拼音输入法大作业

吴佳启 2022010869 计26

一 实验环境

1. 一元二元语法处理时，使用的小规模数据，可在本地进行，实验环境如下：

```
Distributor ID: Ubuntu
Description:    Ubuntu 22.04.4 LTS
Release:        22.04
Codename:       jammy
```

2. 当使用到三元语法时，数据规模太大（可达GB量级），使用服务器运行，具体环境如下：

```
Distributor ID: Ubuntu
Description:    Ubuntu 22.04.3 LTS
Release:        22.04
Codename:       jammy
```

3. 使用到的python相关库：tqdm jieba pypinyin collections re json sys，在linux系统下均可使用类似如下命令安装：

```
pip install tqdm
```

二 语料库的使用和数据预处理

语料库

使用语料库为下发的新浪新闻2016年的新闻语料库、微博情绪分类技术评测中通用训练集的微博语料库，经测试发现使用新浪新闻语料库中的某一部分文件后效果就可以达到相当客观的水平，后续的上升趋势较为缓慢。

数据预处理

1 开始的尝试

一开始时我准备将数据根据“，。”等标点符号划分，然后一句一句进行批量处理，后来发现这样的for循环对时间要求太高了，于是我准备优化对每一句的处理，于是我使用

jieba库进行了类似如下的划分后，根据单词长度分类，但最终效果显示句正确率仅达百分之二十，我猜测可能是由于jieba的划分并不完全正确导致的错误。

```
words = jieba.lcut(sentence)
    for word in words:
        if len(word) == 1:
        elif len(word) == 2:
```

2 方法更改

考虑到之前所提的巨大时耗，我准备使用正则表达式进行处理，直接找到文章中的所有汉字，然后拼接，这是由于考虑到句首和句尾可能也有一定联系。 我将得到的句子看做一个string，然后每一位、每两位、每三位统计词语的出现频率，将结果存储到字典里，最后根据写入的汉字，通过“拼音汉字表”查询对应的拼音，根据拼音分组后得到统计结果。

```
with tqdm(total=len(chinese_text), desc="TASK2: Processing sentences") as pbar:
    for i in range(len(chinese_text) - 2):
        bigram = chinese_text[i] + ' ' + chinese_text[i + 1]
        chinese_bigrams[bigram] += 1
        unigram = chinese_text[i]
        chinese_unigrams[unigram] += 1
        trigram = chinese_text[i] + ' ' + chinese_text[i + 1] + ' ' +
chinese_text[i + 2]
        chinese_trigrams[trigram] += 1
        pbar.update(1)
```

3 出现问题

在这时我发现使用当前的数据集测试时会出现“KeyError”，仔细研究发现出现了多音字的情况，比如“长”,如果用之前读取的方法，那么“长”会被读作“chang”和“zhang”中的一个，这样就会导致出现在查询另一个时无法查到结果 为了处理这个情况，我只能采用蛮力的枚举方法，即在建立字典时，一个字可以到多个读音，获取一元数据时，将所有读音都添加到字典中，这样就可以避免出现“KeyError”的情况。 在我只读入了新浪新闻语料库的一个文件“2016-04.txt”时，同时使用二元模型时，句准确率已经达到34.9%，字准确率也在80%以上，可以说是已经比较不错了。

4 优化方案——多音字处理

但多音字的处理始终是个问题，于是我想到了使用“pypinyin”库来协助我，但我发现“pypinyin”库也不是一定准确，而且可能会出现“lue”和“lve”的混淆情况导致再次出现

“KeyError”，于是我的做法是先试用“pypinyin”库获得一个可能正确的拼音test_pinyin，然后再所有多音字的可能组合里查找，如果存在即使用test_pinyin，反之则枚举所有可能的多音字增加出现次数。 这样仅仅是一个文件+二元，最终的句正确率就可以达到百分之四十一，极大提高了准确率。

```
test_pinyin = lazy_pinyin(bigram[0]+bigram[2])
    test_pinyin1 = test_pinyin[0]
    test_pinyin2 = test_pinyin[1]
    if test_pinyin1 in pinyins1 and test_pinyin2 in pinyins2:
        pinyin_pair = f"{test_pinyin1} {test_pinyin2}"
        if pinyin_pair not in output_data_bigram:
            output_data_bigram[pinyin_pair] = {"words": [], "counts": []}
        output_data_bigram[pinyin_pair]["words"].append(bigram)
        output_data_bigram[pinyin_pair]["counts"].append(count)
```

5 方法对比——三元优化

基于服务器的性能，我准备尝试利用正则表达式将句子划分后处理数据，即划分为一句一句后再获取词频，结果显示如下（二元语法）：（二元参数为0,9,0；三元为0.9,0.9）

数据类型	按句子划分	整体读入
小规模（500句）	19.36	20.56
中等规模（1个文件）	41.20	40.96
大规模（多个文件）	41.52	41.52

三元时：

数据类型	按句子划分	整体读入
小规模（500句）	21.56	20.96
中等规模（1个文件）	53.69	53.09
大规模（多个文件）	63.67	61.88

在时间上，直接读入的速度大于句子划分的速度，大概在1.25倍左右。 由此可见，在二元语法上，当数据规模比较小时，使用逐句读入有利于准确率的提高，但是当数据规模足够大后，二者准确率接近，这也是因为句末和句首连接的错误词语出现的概率逐渐减少导致的，与我们的预期相符合。 但是在三元中，由于句子的划分对三元数据的影响还是很大的（比如两个句子长度为5和7，划分后得到的三元词数量为3+5=8，但不划分可

以得到10个！），所以在三元语法上，通过划分最终句子的准确率可以得到不少的提高，达到63.67.

三 基于字的二元模型的拼音输入法

1 相关概念

公式推导

1. 根据贝叶斯公式，我们有$P(S|O)=\frac{P(O|S)P(S)}{P(O)}$，其中O是指当前给定的拼音，S时指它对应的句子，所以我们需要寻找的是使这个概率最大的S，同时$P(O)$为常量，$P(O|S)$再不考虑多音字的情况下接近于1，因此问题转化成了求$P(S)$最大。

2. 而$P(S)$的意义又是这是一句话的概率，即我们拥有了一些汉字计算他们组合在一起是一句话的概率,即

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

但这个十分难处理，我们可以考虑二元语法：即一个字的出现只与它前面的字有关，这样就可以将问题转化为求$P(w_1w_2)$，即

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

这样大大简化了计算量，我们只需对这个式子求最大值即可。但是乘法的运算不是很方便，于是我们要进行取对后转化为加法，然后再取一个负号，即求：

$$min(\sum_{i=1}^{n}-log(P(w_i|w_{i-1})))$$

Viterbi算法

1. Viterbi算法通过在所有可能的隐藏状态序列中找到具有最大概率的那个序列来解码。这个最大概率的隐藏状态序列通常被认为是生成观察序列的最可能路径，通常应用于隐马尔可夫模型。

2. 具体来说，我们可以这么刻画这个模型：一共有n列，每一列的元素个数不一样，我们要求经过每一列到达最后一列花费最小，我们用$w_{i,j}$来表示第i行第j个位置。 如

果我们用$Q(W_{i,j})$来表示起点到$w_{i,j}$的最佳距离，用$D_(W_{i-1,j}, W_{i,k})$来表示$w_{i-1,j}$到$w_{i,k}$的花费，那我们可以得到如下公式：

$$Q(W_{i,j}) = \min(Q(W_{i-1,j}) + D_{(W_{i-1,k},W_{i,j})}) \qquad \text{if} \qquad \text{i} = 0$$

基本思路

1. 首先利用之前清洗后得到的数据获取各个一元二元字出现的词频。这样我们在计算概率的时候，可以用频率来近似模拟，比如$P(B|A) = frequency(AB)/frequency(A)$
2. 接下来则使用Vertbi算法，把每一列点看做一个拼音，每一列点中的各个点则是这个拼音对应的汉字，两个点之间的花销即二元语法所计算的代价，这样我们就可以求得每一列点中最佳路径。
3. 最后我们根据标记的最佳路径，一步一步往前回溯，这样就得到了最终的答案。

2 实现过程

数据处理

首先，我们调用数据处理的文件，从原语料库中获得中间数据，将中间数据存储为我们需要的json格式。 然后运行主程序后，从这些中间数据中获取我们需要的信息，包括拼音对应的汉字、一元二元三元词语出现的次数等。 获取这些数据后，我们将他存储为dict形式，以二元数据为例：

```
data2 = {
    "chang zhang": {
        厂长 1;
        场长 2;
    },
}
```

这么做是方便后续的读取，比如我需要“厂长”出现的次数，我只需要使用`data2.get(pinyin_together, {}).get(character_together, 0)`即可获得，注意这里也加上了一个默认值属性，避免找不到一些生僻字出现报错。

训练过程

获得相关数据后，我们只需要实现Vertbi算法即可，枚举每一列点，对每一个点来说，先计算出他前面选择哪个点的代价最小，然后记录下来，这样在回溯的过程中可以找到对

应的汉字。

```
if local_cost + matrix[rows_now-1][j].cost_all < matrix[rows_now][i].cost_all:
    matrix[rows_now][i].cost_all = local_cost + matrix[rows_now-1][j].cost_all
    matrix[rows_now][i].front = matrix[rows_now-1][j]
```

优化

注意到课上所提及的平滑处理，即：

$$P(w_i|w_{i-1}) = LAMBDA * P(w_i|w_{i-1}) + (1 - LAMBDA) * P(w_i)$$

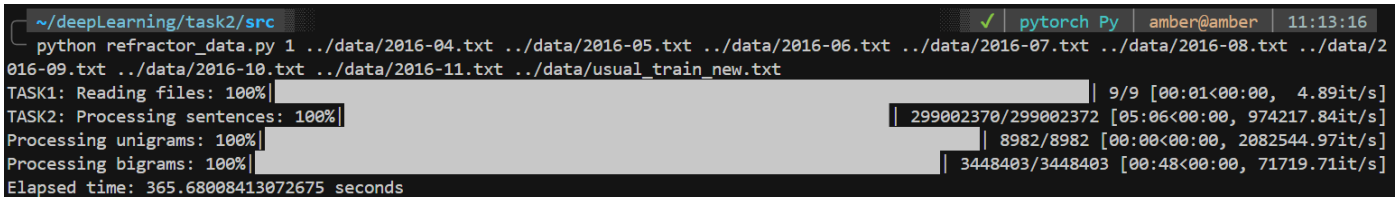
其中LAMBDA为在0-1之间的参数，这样解决了$P(w_i|w_{i-1})$可能为0的问题。

3 实验效果

1. 训练时间，由于本地压力太大，在服务器上运行，此时的命令为：

```
python refractor_data.py 1 ../data/2016-04.txt ../data/2016-05.txt ../data/2016-
06.txt ../data/2016-07.txt ../data/2016-08.txt ../data/2016-09.txt ../data/2016-
10.txt ../data/2016-11.txt ../data/usual_train_new.txt
```

其中1表示只获得一元、二元语法，后面的内容是需要读取的语料，本处实例是把这些语料文件放置在了data文件夹下。 获取所有一元二元语法所用时间为6min左右：



在此处运行结束后，会在src文件夹下产生1_word.txt和2_word.txt，分别存储了一元二元语法，即所需要的中间文件。

2. 测试时间 使用命令为：

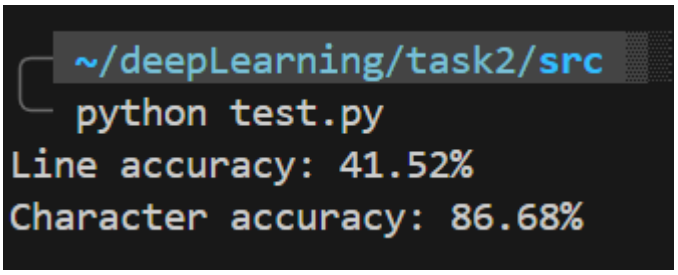
```
python pinyin.py ../data/input.txt ../data/output.txt 0.9 0 1
```

其中0.9表示二元语法平滑参数，0表示三元语法的平滑参数，1表示使用的模型为二元语法。

在服务器上运行，所用很短为3s左右，几乎可以忽略不计，生成一句话的平均时间为3/500=0.006s。

```
~/deepLearning/task2/src                                  ✓ | 6m 6s | pytorch Py | amber@amber | 11:19:26
python pinyin.py ../data/input.txt ../data/output.txt 0.9 0 1
Processing data: 100%|                                                          | 501/501 [00:01<00:00, 318.28it/s]
41.52
Elapsed time: 3.183031427208334 seconds
```

3. 准确率： 在基于二元的模型上，加上平滑参数的处理，句准确率可达百分之四十一，字准确率接近百分之八十七，相当可观。



4. 案例分析 优秀案例：

- 一个自认为潜力不凡的人通过艰苦卓绝的努力
- 最后成为了一个在脸书写代码的普通人
- 大家都喜欢智能体比赛
- 这种情况会对航母打击群的人员造成影响
- 希望自己今后少一些三分钟热度

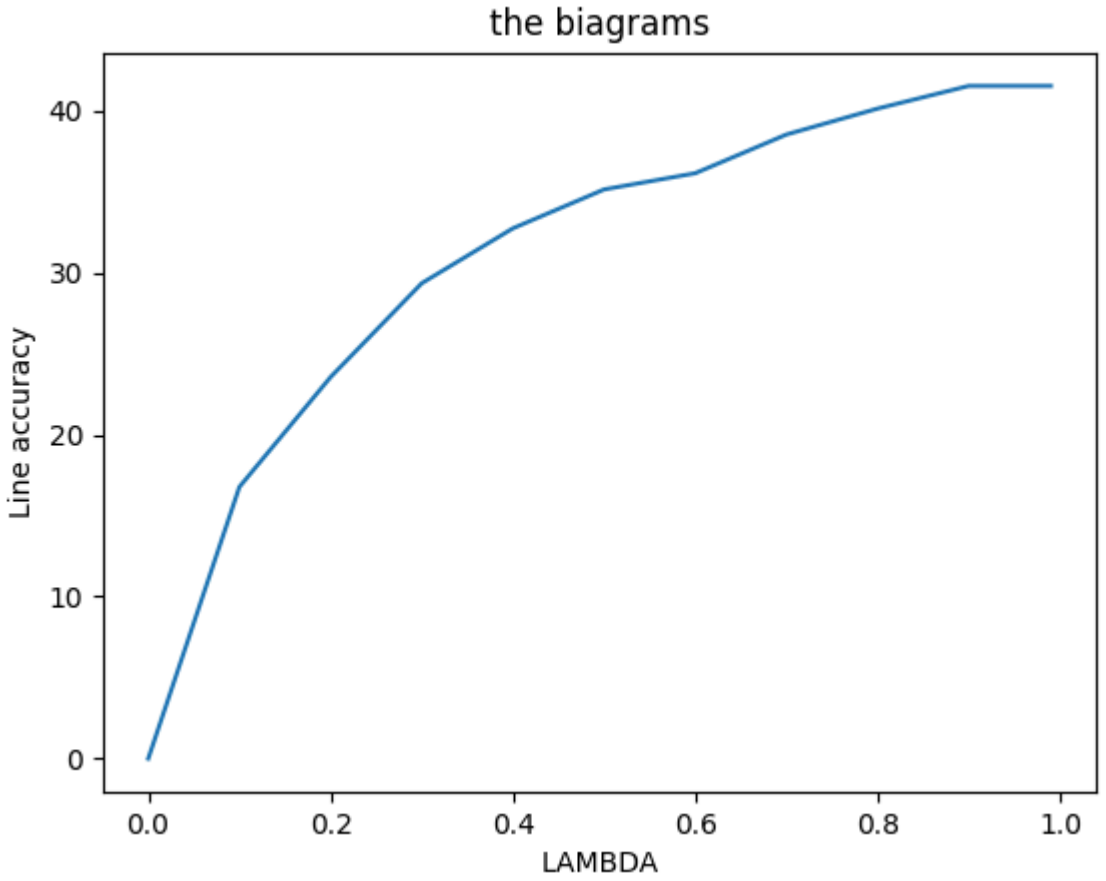
错误案例：

- 深受广大女性的欢迎**合**好评
  深受广大女性的欢迎**和**好评
- **假如**给我三天光明
  加入给我三天光明
- 武汉**零**新增病例给世界以希望
  武汉**领**新增病例给世界一希望
- **青花瓷**真正成为中国瓷器的主流
  清华慈真正成为中国瓷器的主流
- **却有大把**时间用来追剧和攻略
  确有大巴时间用来追剧和攻略
- 梦想就是一种让你**感**到坚持就是幸福的东西
  梦想就是一种让你**赶**到坚持就是幸福的东西

原因分析： (1) 语料库的语料问题，比如“清华”一词出现的频率太高，导致让青花瓷等无法被正确识别。 (2) 二元语法的局限性：比如“欢迎和好评”，在二元语法中“和”只考虑与前一个字的关系，那“迎合”一词也会有着很高的频率，导致最终出现错误，这也激发了我后面对三元语法的尝试。

4 参数选择

在二元模型中，参数的选择即平滑参数的选取，由于字准确率始终较高不作考虑，我们只考虑句准确率，通过对LAMBDA的选取，可以得到如下图像：



5 时空复杂度分析

1. 计算次数估计： 对于二元模型，假设句子平均长度为m，句子总数为n，每个拼音在一元词汇表中对应$O(k_1)$个，每一组二元拼音在拼音词汇表中查找需要$O(k_2)$次（虽然使用了dict，可减小这个k_2的值），在算法执行的过程中，计算每一列的一个点，我们都需要对前一列的点进行一次遍历，同时在这次遍历中，我们会对每一个二元可能查询频率，所以在一次遍历中，我们需要计算$O(k_2 * k_1^2)$次，最后最短路的查询较为简单，为$O(k_1 + m)$，可以得到总的时间复杂度为$O(mn(k_2 * k_1^2) + n * (k_1 + m))$， 但是算法的压力不在此处，而是在于获取一元、二元词的词频，每一次的查找词频会消耗大量的时间。
2. 空间复杂度估计：算法运行的过程中只是会新增m列，每列n个点，这么看空间复杂度为$O(mn)$，即每一列点的存储空间。但同时考虑到一元、二元、三元词的存储，还需要大量额外的空间，不妨设其分别为$O(N_1)$、$O(N_2)$、$O(N_3)$，则总的空间复杂度为$O(N_1 + N_2 + N_3 + mn)$，其中mn可以被忽略。
3. 计算量的估计 粗略估计整个样本：

- 一元词汇长度为16800，key的个数为406，考虑到key占据行，同时每一个汉字会以word和count出现两次，估算每个拼音对应的汉字个数大约为19，这与网上查询得到的20十分接近。
- 二元词汇长度为6539778行，key的数量为143631，估算可得每个词对应的数量为23左右。
- 输入501句，每个句子长度大约为11。 那么带入时间复杂度中的计算，大约计算次数为：$501 * 11 * \ (23 * 19 * 19) \ + 501 * (19 + 11) \approx 46000000$，考虑服务器的性能，估计的计算时间应该为1s左右，但实际却在3s，我个人认为应该是与远程服务器的连接、读取文件的时间导致的。

四 三元模型的实现

基本思路

1. 基于之前二元模型的局限性，我们考虑增加一个三元模型，其中三元数据的获取方式还是和数据处理时一样。
2. 与二元模型不同之处在于公式的计算，我们新的公式如下：

$$
\begin{aligned}
P(w_i|w_{i-1}w_{i-2}) = \ & \text{ALPHA} \times P(w_i|w_{i-1}w_{i-2}) \\
& + (1 - \text{ALPHA}) \times (\text{LAMBDA} \times P(w_i|w_{i-1}) \\
& + (1 - \text{LAMBDA}) \times P(w_i))
\end{aligned}
$$

这里我们还是使用了平滑处理，使用ALPHA和LAMBDA两个参数，更好地刻画了三元语法。 3. Vertbi算法： 三元模型的计算会稍有不同，我的第一反应是进行三次for循环，但发现这样已经违背了我们的初衷：因为根据数学归纳法，前一列的点已经找到了到他花费最小的那个点了，因此我们还是只需要两次for循环，即枚举前一列的点，同时在计算中加入前一个点对应的花费最小的点，这样就可以求得最佳路径。

```
for i in range(len(matrix[rows_now])):
    for j in range(len(matrix[rows_now-1])):
        matrix_front = matrix[rows_now-1][j].front
        character12 =matrix[rows_now-1][j].character + matrix[rows_now]
[i].character
        character012 = matrix_front.character + matrix[rows_now-1][j].character +
matrix[rows_now][i].character
        character01 = matrix_front.character + matrix[rows_now-1][j].character
        unigram_probabilty = get_unigram_probabilty(pinyin2, i)
        biagram_probabilty = get_bigram_probability(pinyin1, j, pinyin12,
character12)
        trigram_probability =
```

```
get_trigram_probability(character01,pinyin01,character012,pinyin012)
        local_cost = cost(unigram_probabilty, biagram_probabilty,
trigram_probability)
        if local_cost + matrix[rows_now-1][j].cost_all < matrix[rows_now]
[i].cost_all:
            matrix[rows_now][i].cost_all = local_cost + matrix[rows_now-1]
[j].cost_all
            matrix[rows_now][i].front = matrix[rows_now-1][j]
```

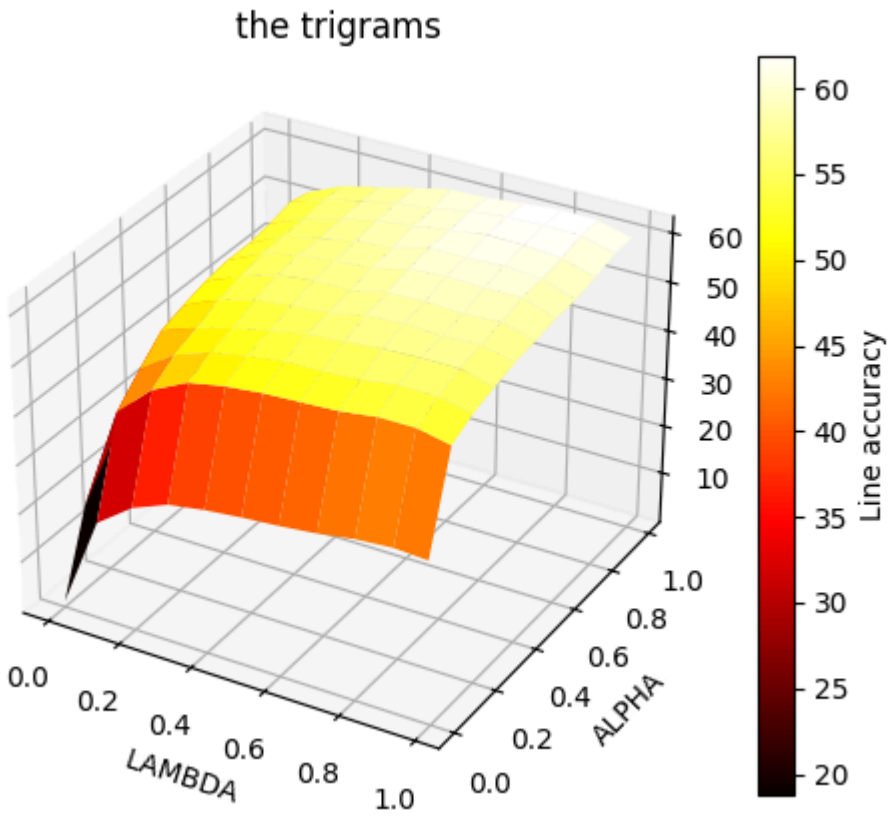
实验效果

1. 训练时间 由于三元数据种类太多，最终的文件很大，训练时间也很长，在服务器上也需要30min左右的时间。
2. 使用命令：

```
python pinyin.py ../data/input.txt ../data/output.txt 0.9 0.9 2
```

其中前两个参数为LAMBDA和ALPHA的取值，后一个参数表示使用模型2——三元语法。 3. 执行时间： 由于三元文件实在太大（在主机上训练得到的大概在1.7G左右），在读取文件的时候需要大量的时间，即使在服务器上总时间大概也需要在40-50s左右

4. 准确率分析 通过调整ALPHA和LAMBDA的取值，我们得到了如下的训练热力图：



不难发现，当ALPHA和LAMBDA均大于0.3以后，最终的句准确率已经超过百分之十五十，当参数选择为ALPHA=0.9，LAMBDA=0.9时，句准确率达到了62%，而字准确率达到了91%，可见三元模型的准确率还是相当高的。 5. 逐句处理优化 在清洗数据时加上逐句处理对三元语法模型的准确率有很大的帮助，此时当ALPHA=0.99，LAMBDA=0.9，准确率最高可达64% 6. 句首优化处理 考虑到在处理第一个字时，我们之前采用的是读取整个语料库中所以一元概率，这样有失偏颇，于是我只读取每句话的第一个组新增了一部分数据，用这部分数据处理第一个字的概率，但很可惜，最后的优化效果并不显著：

数据类型	特殊处理	不特殊处理
三元	63.67	64.07

因此我在最终代码里并未处理第一个字。 7. 案例分析 好的案例：

- 深受广大女性的欢迎和好评
- 青花瓷真正成为中国瓷器的主流
- 智能拼音输入法
- 寻寻觅觅冷冷清清凄凄惨惨戚戚

错误案例：

- 中国**是**人民民主专政的社会主义国家
  中国**诗**人民民主专政的社会主义国家

- 管教他倒戈卸甲以礼来降
  管教他到个鞋架椅里来相
- 计算机组成原理是必修课
  计算机组成员历史必修课
- 这世上只有一种成功就是能够用自己喜欢的方式度过自己的一生
  这是上只有一种成功就是能够用自己喜欢的方式度过自己的医生

原因分析：即是是三元语法也难以顾及到整个句子的语义，比如最后一个错误案例的“一生”和“医生”，同时也是基于语料库对部分文言文语料的缺少，导致在文言文拼音转化时出现了大问题。

五 优化方案总结

1. 处理数据读入尝试jieba库先分词——失败：jieba库与实际的语义还是有较大差距，导致准确率低下；
2. 多音字处理：先使用pypinyin库，查询pypinyin库生成的拼音是否在可能的组合中，避免了多音字的影响，对二元三元都有大帮助。
3. 逐句处理优化：在清洗数据时加上逐句处理对三元语法模型的准确率有很大的帮助，准确率最高可达64%；
4. 平滑处理：调整LAMBDA，ALPHA参数。
5. 句首优化处理：只读取每句话的第一个组新增了一部分数据，用这部分数据处理第一个字的概率，但很可惜，最后的优化效果并不显著。
6. 三元语法：以空间、时间换正确率。

六 其他可能指标

1. 时空复杂度，最主要的为运行时占据的空间，比如当三元模型逐渐增大时准确率自然可以提升，但是空间的损耗也是个问题。
2. 如果实现多个候选词等功能，可以考虑候选词的数量、排序等。

七 对实验的感受和建议

我认为本次实验相当的有意义！ 首先，我之前对python进行数据操作并不熟悉，在本次实验的帮助下掌握了python从文本中截取数据、转化为json文件的方法，学会了这个实用技巧。 其次是对算法和模型的理解，在本次实验中，我第一次接触到了Viterbi算法，

并用它完成了拼音输入法的设计。 然后是在模型建立后参数的调整，比如在我刚开始建立二元模型后，对LAMBDA的取值范围进行调整，最终得到了一个相当高的准确率。 最后是三元模型的处理，让我意识到大数据在重要的同时也会耗费很多的时间，但同时新的尝试也会带来显著的收益。 至于对实验的建议，我认为可以给我们讲解除了基于字的拼音输入法以外的别的方法，作为选做进行性能对比，如果仅仅是字的拼音输入法可能较为局限，同时可以结合实际的拼音输入法，比如添加模糊匹配、自动补全等，作为实验二。