人智导 - 拼音输入法 Report

经*12*-计*18* 张诗颖 *2021011056*

1 完成情况

- **训练语料**
  - [x] 【必做】新浪新闻2016年的新闻语料库（`corpus/2016-XX.txt`，其中`XX`为2至11月）
- **基本要求与选做内容**
  - [x] 【必做】基于字的二元模型（两个版本）
  - [x] 【选做】基于字的三元模型（两个版本）
  - [ ] 【选做】基于字的四元模型
  - [x] 【选做】基于词的二元模型（基于分词的改进尝试）
  - [ ] 【选做】基于词的二元模型
- **最终准确率**：
  - 二元模型（选取最佳版本结果）：
    （词）：0.8425
    （句）：0.3553
  - 三元模型（选取最佳版本结果）：
    （词）：0.8799
    （句）：0.5090

2 文件结构与使用方法

2.1 文件结构

```
pinyin
|--corpus // 存放语料库的文件夹
|--data // 基础数据表
    |--look-up_table.txt // 给定的拼音汉字表
    |--pinyin_mapping.txt // 字典格式，表示给定拼音不同字出现的概率，格式如{"qing": {"清": 0.9, "情": 0.1}}
|--refactored // 生成数据表
    |--SingleCntStat.txt // 给定拼音，对应汉字的统计次数，格式如{"qing": {"清": 0.9, "情": 0.1}}
    |--BiCntStat(ch-ch).txt // 给定汉字，后继出现汉字的统计次数，格式如{"清": {"华": 900, "划": 100}}
    |--BiProbStat(ch-ch).txt // 给定汉字，后继出现汉字的统计频率，格式如{"清": {"华": 0.9, "划": 0.1}}
    |--BiProbStat(ch-py-ch).txt // 给定汉字，后继出现汉字的统计频率，格式如{"清": {"hua": {"华": 0.9, "划": 0.1}}}
    |--BiProbStat(dpy-dch).txt // 给定两个拼音，两者对应汉字的频率，格式如 {"qing hua": {"清华": 0.9, "情话": 0.1}}
    |--InitialBiProbStat(dpy-dch).txt // 同上，但统计样本仅为每一个语料自然分词的的首字词
    |--TriCntStat(dch-py-ch).txt // 给定两个汉字及后继拼音后，后继汉字的统计次数，格式如{"清华": {"da": {"大": 900}}}
```

```
    |--TriProbStat(dch-ch).txt // 给定两个汉字，后继汉字的统计频率，格式如{"清华":
{"大": 0.9}}
    |--TriProbStat(dch-py-ch).txt // 给定两个汉字及后继拼音后，后者汉字的统计频率，格式如
{"清华": {"da": {"大": 0.9}}}
    |--InitialTriProbStat(tpy-tch).txt // 同上，但统计样本仅为每一个语料自然分词的的首字
词
|--main.py // 整个源代码的入口程序，可以根据用户选项选择使用的模型、检验准确率等功能
|--preprocessor.py // 预处理语料库，生成统计文件的程序
|--graph_generator_bi.py // 基于字的原版二元模型
|--graph_generator_bi_plus.py // 基于词的原版二元模型（基于分词的改进尝试）
|--graph_generator_bi2.py // 基于字的二元模型
|--graph_generator_tri.py // 基于字的原版三元模型
|--graph_generator_tri2.py // 基于字的三元模型
|--validator.py // 字句正确率计算
|--input.txt // 用户提供，输入拼音文本
|--output.txt // 模型生成，输出汉字文本
|--std_output.txt // 用户提供，标答汉字文本
|--README.md
```

2.2 使用方法

本次作业共实现了**五个模型**可以选择

> bigram：原二元模型（$P(华|清hua)$版）
> bigram+：改进版二元模型（$P(华|清hua)$ + 分词版）
> bigram_2：二元模型（$P(华|清)$版）
> trigram：原三元模型（$P(大|清华da)$版）
> trigram+：三元模型（$P(大|清华)$版）

```
python3 main.py [-h] [-m {bigram,bigram_2,bigram+,trigram,trigram_2}][-i INPUT]
[-o OUTPUT][-t][-f STD_OUTPUT]
(example): python3 main.py -m trigram -i "./data/input.txt" -o
"./data/output.txt" -t -f "./data/std_output.txt"
(例子含义)：用三元模型输入"input.txt"，将答案输出到"output.txt"，并与标准答案文
档"std_output.txt"比较，计算正确率

optional arguments:
  -h, --help                                show this help message and exit
  -m, --model {bigram,bigram+,trigram}      choose one model to run
  -i INPUT, --input INPUT                   choose an input file
  -o OUTPUT, --output OUTPUT                choose an output file
  -t, --test                                choose whether to test the result
  -f STD_OUTPUT, --std_output STD_OUTPUT    give a std_output for testing
```

注意：本程序的预处理输入文件编码为***utf-8***（根据语料库决定），其它的所有输入输出文件编码**均**为***gbk***（包括`input.txt`，`output.txt`）

2.3 实验环境与依赖

1. 实验环境版本

```
$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 20.04.5 LTS
Release:        20.04
Codename:       focal

$ python3 --version
Python 3.8.10
```

2. 部分依赖项

```
pypinyin              0.48.0
tqdm                  4.64.1
```

3 基本思路与实现过程

本次实验模型均基于基本的马尔可夫过程（HMM / Hidden Markov Model）实现。

对于输入拼音序列$S = s_1 s_2 \dots s_n$，需要生成对应中文汉字序列$W = w_1 w_2 \dots w_n$，使得W最佳。设使用的模型为k元，根据条件概率公式，有以下公式：

$$P(W) = \Pi_{i=1}^{k} P(w_i | w_{i-(k-1)} \dots w_{i-1})$$

在实际测试过程中，采用**对概率取对数**再相加**取极大值**的方式简化计算，即：

$$w = logP(W) = \sum_{i=1}^{k} P(w_i | w_{i-(k-1)} \dots w_{i-1})$$

3.0 数据预处理

本实验使用的语料库为：新浪新闻2016年的新闻语料库（`corpus/2016-XX.txt`，其中`XX`为2至11月）

数据预处理方法和内容已经在文件结构中说明，后续模型分析中也会详细说明。

3.1 基于字的二元模型

> 预处理依赖文件：
>
> 1. `./data/pinyin_mapping.txt`：字典格式，表示给定拼音不同字出现的概率，格式如`{"qing": {"清": 0.9, "情": 0.1}}`
> 2. `./refactored/BiProbStat(ch-ch).txt`：给定汉字，后继出现汉字的统计次数，格式如`{"清": {"华": 0.9, "划": 0.1}}`
> 3. `./refactored/InitialBiProbStat(dpy-dch).txt`：给定两个拼音，两者对应汉字的频率，格式如`{"qing hua": {"清华": 0.9, "情话": 0.1}}`，样本为每一个语料自然分词的的首字词

1. 采用**平滑处理**后的打分公式：

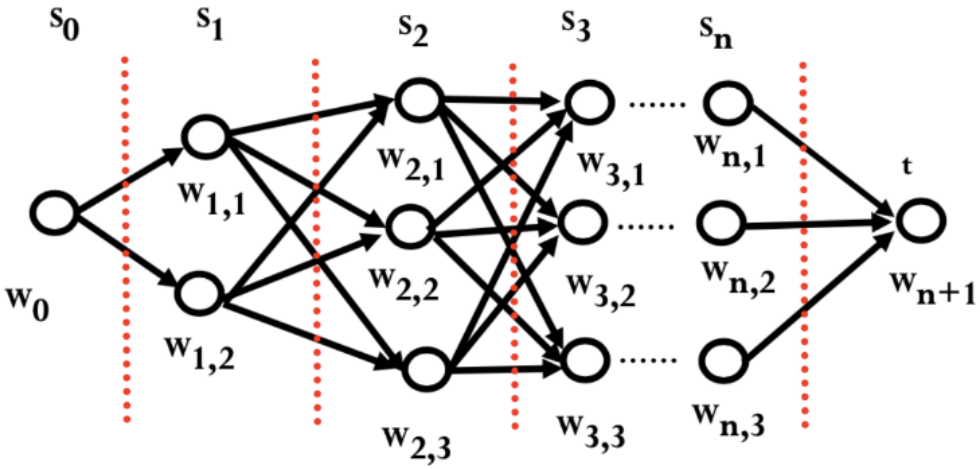
$$P_2(W_i) = \lambda P(w_i | w_{i-1}) + (1 - \lambda) P(w_i | s_i)$$

因此，实际上二元模型是“一元”与“二元”的结合。

2. Viterbi搜索算法

   采用逐层采用得分较高极大值的方式拓展路径。在实际操作过程中，取每层得分前`ALTERNATIVE=50`的路径进行下一层的拓展。

   - 首层的路径拓展来自文件`./refactored/BiProbStat(ch-py-ch).txt`对于首两个拼音-对应汉字的概率排序
   - 其它层的路径拓展来自文件`./refactored/BiProbStat(ch-ch).txt`对于前驱汉字-后继汉字的概率排序
   - `./data/pinyin_mapping.txt`用作单字概率平滑



注意：

在我第一次写二元模型的时候，我采用的概率计算为已知后继拼音的**条件概率**，如在对“qing hua”的`["清"]`路径进行拓展的时候，我计算的打分函数`score`依赖于$P(华|清hua)$。但用于最终的整体最优解选取应该考虑全局量而不仅仅是拼音条件概率，在助教学姐的指点之下，我将打分函数的依赖项改成了$P(华|清)$，这使得二元模型的正确率有了很大的提升。

模型	字正确率	句正确率
二元模型$P(华\|清hua)$版 - bigram	0.6705	0.0898
二元模型$P(华\|清)$版 -bigram2	0.8425 **	0.3553 ***

3.2 基于词的二元模型（基于分词的改进尝试）

此部分基于原二元模型$P(华|清hua)$版

在完成基于字的二元模型后，我发现整个句子靠后字词的准确率很依赖于前驱字词的准确率（特别是当上式λ取较大值时）。但由于中文中分词现象的存在，这种依赖于“字与字之间连接概率”的算法可能会出现较大的偏差，如以下例子：

> wo shang xue qu le
> 错误输出：我上学区了
> 正确输出：我上学去了
>
> qing hua da xue ji suan ji xi
> 错误输出：清华大学技酸及西
> 正确输出：清华大学计算机系

第一个例子判断`qu`时的分数取决于“学”，而“学区”又是概率很大的词，导致了判断错误。第二个例子无法判断出`xue ji`为“学计”这一词语分割，导致后续判断错误。

于是，我进行了一些对此的实验和改进，所谓“基于词”也源于此。

> 增加预处理依赖文件：
>
> - `./refactored/BiProbStat(dpy-dch).txt`：给定两个拼音，两者对应汉字的频率，格式如`{"qing hua": {"清华": 0.9, "情话": 0.1}}`

1. 猜想：如果待判断的相邻拼音概率最高的几个汉字组合出现的概率较小（意味着概汉字组合有很多种可能，较为分散），则推算为此处可能存在**分词**。
2. 参数

   > `BAR = [0.3, 0.2]`
   > `alpha=0.9, beta=0.2`

   `BAR = [0.3, 0.2]`表示：如果以上最大概率的汉字组合（对应拼音设为$s_k s_{k+1}$）大于`BAR[0]`，或者次大概率的汉字组合大于`BAR[1]`，则视为没有分词，按照正常基于字的二元模型进行计算

   否则，按照如下方式进行打分：

   $$score = \alpha \times log(P(w_{k+2}|w_{k+1})) + (1-\alpha) \times [\beta \times logP(w_k) + (1-\beta) \times log(P(w_{k+1}|w_k)]$$

3. 结果对比

模型	字正确率	句正确率
原二元模型（$P(华|清hua)$版）- bigram	0.6705	0.0898
改进版二元模型（$P(华|清hua)$版）- bigram+	0.7056	0.1397

   以上对比为基于相同参数的对应模型正确率对比结果，仅用于横向对比，绝对值可能会随着参数变化而变化

   改进后字正确率略有提高，而句正确率大幅提高，但正确率偏低，仍然不够理想。同时，这部分的结果对比是基于二元模型改进之前，所以其实在改进完二元模型之后参考价值也没有那么大了。算是一次没有太大结果的尝试吧。

3.3 基于字的三元模型

> 预处理依赖文件：
>
> 1. `./refactored/InitialBiProbStat(dpy-dch).txt`：同上，但统计样本仅为每一个语料自然分词的的首字词
> 2. `./refactored/TriProbStat(dch-ch).txt`：给定两个汉字，后继汉字的统计频率，格式如`{"清华": {"大": 0.9}}`
> 3. `./refactored/TriProbStat(dch-py-ch).txt`：给定两个汉字及后继拼音后，后者汉字的统计频率，格式如`{"清华": {"da": {"大": 0.9}}}`
> 4. `./refactored/InitialTriProbStat(tpy-tch).txt`：给定三个拼音，三者对应汉字的频率，格式如`{"qing hua da": {"清华大": 0.9, "情话大": 0.1}}`，样本为每一个语料自然分词的的首字词

1. 采用**平滑处理**后的打分公式：

   $$P_3(W_i) = \lambda P(w_i|w_{i-1}w_{i-2}) + \mu P(w_i|w_{i-1}) + (1-\lambda-\mu)P(w_i)$$

   因此，实际上三元模型是“一元”、“二元”与“三元”的结合。

   - 长度不够的时候退化为二元模型

- 首字词的判断基于`InitialTriProbStat(tpy-tch).txt`或退化的二元`InitialBiProbStat(dpy-dch).txt`决定

2. 结果对比

模型	字正确率	句正确率
二元模型 - bigram_2	0.8425	0.3553
三元模型 - trigram_2		

以上对比为基于相同参数的对应模型正确率对比结果，仅用于横向对比，绝对值可能会随着参数变化而变化

改进后字句正确率均有大幅度提高。

注意：

在我第一次写三元模型的时候，我采用的概率计算为已知后继拼音的**条件概率**，如在对“qing hua da”的`["清华"]`路径进行拓展的时候，我计算的打分函数`score`依赖于$P(大|清华da)$。但用于最终的整体最优解选取应该考虑全局量而不仅仅是拼音条件概率，在助教学姐的指点之下，我将打分函数的依赖项改成了$P(大|清华)$，然而，这样的改变使得三元模型的正确率相较于原版降低，**这点与二元模型并不一样。**

模型	字正确率	句正确率
三元模型（$P(大|清华da)$版）- trigram	0.8799 *	0.5090 **
三元模型（$P(大|清华)$版）- trigram_2	0.8598	0.4072

4 实验结果

模型	字正确率	句正确率
原二元模型（$P(华|清hua)$版）- bigram	0.6705	0.0898
改进版二元模型（$P(华|清hua)$版）- bigram_plus	0.7056	0.1397
原三元模型（$P(大|清华da)$版）- trigram	0.8799 ***	0.5090 ***
二元模型 - bigram_2	0.8425 **	0.3553 *
三元模型 - trigram_2	0.8598 **	0.4072 **

4.1 输出结果效果举例

（基于三元模型 - *trigram*）

- 效果较好的例子

> 人工智能技术发展迅猛
>
> 每隔四年一次的冬奥会在今年召开了
>
> 中国和意大利都是历史悠久的文明古国
>
> 消灭剥削消除两极分化最终达到共同富裕

> 不积跬步无以至千里

- 效果较差的例子

> 该账号开通仅仅四十八小时细分而是救完（吸粉二十九万）
>
> 最后成为了一个在联署（脸书）写代码的普通人
>
> 中国共产党员的出新（初心）和实名（使命）是为中国人民谋幸福为中华民族谋复兴
>
> 四月一日行气散（星期三）
>
> 用于毛毛（勇敢猫猫）不怕困难

其实以上错误例子也能得到一定理解，主要原因应该是语料库词语采样不全或者有偏差，以及部分词语推断涉及更高维度的推断证据（如效果较差例子的第三句，“初心”和“使命”基于前后“中国共产党”和“中国人民”可以推断出来，难以用前后文“员的”和“是为”来推断）

同时，根据检测数据，模型对于数据和专有名词的识别也比较薄弱，如上例第一、二、四句。

4.2 模型间对比（基于“原版”模型）

1. 贪婪的人总是得寸进尺不知满足

模型	output
原二元模型	【谈栏】的人总是【的村级持】【用不满足】×××
改进版二元模型	【探拦】的人总是【德村级持】【用不满足】×××
三元模型	【贪婪】的人总是【得寸及吃】【用不满足】√××

2. 人工智能和机器学习技术发展迅猛

模型	output
原二元模型	【人共治能】和【技企学习】【记述】发展迅猛 ×××
改进版二元模型	【任公职能】和【机器学习】【记数】发展迅猛 ×√×
三元模型	【人工智能】和【机器学习】【技术】发展迅猛 √√√

3. 贪婪的人总是得寸进尺不知满足

模型	output
原二元模型	中国和【一大力】都是【利时有酒】的【问明古国】×××
改进版二元模型	中国和【一大力】都是【历史悠久】的【问明古国】×√×
三元模型	中国和【意大利】都是【历史悠久】的【文明古国】√√√

可以发现，对应改进还是进步比较明显的。尤其是当涉及的词语较长（如“机器学习”、“历史悠久”）的时候，改进版和三元模型的识别能力有所提升。

5 参数选择与性能分析

> **二元模型的参数选择**
>
> $$P_2(W_i) = \lambda P(w_i|w_{i-1}) + (1-\lambda)P(w_i|s_i)$$
>
> 1. ALTERNATIVE（每层拓展路径后加入备选方案的最大数量）= **50**
> 2. PARAMETER（λ） = **0.75**
> 3. sigmoid() 平滑函数：math.log()
>
> **三元模型的参数选择**
>
> $$P_3(W_i) = \lambda P(w_i|w_{i-1}w_{i-2}) + \mu P(w_i|w_{i-1}) + (1-\lambda-\mu)P(w_i)$$
>
> 1. ALTERNATIVE（每层拓展路径后加入备选方案的最大数量）= **40**
> 2. PARAMETER [λ, μ, 1-λ-μ] = **[0.75, 0.2, 0.05]**
> 3. PARAMETER2 [α, 1-α] = **[0.9, 0.1]**
> 4. sigmoid() 平滑函数：math.log()

5.1 参数选择

本部分主要基于三元原版模型的参数选择进行说明

> 所有待确定参数：
>
> 1. ALTERNATIVE：每层拓展路径后加入备选方案的最大数量
> 2. PARAMETER = [λ, μ, 1-λ-μ]：三元模型的平滑权重
> 3. PARAMETER2 = [α, 1-α]：二元模型对应的平滑权重
> 4. sigmoid()：平滑函数

1. **ALTERNATIVE**

> 基于：
>
> PARAMETER = [0.75, 0.2, 0.05]
>
> PARAMETER2 = [0.4, 0.6]
>
> sigmoid() = math.log()
>
> 测试得出

值	字准确率	句准确率
10	0.8793	0.5070
15	0.8795	0.5050
18	0.8797	0.5070
20 ***	0.8799	0.5090
25	0.8795	0.5090
30	0.8786	0.5090

分析：当ALTERNATIVE较少时模型比较“短视”，容易忽略一些后续体现优势的可能性。当ALTERNATIVE较大是产生的干扰和扰动更多，难以精确识别最佳选择。

2. **PARAMETER**

> 基于：
> ALTERNATIVE = 25
> PARAMETER2 = [0.4, 0.6]
> sigmoid() = math.log()
> 测试得出

值	字准确率	句准确率
[0.2, 0.6, 0.2]	0.8701	0.4431
[0.3, 0.4, 0.3]	0.8704	0.4611
[0.4, 0.4, 0.1]	0.8776	0.4850
[0.7, 0.2, 0.1]	0.8793	0.5050
[0.75, 0.2, 0.05] ***	0.8795	0.5090
[0.75, 0.1, 0.15]	0.8776	0.5030
[0.8, 0.15, 0.05]	0.8790	0.5110
[0.9, 0.08, 0.02]	0.8779	0.5070

分析：较大依赖于多词条件概率可以略提高准确率，尤其在于提高句准确率，字准确率的提高不明显。

3. **PARAMETER2**

> 基于：
> ALTERNATIVE = 25
> PARAMETER = [0.3, 0.5, 0.2]
> sigmoid() = math.log()
> 测试得出

值	字准确率	句准确率
[0.9, 0.1]	0.8727	0.4631
[0.8, 0.2]	0.8739	0.4651
[0.5, 0.5]	0.8743	0.4651
[0.4, 0.6] ***	0.8749	0.4671
[0.3, 0.7]	0.8744	0.4671
[0.2, 0.8]	0.8741	0.4671

分析：可能由于三元模型中二元模型参数使用概率不大，调参效果也不显著。

4. **sigmoid()**

我尝试了`tan()`，`probit()`，`logit()`和`log()`等符合在$[0, 1]$区间单调递增特点的其他函数，事实证明`log()`的效果最好。当概率为零的时候，设置`sigmoid(0) = -math.inf`即可。

5.2 性能分析

1. **I/O操作**

   本模型的性能开销主要在于数据预处理文件的I/O操作，下表为三个模型一次完整I/O操作的用时：

模型	I/O 用时
二元模型	1.787164 seconds
三元模型	36.754913 seconds

   这点从数据预处理文件的大小也可以推理得出：

   - `pinyin_mapping.txt`：117KB
   - `BiProbStat(ch-ch).txt`：65.8MB
   - `TriProbStat(dch-ch).txt`：666MB

2. **运行时间**（以测例500句为例）

模型	测例用时
二元模型	9.348216 seconds
三元模型	15.16819 seconds

   均用时较短

6 总结与改进方案

6.1 收获

总体而言，本次实验让我收获颇丰。一方面，这是我第一次上手写python程序，极大程度地提高了我的代码能力；另一方面，这也是我第一次了解和实现马尔可夫过程和拼音输入法的工作原理。

以下是本次作业后我对于人工智能一些认知上的改变：

1. 人工智能本质上还是基于数学原理的数据统计科学。
2. 平滑处理在统计中对于提升准确率很重要
3. 两个版本：带拼音的条件概率（$P(\text{华}|\text{清}hua)$版） or 不带拼音条件概率（$P(\text{华}|\text{清})$版）？其实我觉得两者从数学角度都说的通，二元和三元的不同结果似乎也说明了这个问题。看来还是要实践出真知。
4. （思考）参数对于模型非常重要，如何更有效更省时间的进行参数调节？（毕竟我在本次测例的调节依然是基于控制变量法的手动/脚本测算）

6.2 改进

1. 缩小预处理文件大小，提高I/O效率

   可以直接采用**二进制存储**的方式进行预处理文件存储。推测这样可能可以减小预处理文件大小，提高I/O速度。缺点可能是无法直接读取，调试时可读性较差。

2. 增加更新更全面的语料库