面对的问题中，每个拼音的同音字有多个，对应每个汉字的不同可能状态。从前一个汉字到下一个汉字的预测之间有一个**状态转移矩阵**$M=\{P(w_{is}|w_{(i-1)t})\}_{st}$，代表上一个汉字的每个状态转移到下一个汉字的每个状态的概率。其中w_{is}为第i个字的第s种可能，如果上一汉字有n_{i-1}种可能，下一汉字有n_i种可能，那么M的阶数将为$n_i\times n_{i-1}$。

根据状态转移矩阵和上一汉字的每个状态的概率，利用动态规划算法，可以计算出当前状态最可能从上一字符的哪一状态转移而来。直到最后一个字符的每个状态的概率计算完毕，从中选取出现概率最大的状态，回溯得到该状态之前的所有转移路径即可得到预测字串。根据二元模型的假设，这样得出的结果实际上是**全局最优**的。

利用贝叶斯公式，可以将条件概率转化为二字组合的出现概率与单字概率的商，后者可以通过对训练语料库统计词频得到估计。下式中$C(s)$代表字串 s 在语料库中出现的次数。

$$P(w_i|w_{i-1})=\frac{P(w_iw_{i-1})}{P(w_{i-1})}\approx\frac{C(w_iw_{i-1})}{C(w_{i-1})}$$

但由于语料库内容有限，可能出现$w_i w_{i-1}$词频为0的情况，此时需要做**平滑化**。

$$P(w_i|w_{i-1}) = \lambda \frac{C(w_i w_{i-1})}{C(w_i)} + (1-\lambda)\frac{C(w_i)}{C_0}$$

其中C_0为语料库中单字的个数，后一项代表了对单字概率的估计。λ为一个可调的参数，用于调整二元条件概率和单字概率的加权组合。这样就不会出现概率为0的情况而造成“断流”。

1.2 基于字的三元模型

三元模型假设下一个字的概率分布与前两个字有关。

$$P(w_1 w_2 \dots w_n) = \prod_{i=3}^{n} P(w_i|w_{i-1}w_{i-2})P(w_2|w_1)P(w_1)$$

在该假设下，仍按照二元模型那样做动态规划将很难得到全局最优解。因为二元模型的算法实际上是**贪心算法**，但二元假设保证了结果的最优性。在三元模型中如果只考虑前一个字的状态转移并应用三元概率进行计算，每轮生成一个新字符，并不能直接得到全局最优解。因为在前面可能犯下的“错误”是不可逆的，例如 shi qing hua yi ，前两个字最可能为“事情”，但全局的概率最高为“诗情画意”。

而如果枚举整个拼音串的每种可能性，则计算成本极高。一个输入法软件如果要求用户等待数秒才能得出结果，是无法让人接受的。

为了权衡**正确性**与**效率**，我在原始的动态规划算法上进行了修改。在原来的算法中当前字符的 n_i 种状态根据状态转移矩阵分别从上一字符的 n_{i-1} 种状态中挑选最优的转移路径，可能错过全局最优。在新算法中，（从第二个字符开始）每个当前可能状态要从上一字符的各状态中选择**前 k 优的**转移路径，总共记录 $k * n_i$ 条路径（字串）。这样一来，全局最优解尽管在某一步中不是局部最优的，但是可能是局部前 k 优的因此可以被保留下来，直到最后再脱颖而出。

原始的动态规划实际上等价于该算法在 $k = 1$ 时的情况，该算法的计算量为原算法的 k 倍。对于$i \geq 3$，第 $i-1$ 个字符到第 i 个字符的状态转移矩阵阶数为$n_i \times kn_{i-1}$，根据该矩阵第 i 个字符将记录下 kn_i 个可能的字串。注意在该思路下不能只记录当前状态由哪个状态转移而来，而要记录**整条路径**才可以，这是计算三元条件概率并记录记录前 k 优的方法所必须的。

三元的条件概率估计类似二元模型的处理方法，只不过三元模型将更加稀疏，更容易出现三元组词频为0的情况。此时平滑要综合考虑三元、二元和一元的概率。

$$P(w_i|w_{i-1}w_{i-2}) = \lambda_1 \frac{C(w_i w_{i-1} w_{i-2})}{C(w_{i-1}w_{i-2})} + \lambda_2 \frac{C(w_i w_{i-1})}{C(w_{i-1})} + (1-\lambda_1-\lambda_2)\frac{C(w_i)}{C_0}$$

这里有两个参数λ_1, λ_2要选择，可以用一个参数 γ 来统一，便于参数调整。

$$\lambda_1 = \frac{C(w_{i-1}w_{i-2})}{C(w_{i-1}w_{i-2}) + \gamma}$$
$$\lambda_2 = (1-\lambda_1)\frac{C(w_{i-1})}{C(w_{i-1}) + \gamma}$$

这种平滑方法是自适应的。如果$w_{i-1}w_{i-2}$出现次数较多，则λ_1接近1，三元概率为主导。若否，则为二元或一元概率为主导。

1.3 其他优化处理

考虑多音字。汉语中同一个字的不同读音很可能有完全不同的意义和用法，如果不考虑多音字那么每个词语会有多个读音，而且其中有很多是极其荒唐的。例如 kai ju，预测应为“开局”，但在不区分多音字的情况下会输出“开车”。本程序将读音不同的汉字直接看作不同的字，为此要重新制备字典，为不同音的汉字编号备查。接着利用 `pypinyin` 为语料注音，然后转换为字与字的读音编号再统计词频。

填充前后标记。对于N元模型，在句首添加$N-1$个start标记并在句尾添加 1 个end标记，对语料和预测的字串都做此处理。这样做的好处在于每个字都可以统一按照 N 元概率处理。此外，对于一些显然不出现在句首句尾的组合也能加以排除。例如 de guo，在不填充的情况下输出为“的国”，而添加填充后则能输出“德国”。

使用Trie树存储模型并序列化。语言模型是一系列的“词-词频”键值对，如果全部存入一个大的字典，查找将十分费时。注意到每个字串的前缀字串也都会被统计，因此考虑用Trie树结构，用字典套字典来进行前缀存储，每个“节点”只需要存储一个字符以及从树根到该节点的字符串的词频。一方面这种结构节省内存（节省接近一半），同时提高词频查询的效率（树状多层次查找）。为了保留模型的数据结构并高效读取，使用 pickle 模块直接将整个模型序列化并压缩存储，并使用`pickle.HIGHEST_PROTOCOL`进一步压缩模型文件的大小。（文件大小约300MB）更小的文件也拥有更快的加载速度，经过测试模型的加载时间约为12秒左右。

采用对数概率并用Numpy计算数值。根据词频得出的概率数值极小，随着状态的转移概率将十分接近0，容易受到精度的影响。为此利用词频计算概率时先取对数再返回，在动态规划时即可采用熟悉的相加的方式进行，而且数值的大小控制在$-10^2 \sim -10^3$左右。除了查询词频的操作以外，本程序尽量避免迭代循环，利用 Numpy 矩阵的接口来完成诸如对每个元素执行的四则运算、矩阵与向量按行相加、按行求最大值等等操作。从而利用 Numpy 内部的并行线性代数算法来加速程序运行。

2. 程序说明

语言模型主要实现在`myModel`类中，其各接口的功能为：

```
class myModel(){
    def __init__(self, lamb=0.99, gamma=10):
        #lamb和gamma分别为二元、三元模型的参数
    def getprob1(self, s):
        # 查询s中字符的单字频率的概率，返回numpy一维向量
    def getprob2(self, s1, s2):
        # 查询s2中字符到s1中字符的条件概率，返回numpy转移矩阵
    def getprob3(self, s1, s2, s3):
        # s2与s3长度相等，查询从s2s3到s1的条件概率，返回numpy转移矩阵
}
```

拼音-汉字转换函数为：

```
def convert(data, model, pinyin_dict, duoyin_dict, gram, k, pad=False,
debug=False):
    # 拼音汉字转换器
    # input:
    #    data: 输入的拼音串
    #    model: 词频模型
    #    pinyin_dict, duoyin_dict: 拼音词典，多音字词典
    #    gram: 二元或三元模型
    #    k: 维特比算法记忆前k种种选择
    #    pad: 是否在句首句尾添加st和ed标记
    #    debug: 调试输出前5种选择
    # output:
    #    转换完毕的中文字串
```

程序需要在src目录下执行，否则可能发生路径错误。一个程序运行示例为：

```
$ python pinyin.py ../data/input.txt ../data/output.txt
model: gram=3 gamma=10 k=3
mode: file
Loading model...
Model loaded in 12.221218824386597 seconds
Convert finish! Elapsed: 37.401835680007935 s
```

程序支持两种模式，文件模式和交互模式。其中文件模式需要命令行最后两个参数分别为输入、输出文件路径，当未指定文件路径时，进入交互模式。交互模式中，用户可以不断输入拼音字串并查看转换结果，输入`q`退出，下面是一个示例。

```
$ python pinyin.py
model: gram=3 gamma=10 k=3
mode: interactive
Loading model...
Model loaded in 12.751466035842896 seconds
>wo shi shei wo cong na li lai wo yao dao na li qu
我是谁我从哪里来我要到哪里去
time usage: 154.62779998779297 ms
>q
Bye
```

程序还支持手动指定二元或三元模型及参数，需要传入`gram`、`lamb`、`k`、`gamma`等参数（如果不传则取默认值）。以下是一个示例。

```
$ python pinyin.py gram=3 k=5 gamma=100
model: gram=3 gamma=100 k=5
mode: interactive
...
```

3. 参数选择

在语料库中随机抽取长度为5~15个字符的句子共3000句作为测试集，用`pypinyin`注音作为测例输入，与正确的语句进行比对，分别统计句正确率和字正确率。（见`input.txt`）这个测试集的“过拟合”程度可以说是相当高，但是有效的语言模型的训练语料和实际用途应该是一致的。泛化能力只是衡量模型性能的一个方面，在训练集上的表现同样重要。

3.1 二元模型的参数选择

二元模型的运行时间约为40ms/句，在不同参数λ下的正确率如下：

lambda	字正确率	句正确率
0	50.91%	0.7%
0.5	86.06%	42.6%
0.9	88.40%	49.5%
0.99	88.42%	50.2%
0.9999	88.40%	50.1%

可见二元模型的最佳参数约为 $\lambda = 0.99$ ，此时字正确率为88.42%，句正确率50.2%。

$\lambda = 0$时模型退化为一元，此时模型每次的预测为频率最高的字，基本不可能做出整句的预测。

3.2 三元模型的参数选择

根据本程序三元模型的约定，可调整的参数为权重 γ 以及动态规划的记忆规模 k。

首先，固定 $k = 1$ 来选择γ 的值。

gamma	字正确率	句正确率
1	95.6%	78.4%
10	95.7%	78.4%
100	95.6%	77.9%

可见 γ 的选取对模型的表现影响不大。从二元模型到三元模型，句正确率有奇迹般的提升。这是由中文的语言结构所决定的，三元组提供的上下文信息能基本将字确定下来。

下面固定 $\gamma = 10$，在性能和运行时间上对 k 的选取做权衡，下表中运行时间为3000句转换的总时间。

k	运行时间	字正确率	句正确率
1	152.6s	95.65%	78.4%
2	280.1s	96.91%	83.6%
3	390.2s	97.11%	84.1%
5	639.1s	97.2%	84.5%
7	903.1s	97.2%	84.5%

从实验数据来看，转换耗时基本与 k 成正比，与理论推算结果一致。随着 k 的增大，转换正确率单调上升，这是因为预测结果越来越接近于全局最优。理论上当$k \to \infty$ 时计算出的结果就是全局最优。k 从1增大为2或3时句正确率有较明显的提升，k 变得更大时变化就不大了。综合考虑正确率与速度，默认选择 $k = 3$，此时正确率较高且转换时间约为130ms/句。

3.3 前后填充的效果

在前面的测试中，默认添加了前后填充。下面测试一下在同样的模型和参数下，是否添加填充对正确率的影响。以下测试为三元模型，$k = 3, \gamma = 10$。可见添加填充后正确率有所上升。

添加填充	字正确率	句正确率
是	97.11%	84.1%
否	96.57%	81.2%

4. 效果展示与分析

4.1 好的例子

本节中二元三元模型预测均正确。笔者总结，模型容易预测的句子的特点为：**句法简单、由多字词语拼接而成、不含生僻用法。**

- qing hua da xue ji suan ji ke xue yu ji shu xi

  二元、三元模型：*清华大学计算机科学与技术系*

- mian xiang dui xiang de cheng xu she ji ji chu

  二元、三元模型：*面向对象的程序设计基础*

- xiao chu kong ju de zui hao ban fa jiu shi mian dui kong ju

  二元、三元模型：*消除恐惧的最好办法就是面对恐惧*

4.2 二元模型与三元模型的区别

三元模型相比二元模型的优势在于识别多个字之间的联系，进而显著提高整句预测效果。

- mao ze dong si xiang he zhong guo te se she hui zhu yi li lun ti xi gai lun

  二元模型：*毛泽东思想与中国特色社会主义理论体系**该论*** 三元模型：*毛泽东思想和中国特色社会主义理论体系概论*

在这个例子中二元模型在“概论”二字预测错误，而三元模型预测正确。查看词频可知“该论”二元组的频率比“概论”要高，但语料库中有“体系概论“这样的语句存在，可见三元模型有二元所不具备的整句推断能力。 此外，该例子中仅仅是二元模型就能预测出绝大部分的字。经过测试模型在党建类新闻语句的预测方面性能都不错，党和国家领导人的名字也都能预测对。这是由于语料库中的内容大多为语言正式、官方的新闻所致的。

- ta yang le yi zhi qing wa dang chong wu

  二元模型：*他养了**一致青瓦**当宠物* 三元模型：*他养了一只青蛙当宠物*

  这个例子体现出，二元模型只能考虑二字词语，而”一致“、”青瓦台“出现频率较高故发生错误。三元模型则能识别出”一只青蛙“这样的短语。

- xian di chuang ye wei ban er zhong dao beng cu

  二元模型：*现地创业委办二中到崩殂* 三元模型：*先帝创业未半而中道崩殂*

  能够预测出古文并非是模型有多智能，只是语料库中恰好有这样的语句。不过三元模型的优势在于能够准确的识别出整句从而做出正确的预测。

4.3 三元模型的多个候选

稍稍修改代码，可以让三元模型在交互模式中每次输出预测概率为靠前的几个选择，这和实际的输入法运作方式相同。

- bo da jing shen

  1：博大精神 2：博大精深 3：拨打景审

  在这个例子中很不幸模型没能预测正确，可见基于字的三元模型在处理四字词语上还有很多不足。但得益于多个候选的机制，还是能在前几种选择中命中答案。

4.4 坏的例子

在很多语句上模型的表现并不好，一方面由于语料范围有限，另一方面在于基于字的模型算法较为朴素灵活性不佳。

- huo er de lian xu han shu kong jian shi ba na he kong jian

  模型预测：霍尔的连续函数空间是巴娜和空间 正确答案：霍尔德连续函数空间是巴拿赫空间

  语料库中没有的**专业术语**不可能做出正确预测。

- fen zhan san xing qi zao tai ji suan ji

  模型预测：奋战三星启灶台计算机 正确答案：奋战三星期造台计算机

  非正式、不完整的句子很难预测。如果语句为“奋战三个星期制造一台计算机”就能够预测正确。

5. 总结与改进

在实验中我总结出关于模型的几点经验，前文中已经提到，在此重述。

- 动态规划多记忆几种选择可以使结果更接近全局最优。
- 模型适用于句法简单、由多字词语拼接而成、不含生僻用法的句子。
- 训练语料与执行的预测任务一致时，模型效果好。

本次实验使用的模型相对简单，还有很多可能改进的策略。

- 使用基于词的模型，而非简单统计字的组合。但可能遇到词语非常稀疏的问题，而且会忽略掉词语之间的关联。可能需要将字与词的模型结合起来效果才好。
- 去掉出现频率非常小的字的组合。能够极大的减小模型大小，且性能影响不大。
- 使用更大规模的语料库。因此能适应不同种类的语句。