拼音输入法实验报告

张驰 2022010754 zhang-ch22@mails.tsinghua.edu.cn

一、实验环境

python 3.9.12
ujson 5.4.0（用于快速读取json文件）
无其他需要额外下载的库。

二、语料库和预处理

本次实验的语料库采取新浪新闻2016年的新闻作为训练语料，共477866650字。

在数据预处理中，我们整理了所有单字、二元词、三元词的出现频率，将他们纳入到四个 `json` 文件中： `words.json` ， `biwords.json` ， `triwords.json` ， `tri_words_2.json` 。其中，为了输入法能够方便地提取词汇信息，我们把文件格式设置如下：

`words.json`：

```
{
    "a": {
        "count": 90423,
        "阿": 75141,
        "啊": 14156,
        ...
    },
    "ai": {
        "count": 266502,
        "爱": 173646,
        "埃": 16669,
        ...
    },
    ...
}
```

其中， `"count"` 记录了该拼音出现的总次数，在计算概率时需要使用这个数据。

二元词和三元词的记录方式类似。以 `tri_words.json` 为例：

```
{
    "a a a": {
        "count": 39,
        "啊 啊 啊": 39
    },
    ...
}
```

由于 `tri_words.json` 文件过大，且语料库包含的很多三个连续字并不是真正意义上的“三元词汇”，我们把 `tri_words.json` 进行了缩减，删除了所有的出现不多于15字的三元词，大大减少了文件大小，得到 `tri_words_2.json` 。

三、基于二元模型的拼音输入法

3.1. 思路、推导、实现

对于给定拼音串 $y_1y_2...y_n$，我们希望找到一串汉字 $w_1w_2...w_n$ 使得 w_i 的拼音为 y_i 且条件概率 $p(w_1...w_n|y_1...y_n)$ 最大。**由于我们接下来讨论的概率都是在 $y_1y_2...y_n$ 作为给定条件的条件概率，我们在写法上把这个条件省去**，把上面的概率写作 $p(w_1...w_n)$。我们使用下面的条件概率公式计算此概率：

$$p(w_1w_2...w_n) = p(w_1)p(w_2|w_1)p(w_3|w_1w_2)...p(w_n|w_1...w_{n-1})$$

在二元语法中，我们认为语言的基本组成元素长度都是两个字及以下，不存在超过三个字的字间相关性，每一个字只与它的前一个字有关。那么有：

$$p(w_1w_2...w_n) \approx p(w_1)p(w_2|w_1)p(w_3|w_2)...p(w_n|w_{n-1})$$

通过统计语料库中所有字和“双字”出现的次数可以得到频数 $N(w_i), N(w_iw_{i+1})$，再除以它们的拼音的频数 $N(y_i), N(w_iw_{i+1})$ 可以得到再对应拼音下的条件概率 $p(w_i), p(w_iw_{i+1})$。使用 $p(w_{i+1}|w_i) = p(w_iw_{i+1})/p(w_i)$ 即可得到条件概率。

计算出所有需要的二元条件概率之后，我们需要找到一种方式来找到最小的乘积。我们可以考虑，把每一个拼音 y_i 对应的字 $w_i^1...w_i^{k_i}$，视为一“层”节点。这些“层”会构成一个网络 $\{\{w_i^j\}_{j=1}^{k_i}\}_{i=1}^n$。任意相邻两层之间的节点 $w_i^{j_i}$ 和 $w_{i+1}^{j_{i+1}}$ 都对应着一个二元条件概率 $p(w_{i+1}^{j_{i+1}}|w_i^{j_i})$。我们希望找到一个路径 $w_1^{j_1}...w_n^{j_n}$，使得这个路径上的所有二元条件概率乘积最小。将所有的条件概率取对数，那么最小乘积就变成了最小和，即最长路径问题。

我们使用基于贪心算法的Viterbi算法来解决这个问题。对于任意层数 i，设全局最长路径到达了 $w_i^{j_i}$，那么它一定是连接前 i 层的一个最长路径。因此，我们只需要对每一层的每一个节点，存储到达此节点的最长路径。在计算下一层的最长路径时，根据上一层的最长路长度和新的对数概率计算新的最长路，这样迭代地计算完 n 层节点，得到全局最长路径和最大。

在代码实现里，我们定义 `Bi_Net` 类来完成网络的存储和最长路的查找。对每一串拼音，我们根据每一个拼音对应的可能的字创建网络，在构造函数中，在每一个节点中存储最长路在上一层经过的节点 `prev` 和到此节点的最长路径长度。随后，我们再使用一个函数 `get_max_sentence`，先找到最后一层中最长路最长的节点，再从后向前依次选出这些 `prev`，完成最长路的选择。

3.2. 实验效果

基于二元模型的输入法的实验效果如下：

```
句准确率：39.72%
字准确率：85.42%
训练时间：47 min（与三元语法模型的总训练时间）
生成测例总时间：10.8秒（平均）
单句平均生成时间：0.022秒
```

3.3. 好、坏例子分析

好例子：

> 因为天气刚刚好

语句常见，没有生僻字，没有多音字。相邻两字都是自然的衔接。全句符合二元语法，每个字基本只和前、后字有关，与更远的字没有显著语义联系。

> 我想吃冰激凌

没有生僻字和多音字。除了“冰激凌”外基本都符合二元语法，而“冰激凌”中的任何一个双字都没有别的常见二元词与之竞争。

坏例子：

> 握好害怕

典型的由于二元语法的局限性带来的问题。“握好”相比于“我好”是一个更频繁的双字，如下：

```
"wo hao": {
    "握 好": 2046,
    "我 好": 702,
    ...
}
```

尽管“我好害怕”显然是比“握好害怕”更为合理的一个句子，但是由于二元语法输入法只能同时看到相邻的两个字，是无法同时看到“害怕”部分和“wo”拼音的。它只能根据“wo hao”这一个“双拼音”决定“wo”的选择，因而选出了频数更高的“握好”。

> 我从未见过犹如此后烟雾持之人

同样的问题。二元语法无法同时看到“厚颜无耻”四个字，而只能两个字两个字看。因此，“此后”、“烟雾”作为频数更高的双字胜出。

> 青藏大甩卖

多音字问题。由于“藏”也有读音“cang”，导致“青藏”在我们的预处理中也会被作为“qing cang”的对应双字进入中间数据表。又由于这个词的高频出现，使得“清仓”的概率被一个错误读音的词超越，带来识别错误。由于我们没有双字的拼音表，这个问题较难解决。

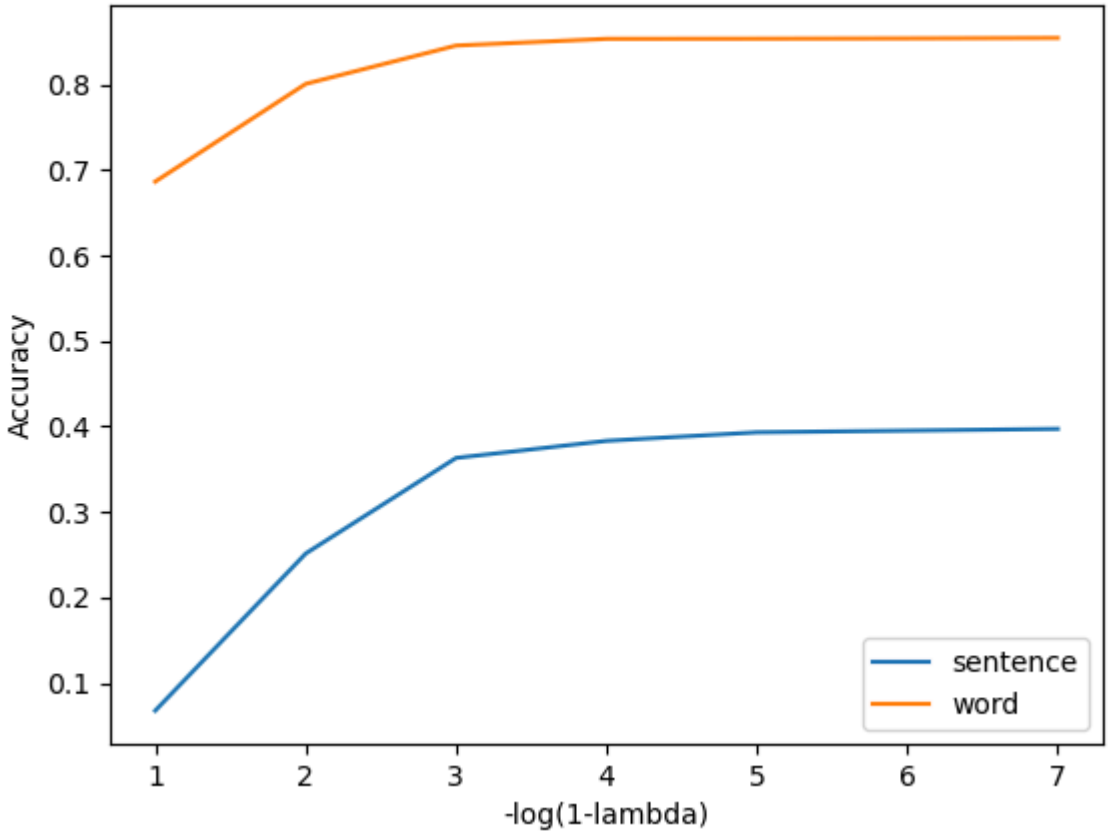
> 现的创业委办二中到崩促（先帝创业未半而中道崩殂）

文言测例/网络用语测例，不在我们的训练数据范围内，无法有效识别。

整体来看，我们基于二元语句的输入法擅长处理无多音字、生僻字，且语法结构较为简单的句子。

3.4. 参数选择

在二元语法的输入法中，我们引入了一个参数 λ 来控制条件概率中一元和二元概率的比例。即：$p(w_i|w_1w_2...w_{i-1}) \approx \lambda p(w_i|w_{i-1}) + (1-\lambda)p(w_i)$，而非全用二元条件概率。这样一方面可以避免二元词表中查找不到词从而出现对 0 概率取对数的情况，另一方面，汉语中有一些字可能不是作为任何二元双字的一部分出现，此时计算其一元概率更为科学。然而经过测试我们发现准确率随着 λ 的变化规律如下：



注意横坐标为 $-\log(1-\lambda)$，是对数坐标。可以看到，随着 λ 的增加，准确率始终在上升。这说明汉语其实并没有什么一元特质。输入法对前后语义信息关联的提取越好，准确率就越高。

3.5. 时空复杂度

时间复杂度：

设拼音串长度为 n，平均每一个拼音对应的字符数量为 k，则输入法构造的网络大小为 $n \times k$。在执行Viterbi算法的过程中，我们需要遍历所有相邻层之间的所有可能的双字，确定他们的条件概率。因此，时间复杂度为 $O(nk^2)$。

经过统计，测例中的句子平均长度约 10.4 个字，拼音表中一个拼音对应的汉字数目的平方平均值为 24.5 个（即 $\bar{k^2} = 24.5^2$）。测例共有 501 个句子，需要的总计算次数（在数量级上）约为 $501 \times 10.4 \times 24.5^2 = 3 \times 10^6$ 次计算。通过上网查询，python代码一秒钟大约能够进行 10^6 次计算。故理论上我们需要的计算时间应该是 3 秒的数量级。在实际测量中（见前），平均用时大概是 10 秒左右。考虑到我们的复杂度分析还没有考虑常数，这个估算是很合理的。

空间复杂度：

我们构建的网络的空间复杂度为 $O(nk)$。然而，由于我们的输入法需要完整读入整个记录一元和二元字频率的中间文件，并且全部存储在内存里，所以至少要占用与这两个文件大小相当的空间。这个空间（80M）远超过 $O(nk)$ 量级。实际测量中，程序运行确实占用约80M空间左右，与理论估计一致。

四、基于三元模型的拼音输入法

4.1. 算法介绍

在三元模型中，我们认为每个字不只和前一个字，而是与前两个字有关。即条件概率公式应当改写为

$$p(w_1w_2...w_n) \approx p(w_1)p(w_2|w_1)p(w_3|w_1w_2)...p(w_n|w_{n-2}w_{n-1})$$

因此，我们需要记录所有“三字”出现的频率，计算三元条件概率，最终找到最大的总条件概率。

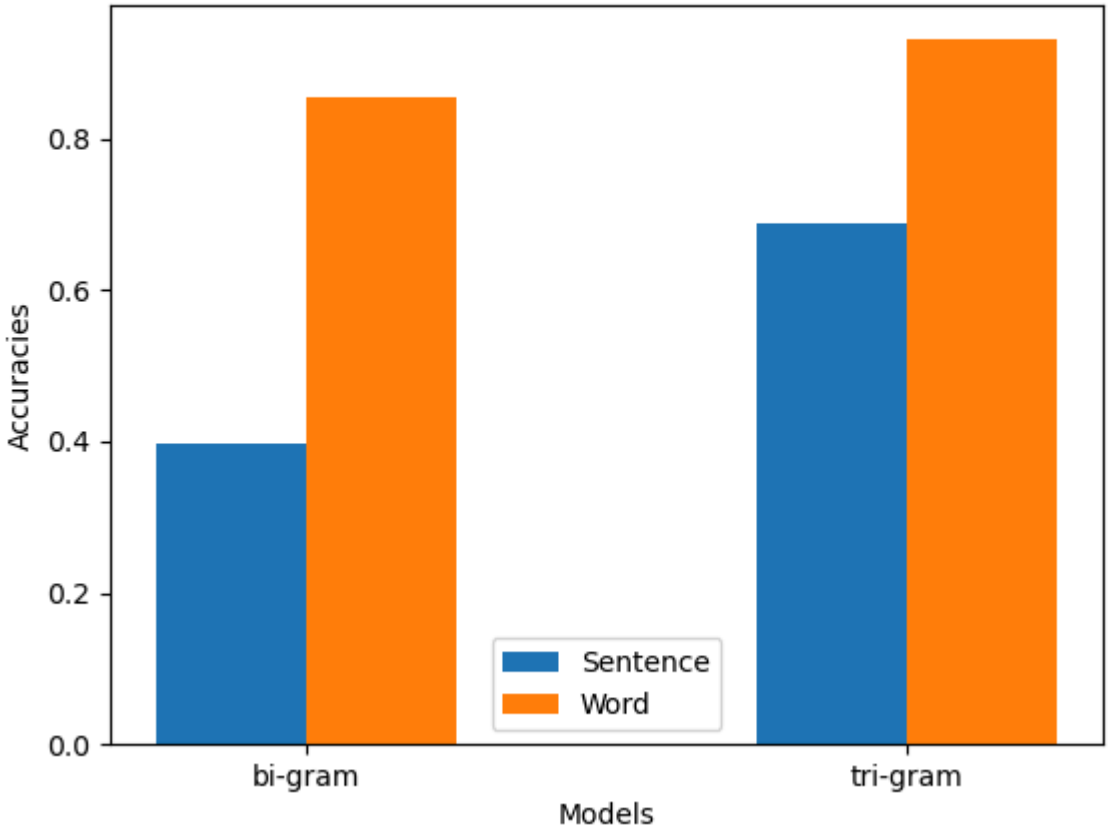
在实现中我们仍然和二元模型一样建立网络。然而，由于每一个字的选取与前两个字有关，所以不能直接使用Viterbi算法寻找最长路径。为此，我们在每一个节点上存储一张表，记录这个节点分别与上一层的所有节点连接起来时，到达这两个节点的最长路径。即在 w_i 中保留 $\{p(w_{i-1}^j w_i)\}_{j=1}^{k_{i-1}}$。在计算下一层节点 w_{i+1} 时，我们遍历 $\{w_i^l\}_{l=1}^{k_i}$，对于每一个 w_i^l，我们计算 $p(w_{i+1}w_i^l) = \max_j\{p(w_{i+1}|w_i^l w_{i-1}^j)p(w_i^l w_{i-1}^j)\}$，并依此更新 w_{i+1}^k 的存储表。最终，仍然是选择最长路最大的尾层节点，然后反向地选择最长路。

4.2. 实验效果、参数选择、对比分析

基于二元模型的输入法的实验效果如下：

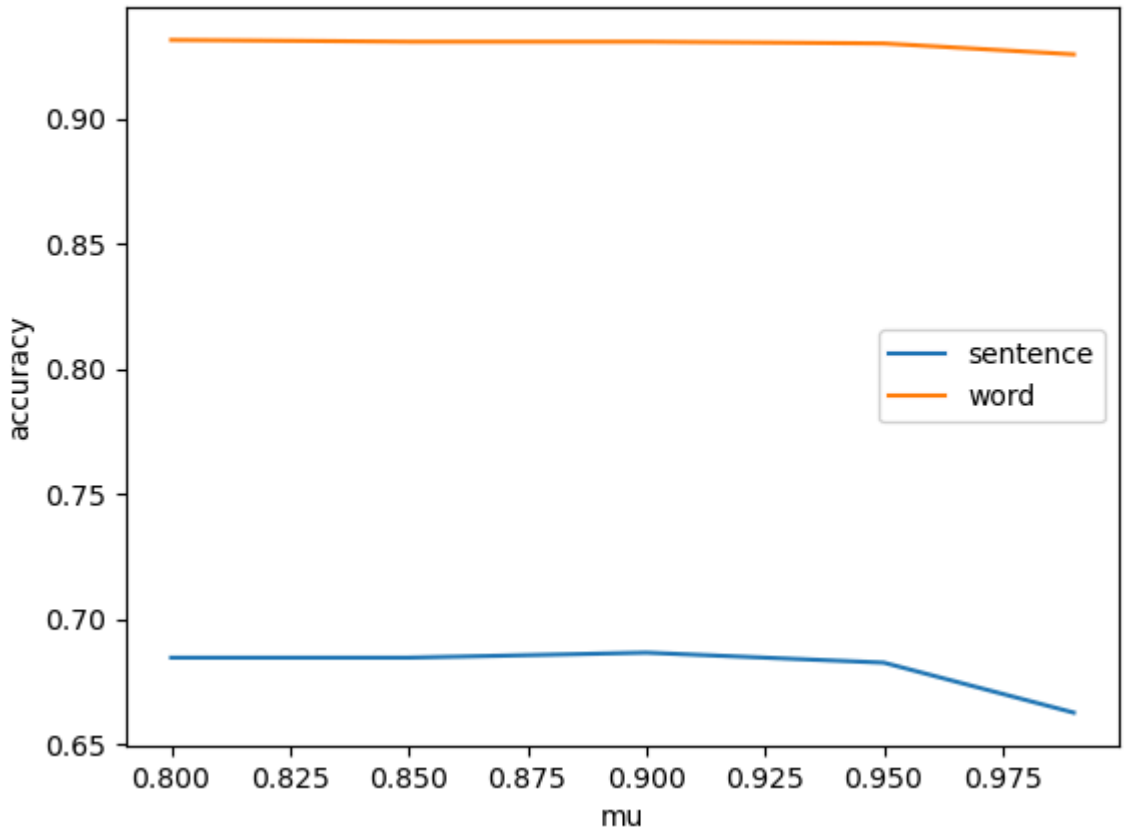
句准确率：68.66%
字准确率：93.06%
训练时间：47 min（与二元语法模型的总训练时间）
生成测例总时间：约800秒（平均）
单句平均生成时间：1.6秒

与二元语法相比，三元语法的输入法准确率大大提高。如图：



与二元语法类似，在实现中，三元语法我们也没有完全使用三元条件概率估计，而是与二元条件概率做了加权平均。具体来说，我们使用 μ 控制三元条件概率和二元条件概率的比例，并延续了二元语法中的 λ 参数。即$p(w_i|w_1w_2...w_{i-1}) \approx \mu p(w_i|w_{i-2}w_{i-1}) + (1-\mu)(\lambda p(w_i|w_{i-1}) + (1-\lambda)p(w_i))$ 。

参数 μ 代表在某些情况下，我们采用二元词而非三元词对进行语义理解。我们测试了不同的 μ 对准确率的影响，结果如下：



可以看到，μ 变化带来的准确率变化并不大，除了在 $\mu = 0.99$ 的时候略有下降。这说明，汉语中是有一定的二元语法成分的。在引入三元语法的时候，最好也保留一定的二元条件概率，在遇到二元词语的时候做出正确判断。

对于上一部分中二元模型输入法无法良好应对的坏测例，三元语法输入法完成情况如下：

> 我好害怕
> 我从未见过有如此厚颜无耻之人
> 清仓大甩卖
> 先的创业为搬而中到崩促

可见，三元语法的输入法能够很好处理二元语法带来的上下文语义信息不足够的问题，甚至多音字问题也得到了一定的解决。然而由于训练语料有限，文言文和生僻字仍然无法解决。

之前我们考虑过删除一部分 `tri_words.json` 中出现次数较少的三元词，以减少文件大小。然而，我们发现，删除后，准确率大大下降了。这样处理后的准确率降低到约 56%。这可能是因为：1. 局部的低频率词汇在很多语句中有可能出现。2. 删除大量的低频率词汇影响到条件概率的正确计算。

4.3. 时空复杂度

时间复杂度：

三元语法的输入法相对于二元语法时间复杂度大大增加。这是因为在相邻两层之间的遍历中，我们需要遍历本层所有节点与前**两层**的所有节点的所有连接。因此，时间复杂度为 $O(nk^3)$。

统计得到，拼音表中一个拼音对应的汉字数目的立方平均值为 31.1 个（即 $\bar{k^3} = 31.1^3$），则总计算次数约有 $501 \times 10.4 \times 31.1^3 = 1.6 \times 10^8$ 次。按照python语言 10^6 次每秒的运算速度，大约需要 160 秒左右的时间。这与实际测量的 800 秒左右的时间在数量级上相符，考虑到复杂度的常数，可以认为估算合理。

空间复杂度：
与前面的二元讨论类似，网络的空间复杂度约为 $O(nk^2)$（因为每一个节点存储了前面一整层节点的信息），然而这远远小于中间文件的大小。实际内存空间占用几乎完全由中间文件大小决定，总共约需要1.8G内存。实际结果与理论估计基本一致。

五、感受和建议

和许多其他课程的大作业不同，本次大作业没有实验框架，而是要完全自己完成整个工程结构，挑战更大一些，但是完成后较有成就感，尤其是输入法能够正确进行拼音解析的时候。不过目前算法离真正的输入法软件还有很大差距，准确率不够高，且反应速度和空间占用太大，建议可以稍微普及一下目前的输入法软件算法（）。