

人工智能课程第一次大作业报告 v3

拼音输入法

第三版改进点：概率转移矩阵（条件概率）三种求法

姓名：沈磊
物理系 2015 级

一、任务描述

拼音输入法可以借助注音符号与汉语拼音这两种汉字拼音方案。汉语拼音输入法的编码是依据汉语拼音方案（汉字的读音）进行输入的一类中文输入法。本次大作业的任务是采用汉语拼音的方案，从输入文件中输入一句一句的汉语拼音，通过程序转化成汉字，存在输出文件中，再通过测试文件中正确的目标汉字进行比较，计算正确率并评价模型。

二、模型介绍

1、字的二元语法模型——一阶隐马尔科夫模型（Hmm）

本次任务的拼音输入法是统计语言模型的一个典型实例。统计语言模型是通过整理大量语料库中的统计数据，实现某些语言操作需求，比如本次作业的拼音输入法。举一个例子，在“wo shi zhong guo ren”这个音节序列中，“wo shi”可能对应了“我是”、“卧室”、“我市”、“卧式”等词，而“zhong guo ren”则可能是“中国人”或者“种果人”，要想得到最可能的汉字句子，就要用到统计语言模型了，我们在这一大堆统计数据中，分别找到词频最大的单词（也即概率最大），如“我是”和“中国人”，句子就可以组合出来了，再取整体概率最大的句子，就是我们猜测的结果。值得注意的是，这种从拼音猜测汉字的方式就是人思考的方式。

为了更加深入地思考，“wo shi zhong guo ren”这个音节序列要想得到“我是中国人”这个汉字句子，整个句子思考的话就是全概率模型。用数学语言表示就是，令 X 为汉字句子， Y 为拼音句子，求

$$X = \max(P(X|Y))$$

满足上式最大概率的汉字句子 X ，就是我们要求的汉字句子。

$$\text{根据贝叶斯公式 } P(X|Y) = \frac{P(X)P(Y|X)}{P(Y)}$$

其中 $P(Y)$ 是拼音的概率，因为我们所输入的拼音是固定的，所以该值也是固定的，取 1。

$P(Y|X)$ 表示的是汉字句子对应的拼音，被称作识别信度。（注意：这个概率不是固定的，因为句子中的汉字可能有多音字。这个概率的意义是，实际考

虑多音字的情况，汉字“朝”是拼音“chao”的概率（“朝”也可以读成“zhao”），本模型把该值设为1，因为我们没有多音字的语料库！这也是本模型的缺点，后面会讨论。）

所以也就是 $P(X) = P(w_1)P(w_2|w_1)P(w_3|w_1w_2) \dots P(w_n|w_1w_2 \dots w_{n-1})$ ，

其中 X 为汉字序列， w_1, w_2, \dots, w_n 对应一个个汉字。意义就是拼音对应一系列同音字（ $w_{11}, w_{12}, w_{13}, \dots, w_{1m}$ ），（ $w_{21}, w_{22}, w_{23}, \dots, w_{2m}$ ） \dots （ $w_{n1}, w_{n2}, \dots, w_{nm}$ ）在汉字序列 X 中组合，我们不仅要考虑 w_1 出现的概率，还要考虑在 w_1 出现的情况下 w_2 出现的概率；接着考虑 w_1w_2 出现的情况下， w_3 出现的概率； \dots ； $w_1w_2 \dots w_{n-1}$ 出现的情况下， w_n 出现的概率。

这就是全概率模型！考虑到随着 n 增加，计算复杂度会几何式增长，而且概率矩阵是很大的稀疏矩阵，计算的性价比比较低，所以二元模型就应运而生了！即每个字的概率只和它的前一个字有关。即：

$$P(X) = P(w_1)P(w_2|w_1)P(w_3|w_2) \dots P(w_n|w_{n-1})$$

求在上述概率最大值情况下，汉字句子由 $w_1w_2w_3 \dots w_n$ 等汉字组成。这种模型也是典型的一阶隐马尔科夫模型（hmm）。

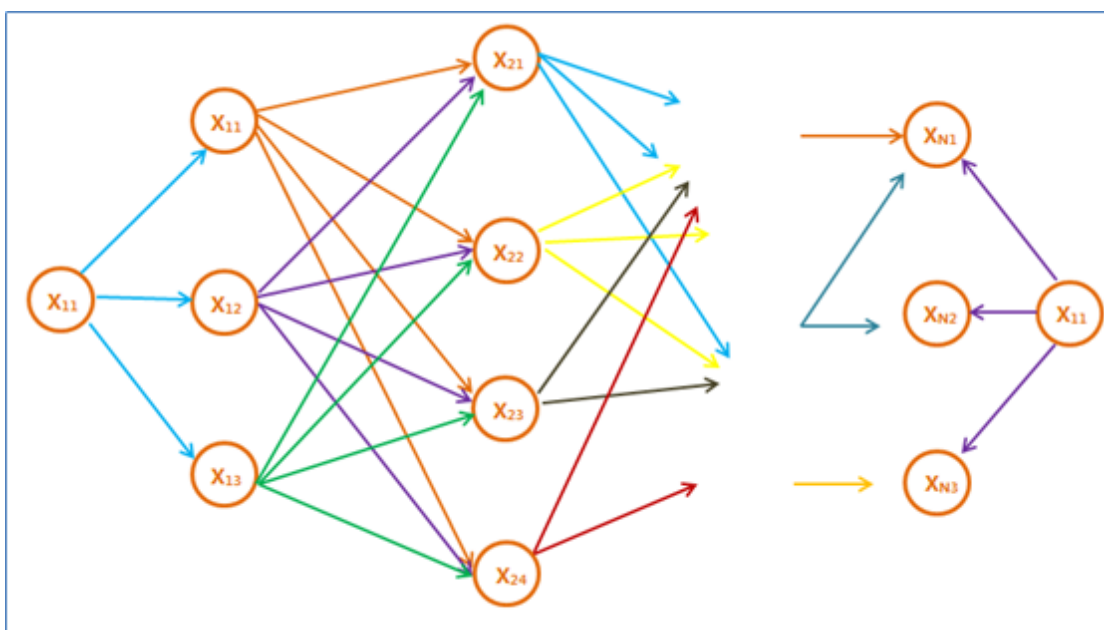
2、viterbi 算法

上述求概率最大值是一阶隐马尔科夫模型中典型的求最佳路径问题。一种常用的动态规划算法，viterbi 算法，是解决该问题的有效手段。该算法是安德鲁·维特比在 1967 年提出来的。该算法可以概括为以下三点：

(1). 如果概率最大的路径 p (或者说最短路径) 经过某个点，比如途中的 X_{22} ，那么这条路径上的起始点 S 到 X_{22} 的这段子路径 Q ，一定是 S 到 X_{22} 之间的最短路径。否则，用 S 到 X_{22} 的最短路径 R 替代 Q ，便构成一条比 P 更短的路径，这显然是矛盾的。证明了满足最优性原理。

(2). 从 S 到 E 的路径必定经过第 i 个时刻的某个状态，假定第 i 个时刻有 k 个状态，那么如果记录了从 S 到第 i 个状态的所有 k 个节点的最短路径，最终的最短路径必经过其中一条，这样，在任意时刻，只要考虑非常有限的最短路径即可。

(3). 结合以上两点，假定当我们从状态 i 进入状态 $i+1$ 时，从 S 到状态 i 上各个节点的最短路径已经找到，并且记录在这些节点上，那么在计算从起点 S 到第 $i+1$ 状态的某个节点 X_{i+1} 的最短路径时，只要考虑从 S 到前一个状态 i 所有的 k 个节点的最短路径，以及从这个节点到 X_{i+1} ， j 的距离即可。



图一、维特比网络

该算法就是本次作业用到的核心算法。

三、实验过程

本次大作业运用的是 python3.7，用的是 python 标准模块。

1、语料数据

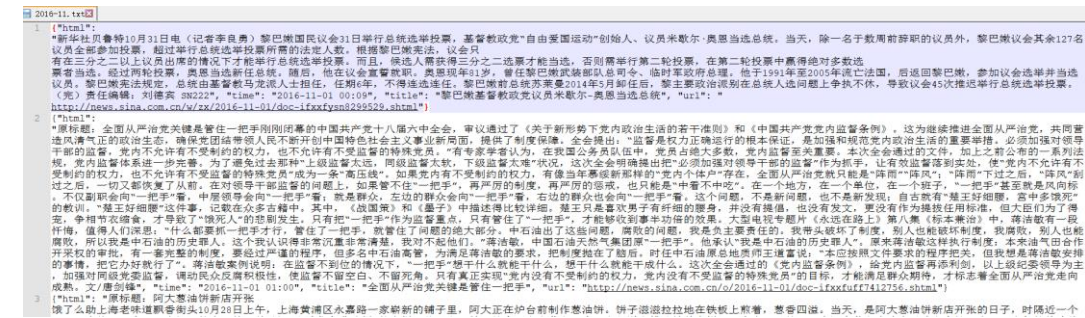
下面介绍一下本次作业所用到的数据：

名称	修改日期	类型	大小
__pycache__	2018/11/19 15:47	文件夹	
__init__	2018/11/17 22:59	Python File	1 KB
2016-02	2017/3/15 9:25	文本文档	120,545 KB
2016-04	2017/3/15 9:25	文本文档	136,175 KB
2016-05	2017/3/15 9:25	文本文档	121,206 KB
2016-06	2017/3/15 9:25	文本文档	124,260 KB
2016-07	2017/3/15 9:25	文本文档	118,608 KB
2016-08	2017/3/15 9:25	文本文档	123,064 KB
2016-09	2017/3/15 9:25	文本文档	113,714 KB
2016-10	2017/3/15 9:25	文本文档	100,363 KB
2016-11	2017/3/15 9:25	文本文档	13,497 KB
hmm_mat	2018/11/19 15:17	Python File	5 KB
py_ch_rel	2018/11/18 0:03	文本文档	20 KB
README	2017/3/15 9:25	文本文档	1 KB
train_str	2018/11/19 15:22	文本文档	837,797 KB
拼音汉字表	2017/3/8 22:06	文本文档	25 KB
一二级汉字表	2017/3/8 21:31	文本文档	14 KB

图二、my_model 模块目录，包含语料数据（train_str.txt），常用汉字数据（一二级汉字表.txt），拼音汉字对应表(拼音汉字表.txt)

上图二文件夹下“2016*.txt”是原始语料数据是爬取的新浪新闻数据，json 格式，gbk 编码。Train_str.txt 是我处理完所有语料文件后存的文本文件，800MB 大小。

除了上述语料外，我们还有 406 个拼音组成的拼音汉字对照表“拼音汉字表.txt”，还有 6763 个常用汉字数据“一二级汉字表.txt”。



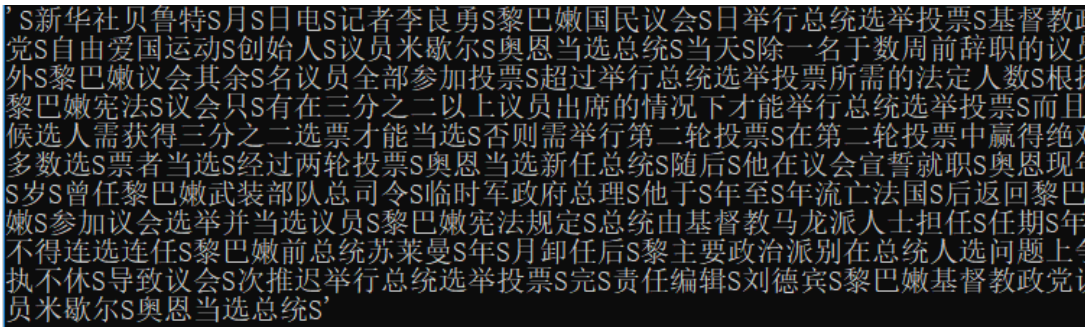
图三、原始语料数据

值得注意，原始语料数据包含很多标点符号、英文字母和特殊字符。这些多余的字符在后期统计词频的时候比较耗时。但是也不能把所有非汉字的字符都剔除，因为这在统计二元汉字的条件概率的时候会不准确（由标点隔开的两个汉字不能当成二元条件概率的贡献）。所以即为了不影响统计二元汉字的条件概率，也为了尽量减小计算量，我利用正则表达式

```
train_str = re.sub( r'[a-z]+|[A-Z]+|[0-9]+|\W+', 'S', data_str)
```

```
train_str = re.sub( r'S+', 'S', train_str)
```

把所有挨在一起的非汉字的字符替换成一个“S”字符，处理成如图四所示。用“S”把句尾和下一句句首汉字隔离开，提高准确率。（这是第二版改进点。）



图四、原始预料数据处理

2、难点分析

由于语料库数据巨大，计算 hmm 模型的三个概率矩阵需要考虑效率问题。在第 4 小节会具体介绍处理庞大语料库的算法。

3、工程结构

本次作业用 python3.7 实现的工程 pinyin2chinese。脚本和模块代码文件有 pinyin.py, hmm_mat.py, viterbi.py 三个。项目最上层目录结构如下图：

40423_1_拼音输入法作业 > 拼音输入法作业 > pinyin2chinese_v3				搜索"pinyin2chi
名称	修改日期	类型	大小	
__pycache__	2018/11/21 14:02	文件夹		
my_method	2018/11/22 13:51	文件夹		
my_model	2018/11/21 17:54	文件夹		
init_dict	2018/11/21 19:28	文本文档	81 KB	
init_dict_p	2018/11/21 19:28	文本文档	193 KB	
init key str	2018/11/21 19:28	文本文档	20 KB	
input	2018/11/20 2:43	文本文档	1 KB	
input test	2018/11/20 1:30	文本文档	6 KB	
pinyin	2018/11/21 16:59	Python File	3 KB	
readme	2018/11/22 0:02	文本文档	1 KB	
test_output	2018/11/20 2:42	文本文档	1 KB	
test output test	2018/11/20 1:29	文本文档	3 KB	
train_trans_mat_dict	2018/11/21 15:20	Python File	2 KB	
trans_mat_dict.json	2018/11/22 11:23	JSON 文件	113,204 KB	
第一次大作业报告v3_人工智能_沈磊	2018/11/22 0:03	Microsoft Word ...	1,537 KB	

图五、最上层目录结构

说明：

pinyin.py: 拼音转换成汉字的脚本文件。

Train_trans_mat_dict.py: 是生成初始概率状态矩阵 (init_dict) 和概率转移矩阵 (trans_mat_dict) 的脚本，并存成文件。由于已经训练完，所以不用再执行该脚本了。

My_Model 模块: 计算一阶 hmm 模型概率矩阵的模块。见图二 hmm_mat.py。

My_method 模块: 实现 viterbi 算法的模块，viterbi.py 在此文件夹下。

Init_dict.txt: 概率初始化矩阵，字典形式，value 是词频。

init_dict_p.txt: 概率初始化矩阵，字典形式，value 是概率。

Trans_mat.dict.json: 是概率转移矩阵，也即条件概率矩阵。

其余黄色方框的 txt 文件都是测试拼音识别的文件。

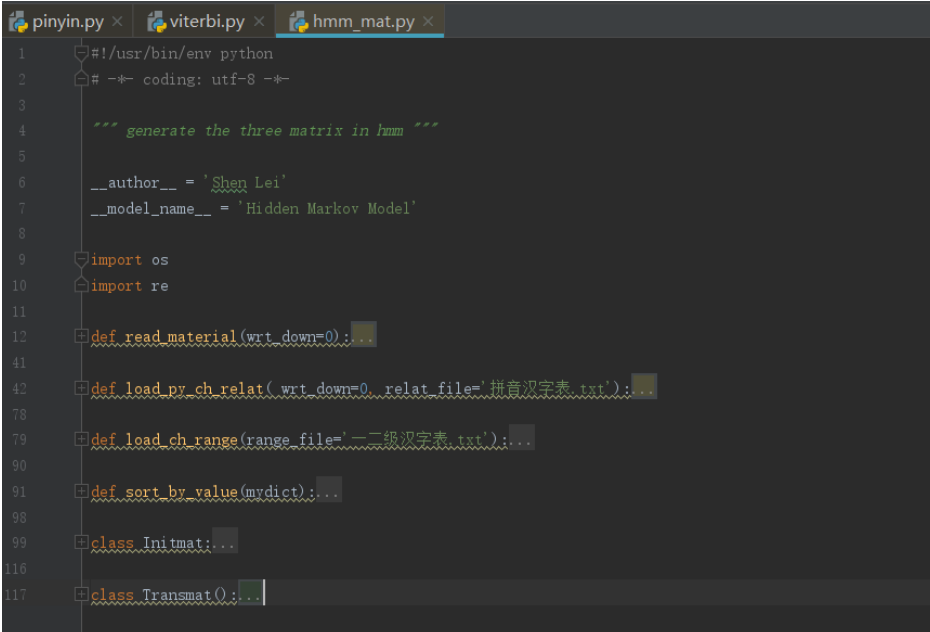
4、实现方法

(1)、计算概率矩阵

首先一阶 hmm 需要计算三个矩阵，初始概率矩阵 (Initmat 类，单个汉字概率矩阵)，概率转移矩阵 (Transmat 类，两个汉字的条件概率矩阵) 和发射

矩阵（注：前面讨论过，本次作业没有多音字语料库，无法计算发射矩阵，这也是本次实验的一个有待改进的地方）。

如下图六，hmm_mat.py 里 read_material 方法是读取语料库并处理的方法，返回 str 型，并存成 “train_str.txt”，方便下次直接读取；load_py_ch_relal 方法是计算或读入拼音汉字对照表的方法，返回拼音汉字对称表的 dict 型，可以实现快速查找。



```
1  #!/usr/bin/env python
2  #-*- coding: utf-8 -*-
3
4  """ generate the three matrix in hmm """
5
6  __author__ = 'Shen Lei'
7  __model_name__ = 'Hidden Markov Model'
8
9  import os
10 import re
11
12 def read_material(wrt_down=0):...
41
42 def load_py_ch_relal(wrt_down=0, relat_file='拼音汉字表.txt'):...
78
79 def load_ch_range(range_file='二级汉字表.txt'):...
90
91 def sort_by_value(mydict):...
98
99 class Initmat:...
116
117 class Transmat():...
```

图六、hmm_mat 模块结构

下面计算初始概率矩阵和概率转移矩阵就是一阶 hmm 中重要步骤（**第三版改进点，三种方式计算概率转移矩阵**）：

➤ 初始概率矩阵：

该矩阵是计算单个汉字的词频统计。

根据下面讲述计算概率转移矩阵（条件概率）三种方法。前两种是求完全的 6763*6763 对二元汉字的概率转移矩阵（条件概率），所以 pinyin2chinese_v1 和 pinyin2chinese_v2 需要的 init_dict.txt 和 init_dict_p.txt 是全部 6763 个汉字的初始概率矩阵（已经做了平滑操作，每个词频加一，可以避免出现 0 词频，对条件概率统计不会出现除以 0 的 bug）。在 800MB 文本做词频统计，耗时大概 30 分钟。

第三种是求概率转移矩阵是（条件概率）稀疏阵，舍去词频为 0 的条目，在后面 viterbi 算法实现部分实现平滑操作（初始概率矩阵对词频为 0 的条目取词频取最小值，概率转移矩阵（条件概率）对词频为零的条目，取前一个汉字初始概率的 0.1 倍）。在 800MB 文本中统计稀疏的初始概率矩阵耗时约 20 分钟。

➤ 概率转移矩阵（条件概率）：

直接地考虑二元语法的条件概率，需要把 6763 个汉字两两匹配，然后在 800MB 文本数据中统计条件概率，这个工作量实在太大了。

汉字 b 在汉字 a 出现的情况下的条件概率应该是：

$$P(b|a) = \frac{\text{number}(ab)}{\text{number}(a)}$$

其中 a 的词频已经在初始概率矩阵中统计过了，可以直接调用，难点在 ab 同时出现的词频统计。这其中有两个地方比较耗时。一是 6763×6763 个汉字匹配，包含先后顺序，二是每次匹配完之后要在 800MB 文本中统计词频。

算法改进：

a) 裁剪语料库词频统计：

我的改进是合理缩减每次词频统计的语料库大小。举个例子，如果要统计“我”和另外 6763 个汉字的词频，我们不用在 800MB 中统计 6763 遍，我们在 800MB 文本中把“我”下一个字符提取出来，并排列组成一个新字符串，然后只要在这个新字符串中统计 6763 次词频。这样一来，比在 800MB 中统计 6763 次词频要快得多。虽然相对速率提高很多，但是绝对速率依然比较慢，只能挂在服务器上慢慢跑了……该方法对应 pinyin2chinese_v1 工程。

```
125 class TransmatO:
126     __table_name__ = 'Initial matrix'
127
128     def __init__(self, train_str, init_dict, init_dict_p, method=0):...
129
130     def genermat(self):
131         trans_mat_dict = {}
132         for x in self.__range_key_list: #6763
133             later_pos = [m.start() + 1 for m in re.finditer(x, self.__train_str)]
134             cut_train_list = [self.__train_str[i] for i in later_pos]
135             for y in self.__range_key_list:
136                 trans_mat_dict[x + y] = 0.99 * cut_train_list.count(y) / self.__init_dict[x] \
137                     + 0.01 * self.__init_dict_p[x]
138
139         return trans_mat_dict
140
141     def calccell(self, input_str, cut_train_list, trans_mat_dict):...
```

图七、裁剪语料库词频统计计算条件概率

b) 动态规划求解条件概率：

依然利用上述裁剪语料库方法统计词频，利用动态规划办法，每次做拼音转化成汉字过程中，遇到新的汉字组合，进行一次词频统计（单次二元词频统计还是很快的），更新到概率转移矩阵 trans_mat_dict 中，并保存到 trans_mat_dict。Txt 中。下次运行脚本拼音转换成汉字过程中，如果字典里已经保存该条件概率，就可以直接用。如果没保存，回到上面继续计算新的条件概率，并保存。

好处：这种计算条件概率的方式，可以避免第一次计算完整的条件概率矩阵，在不断运行时，更新我们的条件概率矩阵的字典。随着运行时间和次数的增加，trans_mat_dict.txt 文件的条件概率会越来越完善，脚本运行时间也会越来越短。

不足：因为每次运行脚本都要计算条件概率，还要读取 800MB 语料库，所以

每次运行脚本，时间会比较长。

```
125 class TransmatO:
126     __table_name__ = 'Initial matrix'
127
128     def __init__(self, train_str, init_dict, init_dict_p, method=2):...
129
130     def genermat(self):...
131
132     def calccell(self, input_str, cut_train_list, trans_mat_dict):
133         if input_str in trans_mat_dict:
134             return trans_mat_dict[input_str], trans_mat_dict
135         trans_dict_p = 0.99 * cut_train_list.count(input_str[0]) / self.__init_dict[input_str[0]] + \
136             0.01 * self.__init_dict_p[input_str[0]]
137         trans_mat_dict[input_str] = trans_dict_p
138         return trans_dict_p, trans_mat_dict
139
```

图八、动态规划计算条件概率方法

c) 求稀疏的概率转移矩阵:

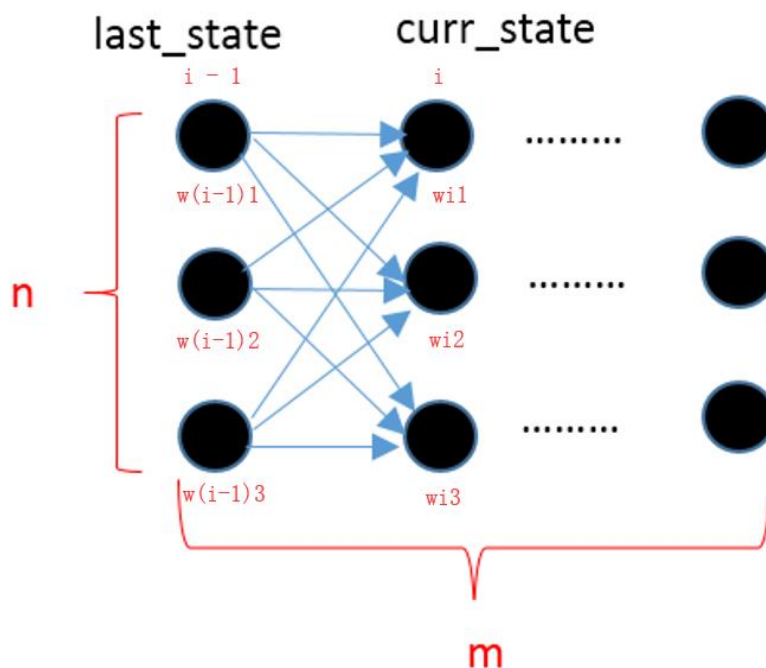
由于 6763 个汉字中很多汉字都比较生僻，在有 255 个汉字在 800MB 文本中词频为 0，所以 init_dict 可以缩减；而且还有更多的两两组合的汉字是词频更少，远远少于 6763*6763 中情况。所有求得的概率转移矩阵（条件概率）更加稀疏，可以更加缩减 trans_mat_dict 的大小。所以在该版本 pinyin2chinese_v3 中，求得的 init_dict 和 trans_mat_dict 是稀疏矩阵，并在 viterbi 算法中进行平滑操作。代码如下图：

```
hmm_mat.py
107 class Initmat:
108     __table_name__ = 'Initial matrix'
109
110     def __init__(self, train_str, range_str, method=2):...
111
112     def genermat(self):
113         init_key = list(self.__range_str)
114         init_value = list(map(lambda x: self.__train_str.count(x), self.__range_str))
115         init_key_r = []
116         init_value_r = []
117         for i in range(len(init_value)):
118             if not (init_value[i] == 0): # 稀疏矩阵
119                 init_key_r.append(init_key[i])
120                 init_value_r.append(init_value[i])
121         init_key_str = ''.join(init_key_r)
122         tot_value = sum(init_value_r)
123         init_value_p = [x/tot_value for x in init_value_r]
124         init_dict = dict(zip(init_key_r, init_value_p))
125         init_dict_p = dict(zip(init_key_r, init_value_p))
126         return init_dict, init_dict_p, init_key_str
127
128 class TransmatO:
129     __table_name__ = 'Initial matrix'
130
131     def __init__(self, train_str, init_dict, init_dict_p, method=2):...
132
133     def genermat(self, init_key_str):
134         trans_mat_dict = {}
135         for x in init_key_str: # 该在词料中出现的字符串
136             later_pos = [m.start() + 1 for m in re.finditer(x, self.__train_str)]
137             cut_train_list = [self.__train_str[i] for i in later_pos]
138             for y in init_key_str:
139                 if y in cut_train_list:
140                     trans_mat_dict[x + y] = cut_train_list.count(y) / self.__init_dict[x]
141         return trans_mat_dict
142
```

图九、两个稀疏概率矩阵的求解

(2)、viterbi 算法实现

根据前面模型介绍分析，求解该一阶 hmm 最优路径问题可以用 viterbi 算法求解。具体过程如下：



图十、viterbi 动态规划计算示意图

根据上述 viterbi 算法思想：序号为第 $i-1$ 的拼音对应一系列同音字 $w(i-1)1, w(i-1)2, w(i-1)3$ ，即图八中的 last_state 这一列。然后是序好后一个第 i 列，这一列同音字 $wi1, wi2, wi3$ ，即状态 curr_state。首先我们知道第 $i-1$ 列每个节点的最佳路径，然后第 i 列的节点的最佳路径一定包含前面第 $i-1$ 列各个节点的最佳路径，所以就转换成 3×3 大小求最佳路径，依次类推可以求得最后一个拼音所有同音字的最佳路径，选出其中最大的路径，就是我们要求的全局最佳路径。即把 n^m 次计算优化为 $(m-1)n^2$ 次计算，计算效率显著提高。

上一小节提到，本次大作业的概率转移矩阵用了两种方法来求，所以 viterbi.py 模块也对应两个版本，一个是朴素的 viterbi 算法过程，还有一个融合了动态规划求条件概率的版本，第三个是稀疏矩阵求法（在 viterbi 算法中做平滑操作），如下两张图：

```

12     """
13     input: pinyin_list (list): 拼音列表
14     output: ch_list: 汉字列表
15     """
16     def sort_by_value(mydict):...
23
24     prob_list_1 = list(relat_dict[pinyin_list[0]])
25     prob_value = list(map(lambda x: init_dict_p[x], prob_list_1))
26     V = dict(zip(prob_list_1, prob_value))
27
28     if len(pinyin_list) < 2:...
32
33     for i in range(1, len(pinyin_list)):
34         py_later_str = pinyin_list[i]
35         ch_later_str = relat_dict[py_later_str]
36         prob_map = {}
37         for j in ch_later_str:
38             """viterbi算法实现"""
39             max_list = []
40             max_prob = 0
41             for phrase, first_prob in V.items():
42                 first_ch = phrase[-1]
43                 comb_ch = first_ch + j
44                 new_prob = trans_mat_dict[comb_ch]
45                 new_prob = new_prob * first_prob
46                 if new_prob > max_prob:
47                     max_prob = new_prob
48                     max_list = phrase + j
49             if max_prob == 0:
50                 continue
51             prob_map[max_list] = max_prob
52         V = prob_map
53     sorted_V = sort_by_value(V)
54     output_ch_str = list(sorted_V.keys())[0] + '\n'

```

图十一、朴素的 viterbi 算法

```

33     for i in range(1, len(pinyin_list)):
34         py_later_str = pinyin_list[i]
35         ch_later_str = relat_dict[py_later_str]
36
37         prob_map = {}
38         for j in ch_later_str:
39             """viterbi算法实现"""
40             max_list = []
41             max_prob = 0
42             if not ((relat_dict[pinyin_list[i-1]][0] + j) in trans_mat_dict):
43                 """获取语料库"""
44                 former_pos = [m.start(0-1) for m in re.finditer(j, train_str)]
45                 cut_train_list = [train_str[i] for i in former_pos]
46             else:
47                 cut_train_list = []
48
49             for phrase, first_prob in V.items():
50                 first_ch = phrase[-1]
51                 comb_ch = first_ch + j
52                 # new_prob, trans_mat_dict = first_prob * trans_mat.calcCell(comb_ch, trans_mat_dict)
53                 new_prob, trans_mat_dict = trans_mat.calcCell(comb_ch, cut_train_list, trans_mat_dict)
54                 new_prob = new_prob * first_prob
55                 if new_prob > max_prob:
56                     max_prob = new_prob
57                     max_list = phrase + j
58
59             if max_prob == 0:
60                 continue
61             prob_map[max_list] = max_prob
62         V = prob_map
63     sorted_V = sort_by_value(V)
64     output_ch_str = list(sorted_V.keys())[0] + '\n'
65     return output_ch_str, trans_mat_dict

```

图十二、动态规划求条件概率的 viterbi 算法

```

hmm_mat.py x viterbi.py x
11 def viterbi(pinyin_list, relat_dict, init_dict_p, trans_mat_dict):
12     """
13     """
14     def sort_by_value(mydict):
15         min_ch = min(list(init_dict_p.values()))
16         prob_list_1 = list(relat_dict[pinyin_list[0]])
17         prob_value = list(map(lambda x: init_dict_p.get(x, min_ch), prob_list_1)) # 可自次字概率平滑操作
18         V = dict(zip(prob_list_1, prob_value))
19
20     if len(pinyin_list) < 2:
21
22     for i in range(1, len(pinyin_list)):
23         py_later_str = pinyin_list[i]
24         ch_later_str = relat_dict[py_later_str]
25         prob_map = {}
26         for j in ch_later_str:
27             """viterbi算法实现"""
28             max_list = []
29             max_prob = 0
30             for phrase, first_prob in V.items():
31                 first_ch = phrase[-1]
32                 comb_ch = first_ch + j
33                 new_prob = trans_mat_dict.get(comb_ch, init_dict_p.get(first_ch, min_ch)/100) # 条件概率平滑操作
34                 new_prob = new_prob * first_prob
35                 if new_prob > max_prob:
36                     max_prob = new_prob
37                     max_list = phrase + j
38                 if max_prob == 0:
39                     continue
40             prob_map[max_list] = max_prob
41             V = prob_map
42         sorted_V = sort_by_value(V)
43         output_ch_str = list(sorted_V.keys())[0] + '\n'
44         return output_ch_str
45
46 viterbi() > for i in range(1, len(pinyin_li... > for j in ch_later_str > for phrase, first_prob in V.item... > if new_prob > max_prob

```

图十三、稀疏矩阵求解中 viterbi 算法对概率平滑操作

5、模型测试和评估

通过上述模型构建，构建出三个版本的工程。一个是利用语料库裁剪求得完全的概率转移矩阵，然后用朴素 viterbi 算法实现拼音转汉字，命名为 pinyin2chinese_v1；第二个就是利用动态规划方法，在求解过程中计算条件概率，并不断补充到概率转移矩阵文件里，供后面计算使用，命名为 pinyin2chinese_v2；第三个利用初始概率矩阵和概率转移矩阵的稀疏性，求解两个概率矩阵，并在 viterbi 算法的拼音识别过程中进行平滑操作，命名为 pinyin2chinese_v3。

在三个版本工程的文件夹下，调用脚本 pinyin.py 的格式是：”python pinyin.py input.txt output.txt”

```

E:\Tsinghua_course\AI_course_ShaopingMa\homework_1\46540423_1_拼音输入法作业\拼音输入法作业\pinyin2chinese_v1>python pinyin.py input.txt output.txt
总共97个字，正确93个字
正确率有：0.9587628865979382

```

```

E:\Tsinghua_course\AI_course_ShaopingMa\homework_1\46540423_1_拼音输入法作业\拼音输入法作业\pinyin2chinese_v2>python pinyin.py input.txt output.txt
总共97个字，正确93个字
正确率有：0.9587628865979382

```

```

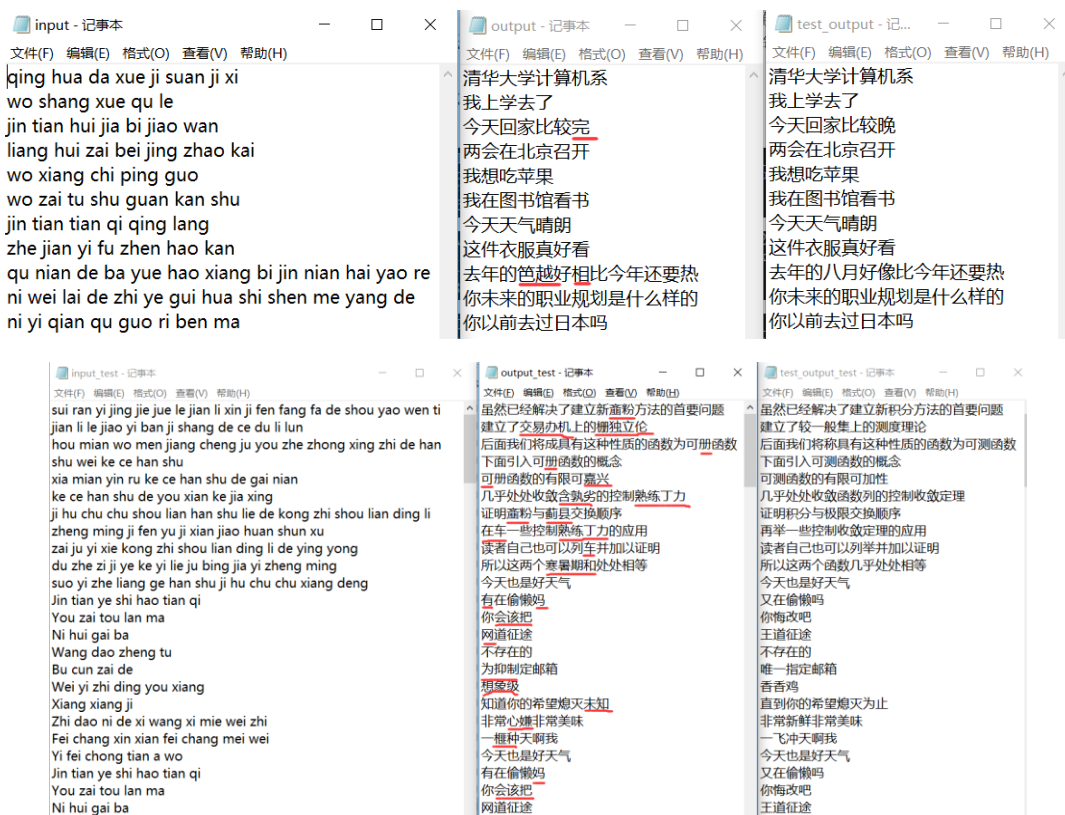
E:\Tsinghua_course\AI_course_ShaopingMa\homework_1\46540423_1_拼音输入法作业\拼音输入法作业\pinyin2chinese_v3>python pinyin.py input.txt output.txt
总共97个字，正确93个字
正确率有：0.9587628865979382

```

```
E:\Tsinghua_course\AI_course_ShaopingMa\homework_1\46540423_1_拼音输入法作业\拼音输入法作业\pinyin2chinese_v1>python pinyin.py input_test.txt output_test.txt
总共1491个字，正确1231个字
正确率有：0.8256203890006707
```

```
E:\Tsinghua_course\AI_course_ShaopingMa\homework_1\46540423_1_拼音输入法作业\拼音输入法作业\pinyin2chinese_v2>python pinyin.py input_test.txt output_test.txt
总共1491个字，正确1231个字
正确率有：0.8256203890006707
```

```
E:\Tsinghua_course\AI_course_ShaopingMa\homework_1\46540423_1_拼音输入法作业\拼音输入法作业\pinyin2chinese_v3>python pinyin.py input_test.txt output_test.txt
总共1491个字，正确1230个字
正确率有：0.8249496981891348
```



图十四、调用格式和正确率

● 三种方法的比较：

1、正确率：

由图十四可以看出，不管是朴素 viterbi 方法，还是融合动态规划求条件概率的方法，计算出来的转移概率矩阵都是完备的，而且概率平滑操作是一样的，所以两种方法的结果是一样的。而第三种求的是稀疏概率矩阵，在平滑操作和前两种尽量保持一致，可以从识别结构看出有微小差异。（平滑操作 $(1 - \lambda)P(Y|X) + \lambda P(X)$ ， λ 取 0.01。）

在给定的 800MB 语料库下，对于一些日常用语识别还是挺好的，比如上图中 input.txt 的识别，正确率可以达到 95.9%；而 input_test.txt 中一些专业术语识别就没那么好了，正确率只有 82.5%。

2、运算速度分析：

第一种方法朴素 viterbi 算法在进行拼音识别之前必须算好，这样才能进行 viterbi 方法求最优路径，开始算概率转移矩阵的耗时非常长(尽管已经用了裁剪语料库的办法)。但是 pinyin.py 脚本执行的时候，概率矩阵加载内存比较慢(尤其是 trans_mat 矩阵，尽管用了 json 格式加快读取)，在两个概率矩阵加载到内存中后，执行速度会特别快！

融合动态规划求条件概率的第二种方法，是在 viterbi 算法需要用条件概率的时候再去算，然后存起来，供以后用，这个方法可以不用耗费非常多时间去计算概率转移矩阵，但是执行 pinyin.py 脚本的时候会比较慢。建议助教师兄测试的时候，输入文件大概 10 行 100 个字，耗时在 20 分钟左右。当然，脚本执行越多次，速度会越快。但是还是会比第一种方法脚本执行的慢，因为这种动态规划求条件概率，每次执行脚本都要读取 800MB 的语料文件，而且逻辑结构也比朴素 viterbi 方法要复杂。最大的好处就是可以不用花那么长时间计算概率转移矩阵，该方法即拿即用，还会自我学习，自我提高。

第三种办法速度介于第一和第二之间，已知部分概率矩阵，在进行拼音识别的时候，会随时计算自己需要的初始概率和条件概率，而这个方法的稀疏概率矩阵还是比第二种动态规划方法一开始存储的概率值多，所以第三种方法快于第二种，慢于第一种方法。

综上所述，前两种方法本质是把求条件概率在不同的阶段实施，但是本质是一样的，第三种方式是利用稀疏矩阵的方式求解概率矩阵。第一种方法文件大，加载到内存比较耗时，但是拼音识别特别快(适合处理特别大的 input.txt 文件，效率高)；第二种方法，概率矩阵文件最小，改用动态规划计算概率矩阵，不需要前期训练，但是拼音识别比较慢(适合处理小的 input.txt 文件，优点是不需要前期训练)；第三种，计算存储了稀疏的概率矩阵，概率矩阵文件不是很大，加载加载快到内存快，但是执行速度介于第一和第二种方法之间(适合处理不很大的 input.txt 文件，介于第二和第二种方法之间)。

四、总结

1、模型优点分析：用三种计算概率转移矩阵的方式，突破了 800MB 大型预料库的概率统计效率优化问题。充分利用了所有的预料数据训练了一个相对比较好的二元语法模型，对日常拼音的识别可以达到 95% 左右的正确率，对于比较生涩的专业用语的拼音识别正确率也可以达到 80% 以上！

2、二元语法模型缺陷分析：由图十四可以看出，该模型比较强的依赖两个字的词组，对于三个词即以上比较弱一点。模型训练的语料库决定对某一类(日常用语)的识别好，对比较生僻的专业用语识别得不是很好。

3、语料库数据预处理方法：根据二元语法模型，很好的预处理成用“S”字符连接相邻的短句，把相邻两句的句尾和句首的异常词频排除。

4、模型提高方案：前面提到的该模型因为没有多音字的语料库，所以发射概率矩阵没有考虑，这也是该模型的不足，有待补充多音字语料库，计算发射概率矩阵，使 viterbi 算法更加完善。

5、过程坑：

a). 第一种方法中，每次统计二元词频的时候原来在全 800MB 语料统计，效率太慢。后来利用二元语法特性裁剪语料库，使得效率有质的提升。But! 即使这样优化，在 8G 内存，4 核 2.4GHz 的服务器上，跑了 12+小时，最后内存爆了……猜测效率慢原因是字典是而非常大，更新非常耗时。所以最后采用分段计算，存成多个 json 文件。

b). 动态规划计算条件概率思想是，某一个需要马上用到的条件概率，可以现场算，然后拿来用，然后把算得的条件概率存起来，下次可以直接用。随时存随时用，随时算随时存。性能有提升很多。

c). 由于语料库巨大，考虑到效率问题，算法尽量优化运用 python 的 map 和生成器等方法应对循环情况，可以提高效率。

d). Viterbi 算法对于拼音长度大于 1 的适用。在单个拼音句子的识别中，直接计算初始概率状态最大的对应汉字，不用 viterbi 方法。

e). 字典为了方便存储和读取，我们存成 json 格式。

f). dict 中键值对是没有顺序的，每次 `list(dict.keys())` 中 list 顺序都不一样……

g). 前两种方法在概率矩阵直接计算时就做了平滑操作，第三种稀疏矩阵的方式在 viterbi 算法识别拼音的阶段才做平滑操作。

源码下载：由于两种方法应用了两种。由于整个工程包含代码和语料库，不方便传上网络学堂，所以这里提供清华云盘的下载链接：

裁剪语料库词频统计方法 - pinyin2chinese_v1:

<https://cloud.tsinghua.edu.cn/d/2bd29e3fa6e946beb23f/>

动态规划求解条件概率方法 - pinyin2chinese_v2:

<https://cloud.tsinghua.edu.cn/d/6fdeaf421dd440a886cb/>

计算稀疏矩阵方法 - pinyin2chinese_v3:

<https://cloud.tsinghua.edu.cn/d/641b4ffa1e83415699d2/>

调用格式：助教老师可以直接下载链接中的工程，文件路径已经设置好，直接在 pinyin2chinese_v1/v2/v3 目录下，运行” `python pinyin.py input.txt output.txt`”，其中 input.txt 是输入拼音文件，output.txt 是脚本生成的识别汉字文件，test_output.txt 是正确的汉字文件，用于正确性检验（脚本已集成计算正确率程序段，把正确的汉字文件命名为” test_[output.txt]” 命名为即可），输出情况下图。可以输出总共有多少字，识别正确多少字。

course_ShaopingMa > homework_1 > 46540423_1_拼音输入法作业 > 拼音输入法作业 > pinyin2chinese_v1 >				
名称	修改日期	类型	大小	
__pycache__	2018/11/19 17:12	文件夹		
my_method	2018/11/20 2:19	文件夹		
my_model	2018/11/21 0:01	文件夹		
init_dict	2018/11/19 2:05	文本文档	84 KB	
init_dict_p	2018/11/19 2:05	文本文档	201 KB	
input	2018/11/20 2:43	文本文档	1 KB	
input_test	2018/11/20 1:30	文本文档	6 KB	
pinyin	2018/11/21 0:39	Python File	3 KB	
readme	2018/11/19 22:48	文本文档	1 KB	
test_output	2018/11/20 2:42	文本文档	1 KB	
test_output_test	2018/11/20 1:29	文本文档	3 KB	
train_trans_mat_dict	2018/11/19 18:11	Python File	2 KB	
trans_mat_dict	2018/11/20 2:47	文本文档	42,943 KB	

```
E:\Tsinghua_course\AI_course_ShaopingMa\homework_1\46540423_1_拼音输入法作业\拼音输入法作业\pinyin2chinese_v1>python pinyin.py input.txt output.txt
总共97个字，正确93个字
正确率有：0.9587628865979382
```

```
E:\Tsinghua_course\AI_course_ShaopingMa\homework_1\46540423_1_拼音输入法作业\拼音输入法作业\pinyin2chinese_v1>python pinyin.py input_test.txt output_test.txt
总共1491个字，正确1231个字
正确率有：0.8256203890006707
```

```

49
50
51 """输出识别汉字文件"""
52 f_output = open(arg_list[2], 'w', encoding='gbk')
53 f_output.write(ch_tot_str)
54 f_output.close()
55
56 """计算准确率"""
57 test_file = 'test_' + arg_list[2]
58 f_otp_test = open(test_file, 'r', encoding='gbk')
59 test_output_str = f_otp_test.read()
60 test_output_str = test_output_str.replace(' ', '')
61 f_otp_test.close()
62
63 test_output_list = list(test_output_str)
64 ch_tot_list = list(ch_tot_str)
65
66 if not (len(test_output_list) == len(ch_tot_list)):
67     print(' the test file is wrong!')
68     print(' generate file length is', len(ch_tot_list))
69     print(' test file length is', len(test_output_list))
70     exit()
71
72 cor_num = 0
73
74 for i in range(len(ch_tot_list)):
75     if ch_tot_list[i] == test_output_list[i]:
76         cor_num = cor_num + 1
77
78 cor_rate = cor_num/len(test_output_list)
79
80 print(' 总共%s个字，正确%s个字' % (str(len(test_output_list)), str(cor_num)))
81 print(' 正确率有： %s' % (str(cor_rate)))

```

图十二、init_dict 和 trans_mat_dict 是模型需要的初始概率矩阵和概率转矩阵，红框是待识别拼音和检测正确性的汉字文件，