# 拼音输入法实验报告

2020211543 杨喜凯 经硕 201

## 目录

## （一）算法原理

### 1.1 HMM 模型

拼音输入法的内在模型实际上为隐马尔科夫模型，我们假设我们要预测的句子为 sentence = $< w_1 w_2 w_3 \dots w_n >$；我们要优化的模型为$P(S) = \prod_{i=1}^{n} P(w_i | w_1 \dots w_n)$，即求解如下问题：

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

为了简化上式，假设该问题满足 k 阶马尔可夫性质，即每个字出现的概率仅与之前 k 个字有关，当 k=1 时，问题便转化为了 2-gram 模型：

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

当 k=2 时，问题转化为 3-gram 模型

$$argmax \ P(w_1, w_2, \dots, w_n) = P(w_1)P(w_2|w_1) \dots P(w_n|w_{n-2}w_{n-1})$$

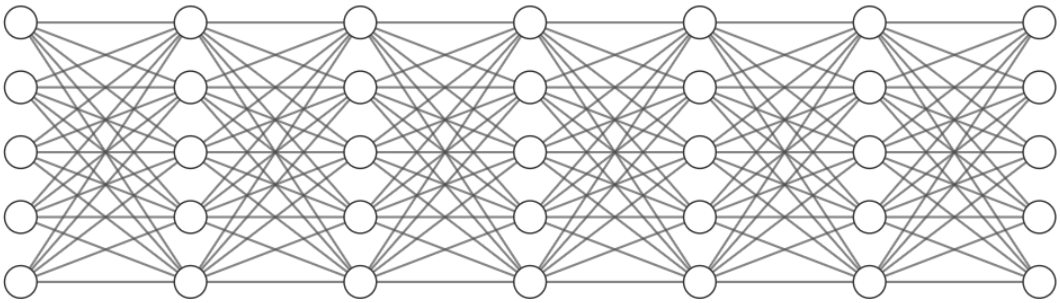
在实际操作中，我们使用每个字或者每个词的频率来帮助我们计算相关的概率和条件概率。

### 1.2 Viterbi 算法

记长度为 i，以 k 为结尾的序列对应的最佳句子为$T_{i,j}$ Viterbi 算法实质上是一种动态规划算法，对于 2-gram Viterbi 算法，

$$P(T_{i,j}) = \begin{cases} max_k \left( P(T_{i-1,k}) * P(W_{i,j} | T_{i-1,k}) \right), & i > 1 \\ P(W_{i,j}), & i = 1 \end{cases}$$

我们首先构建一个以字为节点的网络



之后对于每一层节点，只需在每个节点上保存起点到该节点的最佳路径值以及该最佳路径在上一层节点中所经过的位置；完成所有节点的路径值计算之后，我们只需要做简单的回

溯操作即可确定最终的最优路径。

## （二）优化细节

### 2.1 训练语料处理

我们的原始语料为微博新闻预料，其包含着许多中英文标点符号、数字、字母等等非汉字符号，我们首先利用 zhon.hanzi.stops 中的中文停顿符来进行划分长句，除此之外我们还对常用的逗号分号等符号进行了切割句子，将一个长句切割为若干短句，这样可以避免模型认为一个短句的最后一个字应该和下一个短句的前一个字相连从而发生错误。

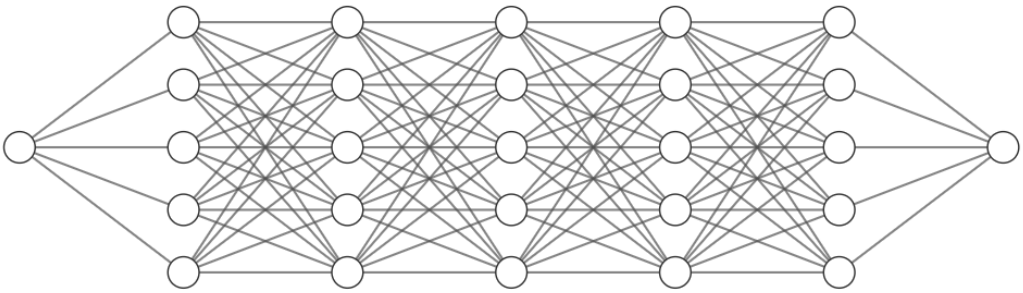
除此之外，我们发现句子中包含很多阿拉伯数字，如果我们单纯地将这部分非汉字的数字从句中删除，我们可能会改变原有句子的结构，例如：发生了 7 级地震；如果我们将句子中的 7 替换为空字符串，那么我们得到的处理后的句子便是：发生了级地震。那么我们的模型便会学习到“了级”这样的词语模式，而这本身是违背我们的初衷的。因此我们利用 cn2an 中的 transform 函数将这部分阿拉伯数字转化为相应的中文数字，从而尽可能保留原有的语句结构。

### 2.2 句首句尾预处理

我们在预料的预处理阶段将文本切割成了许多条短句，那么我们在计算 2-gram 或者 3-gram 的频率时，应该如何对这些短句的句首和句尾词来进行处理呢？最朴素的想法便是“不做额外处理”，计算三元词组的频次时从 index = 2(index = 0,1,2)开始计算，计算二元词组的频次时从 index = 1(index = 0,1)开始计算；但是这样处理存在一定的问题，我在最初没有对句首句尾预处理的时候发现预测出来的错误样例中，很多预测错误都是发生在了句首句尾，例如“今天回家比较顽”以及“激动车驾驶员培训手册”。此时一个很自然的想法便是，不是所有的字出现在句首句尾的概率都相同，某些特定的字更容易出现在句首，某些更容易出现在句尾，同样的，很多字更倾向于出现在句中；那么如果我们能告诉模型某些字更容易出现在句首或者句尾，模型在这方面的表现应该能有所改善。因此我们便在一个句子两端引入了两个哨兵节点(sentinel)，分别标为’B’和‘E’，我们将’B’和’E’也当作正常的汉字来进行计数，例如：“B 发生了七级地震 E”。这样一方面我们修正了上面存在的问题，另一方面因为引入了特殊的哨兵节点，也使我们编程过程中无需考虑句首句尾存在的特殊情况。在引入句首句尾特殊

处理之后，类似于上面的错误例子都得到了修正。

此时我们的 Viterbi 算法网络结构图如下所示，两端的 node 代表句首和句尾的哨兵节点。



2.3 线性平滑算法

我们在计算某个字或者某个词的概率时，如果我们的语料中不包含这个词，那么如果我们将其设置为 0 实际上是不合理的，我们不能因为样本集中不存在该词便认为其概率为 0，这与其真实的分布有所不符。因此对于该类稀疏数据的问题，我们便引入了平滑算法。

通过查阅资料，我发现存在若干种平滑算法，其中包括 Add-one smoothing/Add-k Smoothing/Back-off/Interpolation 等等，在本模型中我们采取了 interpolation 的方法来进行平滑，对于 3-gram，其修正后的概率表达式为：

$$P = \alpha * P(W_i \mid W_{i-1}W_{i-2}) + \beta * P(W_i \mid W_{i-1}) + \gamma * P(W_i)$$

其中α+β+γ=1，通过试验我们最终确定α = 0.88，β = 0.1，γ = 0.02；对于 2-gram 模型，上式将退化为：

$$P = \lambda * P(W_i \mid W_{i-1}) + (1 - \lambda) * P(W_i)$$

通过试验我们最终选取λ = 0.95

2.4 Tri-gram 及相应的高阶 Viterbi 算法

对于 Bi-gram 和 Tri-gram，易见 Tri-gram 考虑了当前 word 及其前面两个 word 的关系，在预测每个字的时候，不仅仅只考虑前面一个字的取值，这样预测出的句子会显得更有连贯性，看起来更为"通顺"。当我们实现 2-gram 之后，一些错误样例的原因便是因为向前考虑的周期太短，例如"一直可爱的大荒沟"，而当我们用 3-gram 来预测时便可以得到准确答案"一只可爱的大黄狗"，我们会发现之所以预测结果是"大荒沟"，我们认为是因为"大荒"这

一词出现的次数较多，可能是常见于"北大荒"之类的词语；"荒沟"这一词也是我们常见的词语；因此在这种情况下便预测成了"大荒沟"。然而在实际生活中，"大荒沟"这一词并不常见，这便是由于我们在预测时没有向前考虑充足的周期；当我们采取 3-gram 之后，"大黄狗"一词频率较高，因此我们最终预测正确；另一个例子是"变成了一个高学理思考能力超群的普通人",我们采取 3-gram 后得到的结果是"变成了一个高学历思考能力超群的普通人"，其中我们在预测"历"这一词时会发现，“高学历"出现的频数较高，但可能"学理"比"学历"出现的频次更高，所以导致了 2-gram 时我们预测出错。因此本质上 2-gram 的缺点在于过于"短视"。

Tri-gram 对应的 Viterbi 算法主要的难度在于此时的每个节点并非仅仅对应着一个最优的上层节点，而是对于每一个上层节点，都存在一个最优值；换句话说，对于每一个节点，此时我们需要搜索的空间从一维空间变成了二维空间。关于高阶 Viterbi 算法的实现，我参考了 Yang he 的一篇文章《Extended Viterbi Algorithm For Second Order Hidden Markov Process》，其中详细地讲解了对于二阶马氏链的 Viterbi 算法伪代码。起初我在写相应代码时思路并非特别清晰，阅读该文章对我完成 3-gram 部分帮助很大。

$$P(T_{i,j}) = \begin{cases} max_{k,h}\left(P(T_{i-2,k}) * P(W_{i-1,h}|T_{i-2,k}) * P(W_{i,j}|T_{i-2,k}W_{i-1,h})\right), & i > 2 \\ max_k(P(W_{i,j}|T_{i-1,k}) * P(T_{i-1,k})), & i = 2 \\ P(W_{i,j}), & i = 1 \end{cases}$$

## （三）功能及代码使用细节

本项目大致可以分为三个部分(各部分代码详情请见 readme)，语料预处理 /训练模型/模型预测。本项目基本使用方式为：python main.py input_path output_path

```
D:\Courses\AI\input_method\src>python main.py ../data/拼音汉字表_12710172/input.txt ../data/拼音汉字表_12710172/output.txt
清华大学计算机系
我上学去了
今天回家比较晚
两会在北京召开
数据挖掘
successfully finished!
```

语料预处理阶段我们主要是将 json 文件中的 html 项提取出来并进行分句，以及处理一些特殊的非汉字符号，除此之外我们还加入了“B”和“E”来代表句子的开始和结束。

训练模型阶段我们主要是将处理好的语料做统计工作，统计单个汉字/两个字/三个字分别出现的频次，分别存在 single_word_db/double_word_db/triple_word_db 中；以 single_word_db 为例，其为一个 dict of dict，其中第一层 key 为拼音，第二层 key 为对应的汉字，value 为该汉字出现的频次，对于 double_word 和 triple_word，只需要将单个字的拼

音改为两个字或者三个字的拼音即可。对应的输出结果放在 result/文件夹下，训练过程可能耗费五个小时；建议您可以从我的网盘链接（https://cloud.tsinghua.edu.cn/d/b16e531813bd4015b918/）中直接下载我对应的结果文件。

以上两部分功能由 corpusParser 类来完成，其中 parse 方法便是训练相应的语料；您也可以直接运行 parseCorpus.py，它会自动初始化 corpusParser 并且进行训练，最终将训练好的内容存到 result/文件夹中。

模型预测部分我们主要实现了 Viterbi 算法；Viterbi 算法可视化之后很像一个神经网络，因此这里我们也实现了三个类，分别是 Node/Layer/Network；其中 Node 代表一个节点，存储相应的拼音/汉字以及一些其他信息；Layer 代表读音相同的若干个 Node，他们是 Viterbi 算法中的“一层节点”，Network 则代表整个网络，network 中实现了 Viterbi 算法，给定拼音序列，可以输出得到的预测句子。

除此之外我们还使用网络学堂提供的测试集进行了测试，并且用该测试集进行了调参，其具体代码可见 test.py。

## （四）实验效果及分析

我们采取网络学堂提供的往年测试集作为测试集来进行分析，我们衡量的指标有两类，第一类是整句的准确率，即只有整个句子完全匹配才算是正确样例；另一类指标是字准确率，即如果一个句子和正确答案有很多重合的正确的字或词，我们会将这部分内容考虑在内，而不是完全忽略掉这个句子。

如下为 2-gram 和 3-gram 分别的效果：

| | 整句准确率 | 字准确率 |
|---|---|---|
| 2-gram | 0.394 | 0.838 |
| 3-gram | 0.631 | 0.907 |

我们对 2-gram 中 interpolation 中的λ进行调参，测试集为上述测试集，得到的结果如下：

| λ | 0.85 | 0.90 | 0.95 | 0.98 | 0.99 |
|---|---|---|---|---|---|
| 整句准确率 | 0.385 | 0.391 | 0.394 | 0.389 | 0.382 |
| 字准确率 | 0.833 | 0.835 | 0.838 | 0.836 | 0.834 |

我们可以看到无论是整句准确率还是字准确率都呈现了一种先增大后减小的趋势，也许我们可以从统计学习的角度理解为当λ接近 1 的时候模型出现了一定程度的过拟合现象，因此最终我们将 lambda 设置为 0.95。

对于 3-gram 的 interpolation，共有三个参数α/β/γ，由于α+β+γ=1，因此我们仅需要调整两个参数。此处我们调整α和β。

| Alpha/beta | 整句正确率 | 字正确率 |
|---|---|---|
| 0.9/0.05 | 0.617 | 0.904 |
| 0.85/0.1 | 0.631 | 0.907 |
| 0.8/0.15 | 0.635 | 0.908 |
| 0.75/0.2 | 0.628 | 0.906 |
| 0.95/0.03 | 0.617 | 0.904 |
| 0.88/0.1 | 0.631 | 0.907 |

从上表中我们可以得到，alpha = 0.88 beta = 0.1 时可以在测试集上取得最好的效果，因此最终我们选择 alpha = 0.88, beta = 0.1。

我们从测试集中可以看出，测试集的语料与训练集有较大的不同；首先时间上差距较大，比如 2016 年的微博语料中"疫情"一词的频次还不是很高，因此模型在测试集中有关疫情的样例上表现的并不是很好；除此之外，训练的语料为新闻样式，因此我们的模型对于一些较为官方的表达表现得更好，但是对于口语化的表达则表现一般；我们发现测试集中还有一些口语化甚至是方言化的样例，我们的模型在这部分样例中表现一般。例如"快处队自对斯塔啊"(快出对子怼死他啊)。

因此我认为上述测试集时间比较新，而且口语化程度比较高，我们的模型在测试集上的表现应该与其真实性能(或者说在其他测试新闻语料下的表现)相比要更差一些；不过我们的测试集也为我们模型的 performance 提供了一个粗略的 lower bound。

2-gram 正确样例

- 特朗普今天面对记者彻底疯了
- 我和我的祖国一刻也不能分割
- 哲学是一切学科的基础

2-gram 错误样例

- 爱吃小熊丙肝
- 机器学习机器应用
- 爱因斯坦提出了广益相对论

3-gram 正确样例

- 马少平老师如是说
- 对亚洲人的种族歧视已经在全球蔓延
- 量子纠缠作为量子通讯的重要资源

3-gram 错误样例

- 给阿姨到一杯卡布奇诺
- 你的理解释对的
- 用女字旁的他来测试一下

错例分析:

对于 2-gram 的错误，我们会发现其都是出自于同样的原因，那便是太过于短视，"广义相对论"中的义字只考虑到了广对它的影响；"爱吃小熊丙肝"中的"丙肝"似乎并没有受到前面"小熊"一词的影响，可能是因为只有"熊"一个字对其有影响，但这不足以使得后面的预测从"丙肝"变为"饼干"。

对于 3-gram 的错误，我们会发现仍存在一些受 2-gram 影响严重的样例，例如"你的理解释对的"，此处即使考虑了"理解"一词，可能由于"解释"的频次太高了，导致最终还是选择了"释"这个字，此处也就涉及到插值平滑算法中三个参数的权衡，如果我们继续提高λ的比重，也许能够改善这个问题。对于“女字旁的他”这一错误样例，易得这一错误是由于“他”的比例高于“她”导致的；而且我们只用了 3-gram，此处如果想让我们的模型智能到能够理解“女字旁”的意思，也许我们需要增加 n-gram 中 n 的数量并且增加训练集数目，甚至需要更换更强大的模型。

除此之外，2-gram 和 3-gram 还存在很多共同的问题，比如其中一个错误样例中包含"你得(dei)"一词，但是我们的模型在将汉字解析成拼音时并没有考虑这部分多音字的问题，而是一律将“得”的拼音解析成了"de"，这也就导致了模型在一些存在多音字的样例中表现不佳。

## （五）反思及改进方案

### 5.1 模型层面的思考

- N-gram 模型只认为一个字与其前面的若干字有关，但是在中文中一个字是与其上下文紧密关联的，也就是说我们不能只向前考虑，我们还应考虑这个字后面的字的情况；因此我认为 Bidirectional-n-gram 可以对原模型有所改善；我的初步想法是除去最初的 n-gram 之外，我们将句子逆序后重新训练一个 n-gram 模型，然后做从左到右以及从右到左的两个 viterbi 算法，最后将其结果结合起来，一个最朴素的想法便是对每个 Viterbi 算法都得到一个概率最大化的句子，然后我们取两者的最大值对应的句子；
- 增加训练语料；从前面的分析中我们也可以看到模型在测试集上不同的样例上的表现很大程度上取决于放入模型的语料特点；为了让模型更具泛化能力，我们可以考虑放入一些非新闻语料(比如口语化的表达)；除此之外，不同时期有不同的语言特色和语言热词(比如今年“疫情”便是一个热词)，让我们的训练语料尽可能多地涵盖多个不同的时期相信也能够提升我们模型的表现。
- 更高阶的 N-gram 模型，我们仅仅采用了 2-gram 和 3-gram，但是很多时候 3-gram 远远不够，我们还可以采用更高阶的 N-gram 模型
- 采用以“词”为基本单位的 N-gram 模型。在我们的模型中我们的 building block 是单个的字，但汉语中我们的常用单位应该是词；因此我们可以考虑将句子进行分词并实现基于词的 N-gram 模型
- 从拼音到汉字是一个 sequence-to-sequence model，我们可以采取深度学习的方法，比如 GRU/LSTM 等等

### 5.2 工程层面的思考

- 在整个代码的实现过程中，存在一个始终没有得到妥善解决的问题，那便是我的语料训练时间过长；我采取的方法是对于 single-word,建立一个 dict of dict，其中 dict 第一层的 key 是拼音，每个拼音对应的字典 keys 是汉字，value 是其出现的频次；对于 double-word 以及 triple-word 也是同样的处理；因此从理论上思考只需要将所有的句子遍历一遍，每次对字典的查找以及更新都是 O(1)复杂度，因此我认为整体来看应该是 O(n)复杂度；但实际上在跑代码的过程中耗时非常长，大约是在我的 PC 上一共跑了五六个小时

(所以助教学长/学姐如果想复现的话可以把代码弄到服务器上跑，或者直接用我提供的训练好的结果也可)。我思考了一下可能是因为建立的 dict 太大了，不能放到我的 L1-cache 甚至是 L3-cache 里面，可能每次查找操作都需要从 main memory 中读取，不能很好地利用 cache locality; 也许更改成矩阵能有所改善，不过限于时间因素这个问题我没有做进一步的优化了。

## （六）总结

之前的我接触人工智能仅限于当一个 python 调包侠，顶多也就是自己实现一些决策树之类的算法；此次是我第一次实现人工智能算法的全套流程，包括处理数据/训练模型/测试/预测等等，而且问题的应用场景非常有新意，当看到自己写的“低能”输入法能够预测正确很多样例的时候(尤其是一些很长很复杂的句子)，内心的自豪感满满。

在整个过程中我收获颇丰，从一开始的只知道 viterbi 算法到后来自己查阅资料知道这是 NLP 中一类经典算法(称为 n-gram)，再后来我自己阅读了很多关于 N-gram 的资料，包括一些网上的博客还有一些国外大学的 slides/readings，知道了 N-gram 实际上并没有一开始想象的那么简单，还需要考虑很多问题(比如说要做线性平滑，否则整个矩阵将非常稀疏)，在这个过程中自己也渐渐学到了一些优化的技巧，比如说要在句首句尾加一些标记符，可以对插值算法的参数进行调参等等；最后在实现二阶 Viterbi 算法的时候因为需要考虑的事情从一维空间变成了二维空间，一开始写代码思路不清晰，后来我又查阅了一些讲 higher order Viterbi 的资料才捋清了自己的思路从而成功写出了 3-gram 模型。在这之后，我对之前一直觉得很神秘的 NLP 有了进一步的了解，明白了它的背后实际上也是一个精妙的统计模型，也进一步体会到了计算机的魅力。

我是一个非 CS 科班的同学，但我从本科就开始就对 CS 产生了兴趣，每次写完代码都能给我以满满的成就感。感谢马老师和助教学长学姐提供的这次大作业机会，让我收获颇丰，感谢悉心指导！