实验报告

——拼音输入法

1 实验原理

1.1 算法基本思路–$viterbi$算法

本实验中，采用基于字的二元、三元、四元模型，以及基于词的二元、三元模型，分别实现了拼音到汉字的转换。实际上，这是一个隐马尔科夫模型，因为不能够直接得到某一个句子出现的概率，只能通过每个汉字/两个汉字相邻出现的概率去获取我们想要的输出。从而，最根本地，目标是求解一个最大值问题：

$$maxP(S) = max\prod_{i=1}^{n} P(w_i|w_1 \dots w_{i-1})$$

也即某个汉字串出现的概率。

底层思路是使用动态规划的$viterbi$算法。

如右图。在拼音输入法中，每一个节点$w_{i,j}$都对应一个汉字，而s_i则对应字的拼音。输入给定了s_1至s_n的值，也因而可以由此生成右图所示的这样一幅图。

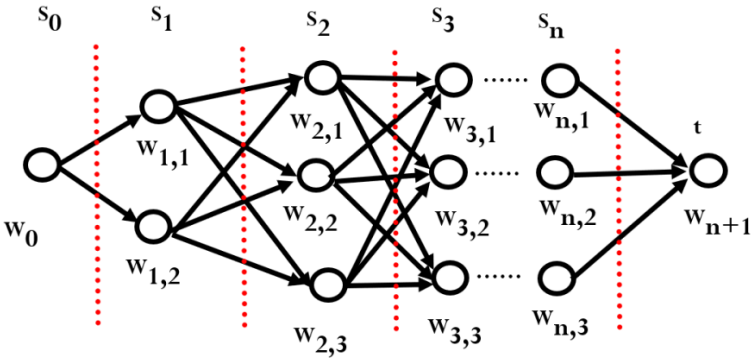


图 1.1 $viterbi$算法示意图

那么所需要完成的事情实际上是寻求一条由w_0而始，至w_{n+1}而终的最短路径，再将每一个节点对应的汉字做一个输出。

从动态规划的思路来看，可知起点w_0到图中某一节点$w_{i,j}$的最短路径值：

$$Q(w_{i,j}) = \begin{cases} \min_k \left(Q(w_{i-1,k}) + D(w_{i-1,k}, w_{i,j})\right), i \neq 0 \\ 0, i = 0 \end{cases}$$

其中$D(w_{i-1,j}, w_{i,k})$表示这两个节点之间的距离。

所谓的“距离最小”，实际上是要求“概率最大”，因此将概率取负对数，就可以转化为距离。

1.2 基于字的二元模型

在二元模型中，某个字出现的概率采用如下式子来计算：

$$P(w_{i,k}|w_{i-1,j}) = \frac{count(w_{i-1,j}w_{i,k})}{count(w_{i-1,j})}$$

同时，考虑到可能会存在$P(w_{i,k}|w_{i-1,j}) = 0$的情况，可以采用如下平滑处理：

$$P'(w_{i,k}|w_{i-1,j}) = \lambda\frac{count(w_{i-1,j}w_{i,k})}{count(w_{i-1,j})} + (1-\lambda)P(w_{i.k})$$

其中λ是一个人为规定的参数。

那么两节点之间的距离：

$$D(w_{i-1,j}, w_{i,k}) = -\log\left(P'(w_{i-1,j}|w_{i,k})\right)$$

因而问题转化为求解：

$$maxP'(S) = \min\left(-\sum_{i=1}^{n}\log\left(P'(w_i|w_{i-1})\right)\right) = \min\left(\sum_{i=1}^{n}D(w_{i-1}, w_i)\right)$$

正如 1.1 中所示，这可以利用$viterbi$算法解决。

在 1.1 的分析中，可知，首先要统计语料中所有字出现的次数；以及两个字紧邻的次数。在本实验中，将语料首先按标点符号进行粗略断句（./preprocessing/json_to_sentences.py），并且输出为一个 txt 文件。之后，对这些句子作统计，需要统计的数据有：单字出现的总次数，以及两字相邻出现的次数。这些数据都保存为 json（./sentences_to_stats.py），key 值为某个字，value 是一个字典，这个字典中的键是该字后面相邻的字，值是两字相邻出现的次数。为了统计方便，这个字典中的键还包括'total'，对应该字出现的总次数；'begin'，对应该字在句首出现的次数；'end'，对应该字在句末出现的次数。

在一开始的算法中，我忽略了w_0到w_1的状态转移的距离（见图 1.1），以至于出现了一些错误，这在后面会进行分析。之后对句首与句末的单字出现概率也进行了考虑，也就是说

$$P(w_1|w_0) = P(w_1\text{在句首出现})$$

之后对其进行平滑处理，并取负对数，作为w_0与w_1之间的距离。

同时，对句末的w_n到终点w_{n+1}的距离可以对如下概率作平滑之后取负对数来获

得：

$$P(w_{n+1}|w_n) = P(w_n在句末出现)$$

1.3 基于字的三元模型

对于三元字模型，可以知道

$$P(w_i|w_{i-2}w_{i-1}) = \frac{count(w_{i-2}w_{i-1}w_i)}{count(w_{i-2}w_{i-1})}$$

但是，如果单纯把它作为$viterbi$算法中两节点之间的距离，是不尽合理的。这是因为有很多词本身就是由两个字w_iw_{i-1}而非三个字组成的，如果使用三元模型对这种词进行模拟的话，显然将不尽如人意。因此，本实验中采用一种加权的方法：

$$D(w_{i-1,j}, w_{i,k}) = -\log A$$

其中

$$A = \lambda P(w_i|w_{i-1}) + (1-\lambda)P(w_i) + \mu P(w_i|w_{i-2}w_{i-1})$$

也即对二元模型、三元模型、字本身出现的概率总体做一个加权。这里存在一个疑问，就是这样加权出的结果不再严格地小于 1，不过可以做一个简单的归一化处理：

$$\frac{A}{1+\mu} = \frac{\lambda}{1+\mu}P(w_i|w_{i-1}) + \frac{1-\lambda}{1+\mu}P(w_{i.k}) + \frac{\mu}{1+\mu}P(w_i|w_{i-2}w_{i-1})$$

取负对数之后，对于同一层之间节点的距离比较而言是没有影响的。因此在程序中不需要考虑这一个归一化系数。

另外，在处理第一、第二层节点w_1和w_2时，不需要用到三元的概率（因为前面只有不到两层节点），处理方法与 1.2 二元模型一致。

1.4 基于字的四元模型

对于四元字模型，可以知道

$$P(w_i|w_{i-3}w_{i-2}w_{i-1}) = \frac{count(w_{i-3}w_{i-2}w_{i-1}w_i)}{count(w_{i-3}w_{i-2}w_{i-1})}$$

同 1.3 分析，采取加权：

$$B = \lambda P(w_i|w_{i-1}) + (1-\lambda)P(w_i) + \mu P(w_i|w_{i-2}w_{i-1}) + \varphi P(w_i|w_{i-3}w_{i-2}w_{i-1})$$

节点距离取用：

$$D(w_{i-1,j}, w_{i,k}) = -\log B$$

在处理第一、第二、第三层节点w_1、w_2、w_3时，不需要用到四元的概率（因为前面只有不到三层节点），处理方法与 1.3 三元模型一致。

2 实验分析：实例与性能

以下多用图来进行展示，原始数据为 myInputMethod/原始数据.xlsx

2.1 二元模型的实例与性能分析

(1) 未处理第一层节点：

在二元模型中，如果认为第一层节点离起点的距离为 0（即等可能出现），正确率不尽如人意。具体数据如下表：

λ	0.6	0.7	0.8	0.9	0.95	0.99	0.999	0.9999
句正确率	0.222	0.237	0.241	0.252	0.2542	0.2579	0.2561	0.2561
字正确率	0.789	0.796	0.8	0.801	0.8015	0.8013	0.8013	0.8018

表 2.1.1 二元模型（未处理w_1）

句正确率很低。对输出进行分析，可以看出一些很明显的错误：

```
['yi', 'zhi', 'piao', 'liang', 'de', 'xiao', 'hua', 'mao']
['圯', '直', '漂', '亮', '的', '消', '化', '贸']
['yi', 'zhi', 'ke', 'ai', 'de', 'da', 'huang', 'gou']
['圯', '直', '可', '爱', '的', '大', '黄', '狗']
```

可以看出，会导致在处理生僻字上出现问题。这是由于“圯”这个字在所有语料中仅出现了一次，而后面跟的字恰好是“直”。这导致：

$$P(\text{圯}|\text{直}) = \frac{1}{1} = 1$$

因此就会被作为距离最短的节点被选中。这是不合理的——我们需要衡量圯这个字本身在句首可能出现的概率。而对于在中间的字，由于出现次数较少的字本身离上一层节点的距离都会很大，因此就不需要把“这个字出现在这个位置的概率”做特殊处理了。

增加了第一层节点（句首）离起始节点的距离（见图 1.1）之后，此类案例存在一些改善：

```
['yi', 'zhi', 'piao', 'liang', 'de', 'xiao', 'hua', 'mao']
['一', '直', '漂', '亮', '的', '消', '化', '贸']
['yi', 'zhi', 'ke', 'ai', 'de', 'da', 'huang', 'gou']
['一', '直', '可', '爱', '的', '大', '黄', '狗']
['ji', 'qi', 'xue', 'xi', 'ji', 'qi', 'ying', 'yong']
['机', '器', '学', '习', '机', '器', '应', '用']
```

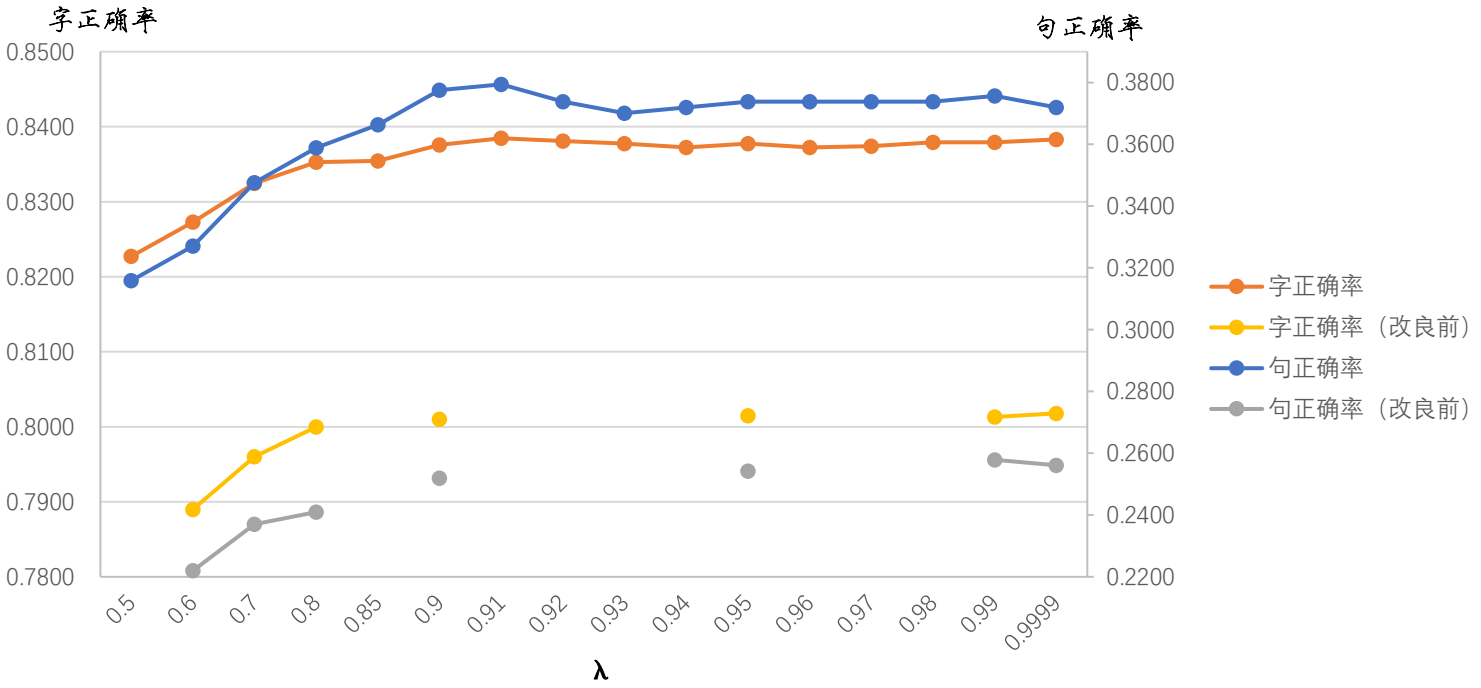
但仍然存在着很多错误，例如在语料中，“消化”出现的概率大于“小花”，而“化贸”又战胜了“花猫”，从而输出为“消化贸”这个令人不明所以的词。这就是二元模型的局限

之处了：对于三字词的处理并不那么令人满意。同时，也导致第三个例子输出为“机器应用”而非“及其应用”。所幸，这些在结合三元模型之后会有明显的改善，这在下一节中会进行分析。

不过，对句首节点做处理之后，至少生僻字不会出现得如此频繁，从准确率上来考察处理w_1的合理性：

λ	0.9	0.91	0.92	0.93	0.94	0.95	0.96	0.97	0.98	0.99	0.9999
句正确率	0.3776	0.3794	0.3738	0.3701	0.3720	0.3738	0.3738	0.3738	0.3738	0.3757	0.3720
字正确率	0.8376	0.8385	0.8381	0.8378	0.8372	0.8378	0.8372	0.8374	0.8379	0.8379	0.8383

画图之后可以更直观地观察：



可以看出有很明显的提升，同时，改良后，λ的最优取值为 0.91，此时句正确率为 0.3794，字正确率为 0.8385。

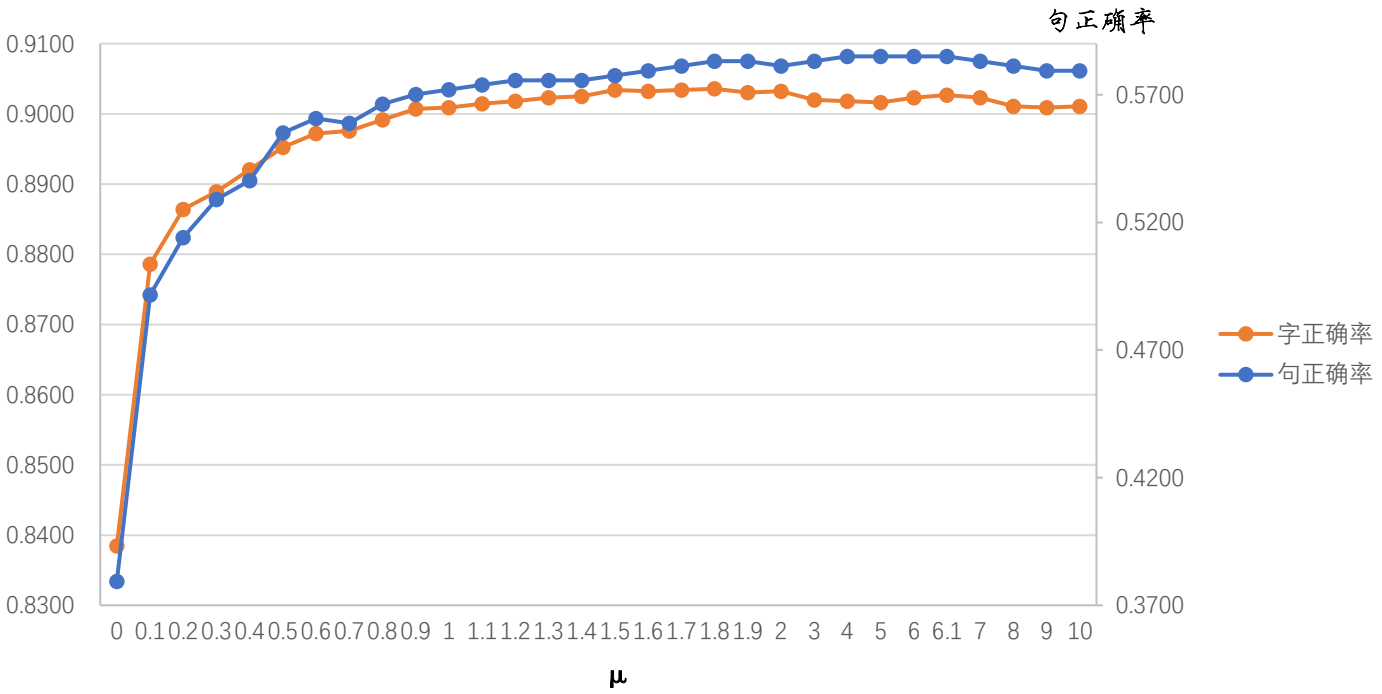
2.2 三元模型的实例与性能分析

使用三元模型，就意味着在确定下一层节点的时候要考虑前面两层的节点。因此，相当于增加了一层节点的信息量，于是也会更倾向于得到正确的结果。看一些明

```
['yi', 'zhi', 'piao', 'liang', 'de', 'xiao', 'hua', 'mao']
['一', '直', '漂', '亮', '的', '小', '花', '猫']
['yi', 'zhi', 'ke', 'ai', 'de', 'da', 'huang', 'gou']
['一', '只', '可', '爱', '的', '大', '黄', '狗']
['ji', 'qi', 'xue', 'xi', 'ji', 'qi', 'ying', 'yong']
['机', '器', '学', '习', '及', '其', '应', '用']
```

显有提升的实例：

这实际上就是增加上下文信息量之后获取到的正确率提升。第一个例子难免令人感觉有些奇怪；但是，从语法和语义上讲，实际上都是说得通的句子了。在 1.3 中，引入了一个权值μ。不妨将λ定为上一节中获得的最优值 0.91，然后去获取这种情况下的最优权值μ（这样获得的结果虽不保证最优，但至少不会差）。



从中获取到了两个较优的μ值：

$\mu = 1.8$时，句正确率为 0.5832，字正确率为 0.9036，取到了字正确率的极大值；

$\mu = 6.1$时，句正确率为 0.5850，字正确率为 0.9027，取到了句正确率的极大值。

可以看出这相比二元模型有着极大的进步（$\mu = 0$时，是二元模型）

但是仍然存在一些问题，例如：

```
['ru', 'guo', 'pin', 'yin', 'shou', 'zi', 'mu', 'da', 'xie', 'hui', 'zen', 'me', 'yang']
['如', '果', '拼', '音', '首', '字', '母', '大', '协', '会', '怎', '么', '样']
```

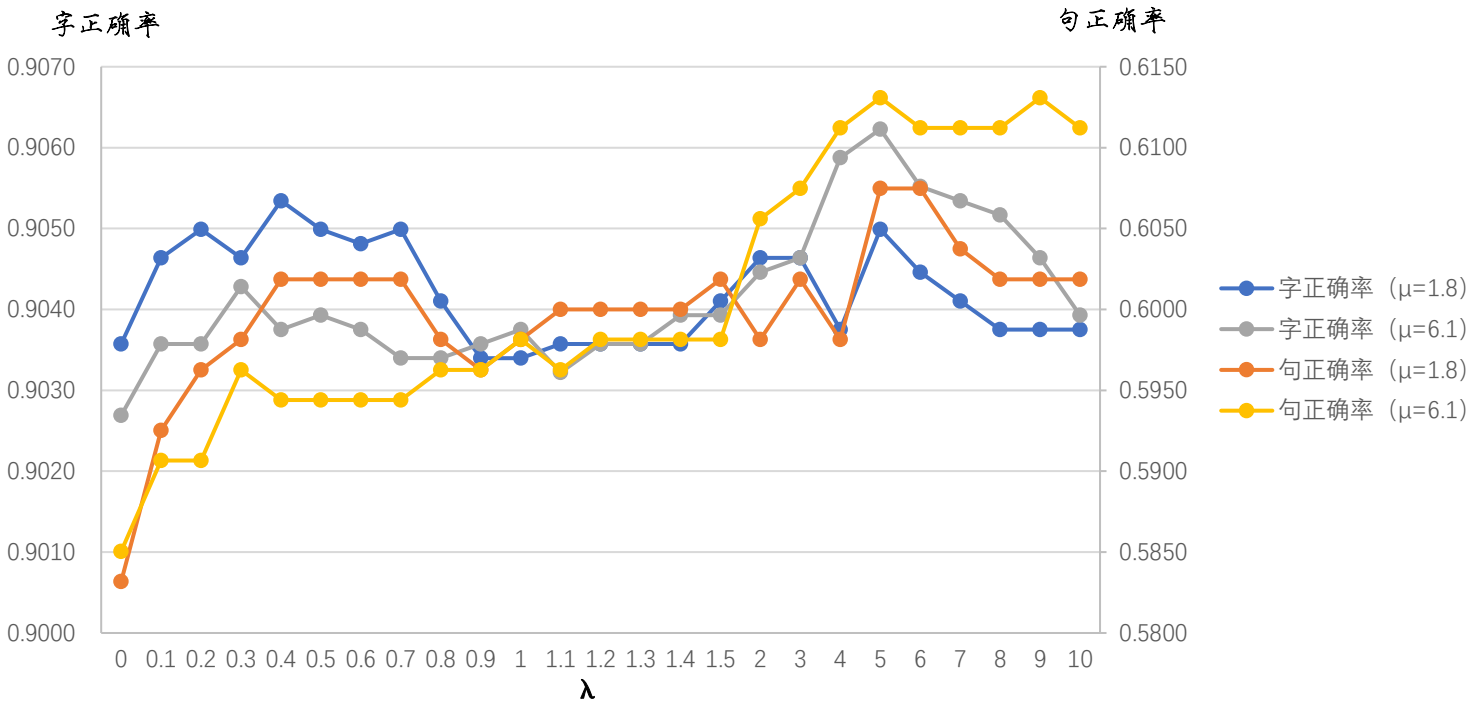
此处的错误，原因应当是在考虑 xie 这个读音时，只考虑了前面两个字的信息，即“母大写”，这个三元词的出现显然是不合理的。但是如果考虑四元词，“字母大写”这个词的出现就变得合理了。因此，如果结合四元模型，正确率应当会进一步提升，见下一节。

2.3 四元模型的实例与性能分析

上述问题在引入四元模型之后得到了解决：

```
['ru', 'guo', 'pin', 'yin', 'shou', 'zi', 'mu', 'da', 'xie', 'hui', 'zen', 'me', 'yang']
['如', '果', '拼', '音', '首', '字', '母', '大', '写', '会', '怎', '么', '样']
```

同样地，进行性能分析；这次选取 2.2 节中所获取的两个极值点来进行φ的最优值选取。



上表虽看似杂乱，但极值很明显：在$(\lambda, \mu, \varphi) = (0.91, 6.1, 5)$时取到最大值，字正确率为 0.9062，句正确率为 0.6131。

3 实验总结

在本实验中，使用 viterbi 算法，采用了基于字的二元、三元、四元模型实现了拼音输入法，并且对三种模型进行比较，并且尝试找出了一个较优的参数配比，使得句正确率达到 60%，字正确率达到 90%（采用网络学堂的测例）。

4 实验收获

1. 对 python 的使用更加熟练。在实验中，需要使用正则进行匹配来提取句子，在进行对比时还需要将结果输出到 excel 中便于观察，使用计时器，在一个 py 文件中调用另一个 py 文件并且获取其输出，这都是非常有益的实践。

2. 效果与时间、空间的权衡。以往，面临的问题常常是时间与空间的这一对矛盾，但是在这次实验中，还面临着效果与这两者的矛盾，也即，如果要将二元模型改

良至三元、四元模型，这势必意味着引入更多信息量，也就需要更多的预处理，读取处理好的文件也会更加费事。权衡无处不在。

5 改进方案

进行分词，使用基于词的模型，应该能进一步提升性能；此外，如果要严格地找到最短距离，那么实际上使用 viterbi 算法是不够的，本实验实际只找到了一个相对短的距离，而不能保证最短。如果保证最短，那么效果应该进一步提升；但是耗时也会增加。

6 附：

（1）目录结构树

```
—myInputMethod：实验报告，原始数据（.xlsx）
  ├—bin：可执行文件
  ├—data：输入与输出
  ├—preprocessing：预处理需要的文件
  |   ├—corpus：语料（gbk 编码）
  |   └—res：对语料作处理之后的文件
  |       └—sentence：语料作处理之后的句子
  └—src：源文件
```

（2）命令行运行源代码的实例：

在 src 文件夹下：

```
python pinyin.py ./test/input.txt ./test/output.txt
```

在 bin 文件夹下：

```
pinyin ./test/input.txt ./test/output.txt
```

（3）由于大小限制，没有上传语料以及使用字的三元、四元模型时处理语料后的文件。如果有需要，可以在清华云盘上下载：

https://cloud.tsinghua.edu.cn/d/02498b84f633479fa209/