人工智能导论

拼音输入法

实验报告

聂鼎宜

计96

2019011346

写在前面

如果您是助教，并且对我这份 7k 字的文档写了什么不感兴趣，只想知道如何运行我的代码、测试我的模型，请直接阅读第 6 节的内容。感谢。

0 关于本项目

0.0 Hello Handkerchief

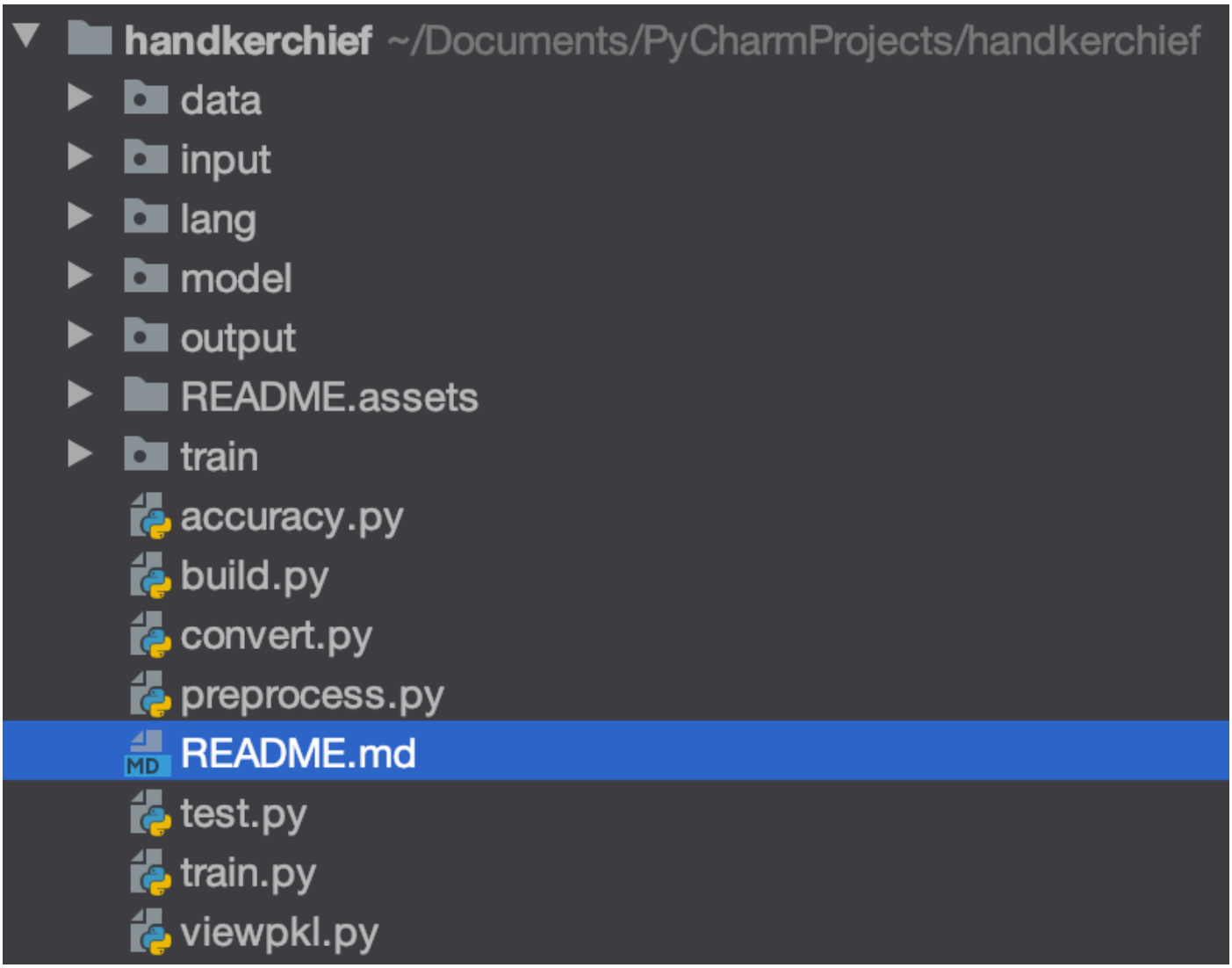
本项目是使用隐式马尔可夫模型实现的汉语拼音输入法 Handkerchief 。

> Handkerchief (手绢)，得名于梗“ 不会打字建议把手绢了”，用来命名这款输入法真是再合适不过，因为经常打不对字。

0.1 语言

- Python：100%；
- 编写、调试环境：macOS Big Sur/Python 3.7/zsh。在其它系统/Python版本/shell 下试图复现将可能带来不同的结果，敬请知悉。

0.2 代码包文件树



- data 目录：输入法读取未经预处理的原始训练数据 (.txt) 的默认目录。
- input 目录：输入法读取测试文件 (.txt) 的默认位置。
- lang 目录：输入法生成的语言包 (.pkl) 的默认存放位置。
- model 目录：输入法生成的模型 (.pkl) 的默认存放位置。
- output 目录：输入法输出运行结果 (.txt) 的默认位置。
- train 目录：输入法读取经过预处理的训练数据 (.pkl) 的默认目录。
- accuracy.py：用于统计输出文件的正确率。
- build.by：构建语言包。
- convert：支持文本文件转码的一个小脚本 (本项目读写文本文件默认编码格式 utf-8，使用其他格式前需要用它转换)。
- preprocess.py：对训练数据进行预处理。
- test.py：测试模型效果，可以接收输入文件将转译结果输出为文本文件，也可以通过命令行直接交互。
- train.py：训练模型并保存。

- viewpkl.py：本项目的储存格式统一使用 pickle，故编写了一个 pickle 实体可视化的小脚本用于测试。
- README.md：本说明文档。
- README.assets 目录：本说明文档所需的图片库。

0.3 如何使用代码包里的代码 (比较重要，请仔细阅读)

0.3.0 前置工作

课程提供的新浪新闻语料集采用 gbk 编码，需先用 ./convert.py 转为 utf-8 再进行后续训练。如果有其他 utf-8 的训练文本，可以跳过这一步。

convert.py usage：

```
handkerchief % python convert.py -h
usage: convert.py [-h] target_path output_path from_which to_which

encoding converter

positional arguments:
  target_path  target file path
  output_path  output file path
  from_which   from which type
  to_which     to which type

optional arguments:
  -h, --help   show this help message and exit

```

代码包里的 data 目录下存放着已经转码好的训练语料集 (2016-*.txt)，直接使用也是可以的。

0.3.1 生成语言包：

本项目用所谓的“语言包”来封装汉字词汇表、拼音-汉字映射表、置换字符等数据，语言包的实质是一个 pickle。

使用 ./build.py 来生成语言包。

build.py usage：

```
handkerchief % python build.py -h
usage: build.py [-h] [--vocab VOCAB] [--dict DICT] [--sub-char SUB_CHAR] [--save
SAVE]

Vocabulary And Dictionary Builder

optional arguments:
  -h, --help           show this help message and exit
  --vocab VOCAB        vocabulary path
  --dict DICT          pinyin-cc index dictionary path
  --sub-char SUB_CHAR  define a substitute char
  --save SAVE          output (.pkl) path

```

- 用 --vocab <path> 指定汇表文件的路径，默认为课程给出的一二级汉字词汇表 ./data/vocab.txt：

> 啊阿埃挨哎唉哀皑癌蔼矮艾碍爱隘鞍氨安俺按暗岸胺案肮昂盎凹敖熬翱袄傲奥懊澳芭捌扒叭吧笆八疤巴拔跋靶把耙坝霸罢爸白柏百摆佰败拜.....

- 用 --dict <path> 指定拼音-汉字映射表的路径，默认为课程给出的字典 ./data/dict.txt:

> a 啊 嗄 腌 吖 阿 锕 ai 镄 暧 爱 呆 嗌 艾 癌 哎 蔼 皑 砹 隘 碍 矮 埃 挨 捱 嫒 唉 哀 霭 嗳 瑷 an 掸 暗 胺 安 广 谙 铵 岸 黯 氨 犴 埯 鞍 鹌 桉 案 按 庵 俺 ang 盎 唔 肮 昂
>
> .....

- 用 --sub_char <char> 指定置换字符。置换字符指的是：在后续训练中，每当输入法从语料库中识别到不在 vocab 中的汉字或其他字符，将自动替换为此字符进行计数、存储。默认为 '0'。

上述三个数据将被输出到指定的 pickle 文件里（默认为 ./lang/lang.pkl），后续代码会经常用到它。

0.3.2 预处理语料集

使用 ./preprocess.py 预处理语料集。usage：

```
handkerchief % python preprocess.py -h
usage: preprocess.py [-h] [--lang LANG] [-t TARGET] [--json] [-k KEYS [KEYS ...]]
[-d OUT_DIR] [--pad] [--clean]

training data preprocessor

optional arguments:
  -h, --help            show this help message and exit
  -l LANG, --lang LANG          load language pack
  -t TARGET, --target TARGET
                        target file path, allow glob
  --json                target is a json (or json list)
  -k KEYS [KEYS ...], --keys KEYS [KEYS ...]
                        if target is json, indicate valid keys here
  -d OUT_DIR, --out-dir OUT_DIR
                        output file directory
  --pad                 pad the sentences with substitute char
  --clean               clean rubbish patterns

```

- 用 -l <path> 来指定语言包。默认为上一步生成的 ./lang/lang.pkl。
- 用 -t <path> 来指定语料集 (.txt) 文件的位置，<path> 中允许 glob 正则。默认为 ./data/2016-*.txt，这些是课程给出的新浪新闻的语料集（在 0.前置工作 中有提到）。
- 用 --json 告诉程序，它将要读取的语料数据文件每一行都是一个格式化的 json 对象。果真如此，需要用 -k <keys> 进一步告诉程序有效的训练数据是 json 对象中哪些键对应的值。由于课程所给的训练数据是这样的：

> {"html": "原标题：快讯：台湾高雄6.7级地震 多幢大楼倒塌 不断传出呼救声中国地震台网测定，今日3时57分在台湾高雄市发生6.7级地震。据台媒报道，地震释放能量相当于两颗原子弹。台南市多处楼房倾斜倒塌。其中，台南市永大路二段一栋住宅大楼倒塌，整栋建筑物倒榻在马路上，建筑物内不断传出呼救声。#高雄6.7级地震#", "time": "2016-02-06 06:45", "title": "快讯：台湾高雄6.7级地震 多幢大楼倒塌 不断传出呼救声", "url": "http://news.sina.com.cn/o/2016-02-06/doc-ifxpfhzk9008548.shtml"}

> {"html": "据台湾中央社报道：高雄美浓今天凌晨发生芮氏规模6.4地震（编者注：即6.7级地震），最大震度6级。台湾“气象局长”辛在勤说，这起地震由复杂破碎的地层错动引起，估计余震会维持1至2个月。辛在勤说，台湾西南部原本就是复杂的破碎地层，有密集的断层带，包括左镇断层、新化断层、甲仙断层和旗山断层，这次美浓地震震央距离2010年3月4日同样规模6.4的甲仙地震位置，才约10多公里，究竟是由那条断层引发，目前不易辨别。气象局估计，余震可维持1到2个月，但规模不太可能超过6。辛在勤请民众不用太惊慌，他说，根据历史资料，台湾西南部的地震，因位在地层破碎带，断层长度不是太长，余震不至于太密集。根据气象局统计，3时57分发生地震后，到清晨6时，余震共40起，其中33起是无感，规模4以上的有5起。", "time": "2016-02-06 07:08", "title": "台气象局:地震因地层错动 余震将持续1至2月", "url": "http://news.sina.com.cn/c/gat/2016-02-06/doc-ifxpfhzq2546760.shtml"}
>
> ......

可以看到有效的条目只有 html 和 title，所以 --json 默认为 True 而<keys> 默认为 html 和 title。

- 用 -d <dir> 指定输出文件目录，默认为 ./train，将在该目录下生成与输入文件 name.txt 一一对应的 name.pkl。
- 用 --pad 来采用 padding（填充）。这意味着，不光不在 vocab 里的字符会在预处理后被替换为语言包所规定的 sub_char（并去重），每个句子的首、尾还会分别加上一个 sub_char。默认为 False。如果你后续训练模型时，希望应用句首、句尾优化（后面会解释），则应该开启它，如无此打算则可以不开启。默认为 True。
- 用 --clean 来命令程序清除无效的字符模式，默认为 True。这些字符模式是编写在 preprocess.py 内部的（全局列表变量 rubbish 里），初始时仅含有“原标题：”一项，因为笔者注意到新浪新闻语料集里以这个开头的句子实在太多了。你可以自己在后面添加不想计入模型的其他字符模式。

如果你**阅读到这里，并且决定使用代码包自带的语料数据和语言包，从而复现笔者的实验**，你现在可以在命令行里不带任何参数地执行：

```
handkerchif % python preprocess.py
```

这样会在 ./train 里生成一系列带有 padding 的预处理好的训练数据 (.pkl)，它们将会在复现实验的阶段被利用。你可以使用如 `python viewpkl.py train/2016-07.pkl 6` 这样的命令来查看生成的训练数据是否符合预期，这行代码一经执行你应该可以看到命令行输出：

```
0火箭军今起换新装0导弹绿0烈焰色0成主打色0新式军服0
0新式军服0
0预计降雨持续至下周一局部有大暴雨0我市今起三天有大到暴雨0
0我市今起三天有大到暴雨0
0日停电检修0早0至下午0景水线0毕阁山0茅屋领0早0至下午0营空线0协和动物中心专用备供0亚洲大酒店0陈四线0长福小区0武汉公安干部学院0麦德龙超市0湖北警官学院及其周边一带0早0至晚0景林线0剑桥春天0栋0供电信息0
0供电信息0
```

0.3.3 训练模型

做好语言包和预处理后，可以进行模型训练了。使用 train.py，usage：

```
handkerchief % python train.py -h
usage: train.py [-h] [--data DATA] [-n N_GRAM] [--save SAVE]

Train Model

optional arguments:
  -h, --help            show this help message and exit
  --data DATA           training data (.pkl) path, allow glob)
  -n N_GRAM, --n-gram N_GRAM
                        train an n-gram model
  --save SAVE           output model pickle path

```

- 使用 --data <path> 指定训练数据文件 (.pkl)，允许 glob 正则匹配多个文件。默认为 .train/2016-*.pkl。
- 使用 -n <num> 指定你要训练一个 <num> 元的模型。模型的维度是向下兼容的，因此训练一个三元模型将能在测试阶段使用二元模型的能力进行测试。由于模型维度增大，存储空间将指数地增大，本项目最高支持四元模型的训练，即 <num> = 2, 3, 4。
- 使用 --save <path> 指定输出的模型 pickle 文件（本质是一个字典）路径，默认为 ./model/model.pkl。

如果你读到这里并试图复现笔者的实验，可以依次执行：

```
handkerchief % python train.py -n 2 --save model/2gram.pkl
handkerchief % python train.py -n 3 --save model/3gram.pkl
handkerchief % python train.py -n 4 --save model/4gram.pkl
```

来分别训练一个 2、3、4 元模型，为后文内容作准备。

0.3.4 测试模型效果

得到一个训练好的模型 pickle，即可开始测试模型的效果。使用 test.py，usage：

```
handkerchief % python test.py -h
usage: test.py [-h] [--model MODEL] [--lang LANG] [-i IN_PATH] [-o OUT_DIR] [-c]
[-n N_GRAM]
               [-b BIAS] [--fp] [--rp] [--smooth SMOOTH [SMOOTH ...] | --dyn-
thresh DYN_THRESH]

Test Model

optional arguments:
  -h, --help            show this help message and exit
  -m MODEL, --model MODEL         model pickle path
  -l LANG, --lang LANG           load language pack (.pkl)
  -c, --thru-cmd        interact through cmd
  -n N_GRAM, --n-gram N_GRAM
                        apply n-gram model, downward compatible
  -b BIAS, --bias BIAS  positive bias for all possibilities
  --fp                  apply front padding
  --rp                  apply rear padding
  --smooth SMOOTH [SMOOTH ...]

```

```
                        give n positive numbers to set smoothing weights for n-dim
and less dim probabilities downwards
```

这一部分仅对各参数的作用进行解释，具体的使用会在后文进行。

- 使用 -m <path> 指定模型 pickle 文件路径，默认为 ./model/model.pkl。
- 使用 -l <path> 指定语言包路径，默认为 ./lang/lang.pkl。由于模型内部储存的数据与语言包密切相关，所以请指定与预处理时所使用的相同的语言包。
- 使用 -n <num> 来测试 <num> 元模型的效果。
- 使用 -b <bias> 来设置 bias（偏置）。默认为 1e-10。偏置是指：在计算 HMM 中的概率项时，对所有概率加以一个正常数 <bias>，从而使所有概率严格大于 0，以规避一些风险，比如实现中出于计算精度考量会对概率取对数而带来的风险。
- 接下来是 io 相关设置：
  - 使用 --c 来通过命令行交互。指：输入一行拼音（空格分隔），回车，输入法将从命令行输出预测结果。默认为 False，代表通过文件 io 交互。
  - 使用 -i 指定输入的测试文件路径。允许 glob 正则。默认为 ./input/test.txt，这是助教在课程群里共享的测试集，主要包含一些日常用语、梗句、黑话等。我把每一句的拼音提取出来放在了这一个文件里。例子：

    > bei jing shi yi ge mei li de cheng shi yi zhi piao liang de xiao hua mao yi zhi ke ai de da huang gou ji qi xue xi ji qi ying yong
    >
    > ……

  - 使用 -o 指定输出预测结果文件的目录。默认为 ./output/。对于每个输入文件 name.txt，会输出一个对应的 name.txt，每一行对应为输入文件中拼音序列的转译结果。
- 接下来是可以 fine-tune 的参数：
  - 使用 --fp 来激活句首优化、--rp 来激活句尾优化，二者的定义后文会解释，默认均为 False。
  - 使用 --smooth 来设置光滑系数。后文会解释。
  - 使用 --dyn-thresh 来设置动态阈值。后文会解释。

0.3.5 统计正确率

最后，如果你不仅有一个拼音输入文件，还有与之对应的 ground truth，可以进一步使用 accuracy.py 统计模型预测结果的正确率，usage：

```
handkerchief % python accuracy.py -h
usage: accuracy.py [-h] [-o OUTPUT] [-gt GROUND_TRUTH] [-s | -c]

Compute Accuracy

optional arguments:
  -h, --help            show this help message and exit
  -o OUTPUT, --output OUTPUT
                        output file path
  -gt GROUND_TRUTH, --ground-truth GROUND_TRUTH
                        output file path
  -s, --by-sentence     compute by sentence
  -c, --by-char         compute by char
```

```
14
```

- 使用 -o 指定模型输出的预测结果的路径，默认为 ./output/test.txt。
- 使用 -gt 指定 ground truth 文件路径，默认为 ./input/ground_truth.txt。这个文件同样来自课程群里助教分享的测试集，我把汉字序列逐句提取出来放在了一起。例子：

  > 北京是一个美丽的城市 一只漂亮的小花猫 一只可爱的大黄狗 机器学习及其应用
  >
  > ……

- 互斥对 -s、-c，前者计算整句正确率（即正确率=完全正确的句子数/句子总数），后者计算逐字正确率（即正确率=正确的字数/总字数）。

代码会把统计结果打印在命令行。

1 算法

下面介绍本项目使用的主要算法：隐马尔可夫模型 (Hidden Markov Model/HMM) 。

HMM，应用于拼音-汉字序列转译，实际上就是两条假设：

- 对于符合拼音序列的一个汉字序列（句子），假设其出现的概率为句中的每个字在前缀确定的条件下出现的概率（即条件概率）的累乘；
- 假设每个条件概率仅与候选字本身及其前 (n - 1) 个字有关，并称满足该假设的一个具体模型为一个 n 元模型。

设 s 为汉字序列，定义 s[i] 代表 s 中的第 i 个字（i 从 0 起计，下同），s[a, b] 代表 s 中第 a 个字到第 b 个字的子串，则 s 出现的概率可以表示为：

$$p_s = \prod_{i=0}^{\text{len(s)-1}} p(s[i]|s[i-n,i])$$

取训练语料集里字或串出现的频率作为概率，对于符合拼音序列的每一个汉字序列，按上式计算概率，输出概率最大的 s，这就是 Handkerchief 的基本算法。

事实上，上式给出的是一个非常本质而脆弱的模型，后续会解释这一点。继而，一系列的优化将被采用，从实验结果中我们将能看到这些优化带来的收益。

2 使用动态规划

上节中指出，我们应该对每一个可能的句子，计算概率、选出最大者，但这么干没人会不嫌慢。

回顾 HMM 的第二条假设：候选字出现的条件概率仅与以它为后缀、长度为 n 的子串有关。这意味着什么？

设使用 n 元模型转译一个长度为 k 的句子 s，那么 s 的前 k - n + 1位一定使得其后接 s 的最后 n - 1位时，整句的概率最大。用一个例子来说明：

> 三元模型转译 qing hua da xue。

> 假设每个拼音有如下候选：
>
> - qing：青、清
> - hua：花、华
> - da：大、打
> - xue：学、雪
>
> 模型现在计算出：
>
> - p(青花大) = 1e-5，p(清花大) = 1e-7
> - p(青华大) = 1e-8，p(清华大) = 1e-2
> - p(青花打) = 1e-6，p(清花打) = 1e-9
> - p(青华打) = 1e-10，p(清华打) = 1e-3
>
> 在计算 xue 的两个候选与目前的 8 个前缀任意组成的串出现的概率前，注意到：
>
> - 在“青华大 (1e-8)”和“清华大 (1e-2)” 中，无论其后接“学”还是“雪”，以“清华大”为前缀都应是更优预测。
>
> 这是因为，三元模型的“视域”只有 3 位：第 1 位的“青”或“清”已经不能影响第 4 位的“学”或“雪”出现的概率。
>
> 因此，在计算 xue 的概率前，可以先进行一次**剪枝**：对于所有以“华大”结尾的串，只保留概率最大者；对”华打”、“花大”、“花打”也是如此。
>
> 这样，候选前缀仅剩下：
>
> - p(青花大) = 1e-5，~~p(清花大) = 1e-7~~
> - ~~p(青华大) = 1e-8~~，p(清华大) = 1e-2
> - p(青花打) = 1e-6，~~p(清花打) = 1e-9~~
> - ~~p(青华打) = 1e-10~~，p(清华打) = 1e-3
>
> 对于更长的句子，此后每一位都作此剪枝，将大大降低算法的复杂度。

n 元模型的计算过程的每一步中，对于每种长为 n-1 的后缀，只保留一个概率最大者，经过这样的剪枝优化可以得到效率更高的动态规划算法，也就是本项目中 test.py 所使用的基本算法。

由于这一优化只提升算法效率，不影响正确率，故不对此做对照实验。

3 迭代：二元模型

3.0 Baseline：裸二元

使用课程给出的测试集。

首先看一下裸二元模型的效果，作为我们的 baseline。做完第 0 节的内容后，命令行执行：

```
handkerchief % python test.py -m model/2gram.pkl -n 2
```

将使用之前训练好的二元模型转译课程助教提供的测试文件 ./input/test.txt，并将结果输出至 ./output/test.txt。

使用 accuracy.py 统计正确率：

```
handkerchief % python accuracy.py -c
Accuracy: 0.8381095273818454
handkerchief % python accuracy.py -s
Accuracy: 0.36288998357963875
```

可以看到逐字正确率已经可以达到 83.8%，可以说是非常不错。不过整句正确率仅 36.3%，还有很大进步空间。

3.1 二元+句首优化

先看看 baseline 的这样一个行为：

```
python test.py -m model/2gram.pkl -n 2 -c
You've chosen to interact with Handkerchief thru cmd.
jiao che
交车
```

原因是："公交车"这个词组出现的频率很高，模型记住了"交车"这个模式，但事实上这并不是一个词语。

如果我们启动句首优化（--fp），模型在计算第一个字出现的概率时，不再计算每个候选字单独出现的频率，而是计算语料库中**这个字在句首出现的概率**（这里的"句首"指的是任何紧跟在 sub_char 后的字符，也就是说标点符号和阿拉伯数字等不属于汉字的字符后紧跟的汉字也算入句首成分）。这样一来：

```
handkerchief % python test.py -m model/2gram.pkl -n 2 -c --fp
You've chosen to interact with Handkerchief thru cmd.
jiao che
轿车
```

模型成功转译出词语"轿车"。

来看看启动句首优化后，模型在测试集上的表现：

```
handkerchief % python accuracy.py -c
Accuracy: 0.8393098274568642
handkerchief % python accuracy.py -s
Accuracy: 0.3694581280788177
```

提升其实很微妙。

3.2 二元+句尾优化

再来看 baseline 的这样一个行为：

```
handkerchief % python test.py -m model/2gram.pkl -n 2 -c
You've chosen to interact with Handkerchief thru cmd.
liao li
了李
```

即时启动句首优化：

```
handkerchief % python test.py -m model/2gram.pkl -n 2 -c --fp
You've chosen to interact with Handkerchief thru cmd.
liao li
了李
```

仍然打出了莫名其妙的“了李”。

句尾优化（--rp），类似于句首优化的概念，它能使转译结果的尾部更加合理。如果开启，模型在计算整句出现的概率时将额外乘上最后一个字在句尾出现的概率。从而：

```
handkerchief % python test.py -m model/2gram.pkl -n 2 -c --rp
You've chosen to interact with Handkerchief thru cmd.
liao li
料理
```

模型打出了更为合理的“料理”。

句尾优化在测试集上的表现：

```
python accuracy.py -c
Accuracy: 0.8447111777944486
python accuracy.py -s
Accuracy: 0.3842364532019704
```

相较句首优化有更好的提升。

3.3 二元+句首句尾优化

来看这样一个例子：

```
handkerchief % python test.py -m model/2gram.pkl -n 2 -c
You've chosen to interact with Handkerchief thru cmd.
li zhi
力支
```

```
handkerchief % python test.py -m model/2gram.pkl -n 2 -c --fp
You've chosen to interact with Handkerchief thru cmd.
li zhi
李志
```

```
handkerchief % python test.py -m model/2gram.pkl -n 2 -c --rp
You've chosen to interact with Handkerchief thru cmd.
li zhi
理制
```

无论 baseline 还是单独启动 fp/rp，输入法在转译“li zhi”时都得到了奇怪的结果。

同时开启两者：

```
handkerchief % python test.py -m model/2gram.pkl -n 2 -c --fp --rp
You've chosen to interact with Handkerchief thru cmd.
li zhi
荔枝
```

得到的结果“荔枝”就要合理许多。

同时启动句首句尾优化在测试集上的表现：

```
handkerchief % python accuracy.py -c
Accuracy: 0.8463615903975994
handkerchief % python accuracy.py -s
Accuracy: 0.39080459770114945
```

是目前为止表现最好的一个迭代。

3.4 二元+光滑

汉语的意群与意群之间并没有过于强的相关性。比如：任何一个形容词+任何一个名词的组合，不大可能有严重的语法错误。而 HMM 假设会导致一个问题，即在上述情况下，**模型会重视形容词的尾部与名词的头部的关联性**，这很可能导致一些不合理的结果。

为了解决这个问题，一个自然的想法是：对于句中的那些候选字，仅考虑他们在前缀出现的条件下出现的条件概率是否不太公平？是否可以将它们**单独出现的概率**也纳入考量？

我们先在测试集上做一次测试。命令行运行：

```
handkerchief % python test.py -m model/2gram.pkl -n 2 --smooth 9 1
```

在二元情形下，--smooth 设为 $[a_2, a_1]$=[9, 1] 意味着每个条件概率项，均有 **0.1 的权重来自于候选字单独出现的概率**。这即是说，以如下方式定义的 P^* 替换原算法中的条件概率：

$$P^*(B|AB) = \frac{w_2}{w_2 + w_1} P(B|AB) + \frac{w_1}{w_2 + w_1} P(B)$$

我们 vimdiff 一下光滑的结果和 baseline 的结果，发现了这样的例子（左光滑，右 baseline）：

可以看到，光滑的模型成功的分开了“不断”、“获得”两个意群，而 baseline 则将“断货”杂糅在了一起。

再比如（左光滑，右 baseline）：

但在一些句子上，光滑的表现变得更差（左光滑，右 baseline）：

所以，想要光滑发挥最大正效益，需要反复调整找到最佳的光滑系数。

这里简单评测一下光滑系数 9:1 的二元模型在测试集上的表现，作为参考：

```
handkerchief % python accuracy.py -c
Accuracy: 0.8393098274568642
handkerchief % python accuracy.py -s
Accuracy: 0.36617405582922824
```

与 baseline 的表现比较，可以看到光滑是可以提供正和的整体收益的。

3.5 二元+句首句尾优化+光滑

作为二元模型的结尾，我们同时应用句首句尾优化和光滑系数 9:1 的光滑优化，给后续实验更高维模型提供一个参考。

```
handkerchief % python test.py -m model/2gram.pkl -n 2 --fp --rp --smooth 9 1
```

```
handkerchief % python accuracy.py -c
Accuracy: 0.844861215303826
handkerchief % python accuracy.py -s
Accuracy: 0.38752052545155996
```

4 进一步迭代：三元模型

4.0 Baseline：裸三元

命令行执行：

```
handkerchief % python test.py -m model/3gram.pkl -n 3
```

来使用不带优化的三元模型测试。在测试集上的表现：

```
handkerchief % python accuracy.py -c
Accuracy: 0.90847711927982
handkerchief % python accuracy.py -s
Accuracy: 0.6075533661740559
```

比起二元是质的飞跃。几个例子：

```
handkerchief % python test.py -m model/2gram.pkl -n 2 -c
You've chosen to interact with Handkerchief thru cmd.
yi da li
一大力
ka bu qi nuo
喀布奇诺
```

```
handkerchief % python test.py -m model/3gram.pkl -n 3 -c
You've chosen to interact with Handkerchief thru cmd.
yi da li
意大利
ka bu qi nuo
卡布奇诺
```

4.1 三元+句首句尾优化

与二元的算法没有本质区别，只是额外考虑前两个字作句首和最后两个字作句尾的概率。

```
handkerchief % python test.py -m model/3gram.pkl -n 3 --fp --rp
```

测试集表现：

```
handkerchief % python accuracy.py -c
Accuracy: 0.9162790697674419
handkerchief % python accuracy.py -s
Accuracy: 0.6354679802955665
```

比起裸三元更进一步。

一些例子（左句首句尾优化，右无优化）：

4.2 三元+句首句尾优化+光滑

三元的光滑系数 $[w_3, w_2, w_1]$ 参与运算的方式就是二元版本的一个推广：

$$P^*(C|ABC) = \frac{w_3}{w_3+w_2+w_1}P(C|ABC) + \frac{w_2}{w_3+w_2+w_1}P(C|BC) + \frac{w_1}{w_3+w_2+w_1}P(C)$$

执行：

```
handkerchief % python test.py -m model/3gram.pkl -n 3 --fp --rp --smooth 90 9 1
```

查看表现：

```
handkerchief % python accuracy.py -c
Accuracy: 0.9294823705926482
handkerchief % python accuracy.py -s
Accuracy: 0.6814449917898193
```

与二元的结果不同，三元下句首句尾和光滑优化一起作用，准确率再次提高了。

5 继续迭代：四元模型

由于四元模型实在过于耗费时间和计算资源，故只进行一次句首句尾优化、光滑优化全上的实验，通过命令行测试能否打出之前的模型打不出来的句子：

```
handkerchief % python test.py -m model/4gram.pkl -n 4 --fp --rp --smooth 900 90 9 1 -c
You've chosen to interact with Handkerchief thru cmd.
shen shuo yao you guang
神说要有光
yu shi you le guang
于是有了光
```

在三元模型的最佳版本里，它的结果是这样的：

```
handkerchief % python test.py -m model/3gram.pkl -n 3 --fp --rp --smooth 90 9 1 -c
You've chosen to interact with Handkerchief thru cmd.
shen shuo yao you guang
参数要有光
yu shi you le guang
于是有了光
```

遂心满意足地退出程序（然后等了足足 1 分钟才完全结束进程）。

6 如何测试我的代码?

第 3、4、5 节的讨论可以明确，综合正确率、效率等因素，最佳的模型应该是三元、带句首句尾优化和光滑的版本。

如果您完整地阅读了之前的所有内容，您应该已经知道如何测试它了。不然，这里有您所需要的所有命令行指令（在 handkerchief 目录下执行，且前提是您没有修改或移除我提交的代码包里的任何一个文件）：

- 执行 `python build.py`，这条指令会利用 ./data 中的一二级汉字表、汉字-拼音映射表生成一个语言包保存在 ./lang 中备用。
- 执行 `python preprocess.py`，这条指令会预处理 ./data 中的训练数据并将结果输出到 ./train 中备用。
- 执行 `python train.py -n 3`，这条指令会基于前面生成的东西训练一个三元模型保存在 ./model 里。

- 执行 `python test.py -n 3 --fp --rp --smooth 90 9 1 -i <your_input_file>`，这条指令会自动加载之前训练好的模型，并设置开启首尾优化和光滑优化，然后从路径 <your_input_file> 读取输入文件并转译，最后将转译结果输出到 ./output/<对应的名字>.txt。
- 执行 `python accuracy.py -o <output_file> -gt <ground_truth_file> -c`，这条指令会从比较 <output_file> 与 <ground_truth_file> 的区别并逐字统计正确率。