拼音输入法 实验报告

2018011365 张鹤潇 731931282@qq.com

July 1, 2020

目录

1 基本算法

1.1 HMM

拼音输入法的核心在于建立汉语语言模型，即对汉字序列出现的概率进行建模。具体地说，记要预测的汉字序列 $W =< w_1, \cdots, w_n >$，注意，这里的 w 不仅包括汉字，还包括起始符和终止符。我们要估计：

$$\underset{w \in W}{argmax}\, P(w_1, \cdots, w_n) = \underset{w \in W}{argmax}\, P(w_1)P(w_2|w_1)\cdots P(w_n|w_1, \cdots, w_{n-1}) \quad (1)$$

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，则 $k = 1$ 时，问题简化为：

$$\underset{w \in W}{argmax}\, P(w_1, \cdots, w_n) = \underset{w \in W}{argmax}\, P(w_1) \prod_{i=2}^{n} P(w_i|w_{i-1}) \quad (2)$$

用频率估计概率，则：

$$P(w_i|w_{i-1}) = \frac{\#(w_{i-1}w_i)}{\#w_{i-1}}$$

$\#(w_{i-1}w_i)$ 表示 $w_{i-1}w_i$ 在训练语料中出现的次数。

这就是 bigram(二元语法) 模型。$k = 2$ 时，称为 trigram(三元语法) 模型。

1.2 Viterbi 算法

用动态规划的方法最优化2式。记序列 W 中第 i 个字的第 j 个候选为 w_{ij}，长度为 i，结尾为 k 的最佳前缀为 $T_{i,j}$，则：

$$P(T_{i,j}) = \begin{cases} \max\limits_{k}\{P(T_{i-1,k})P(w_{i,j}|T_{i-1,k})\} & i \geq 2 \\ P(w_{i,j}) & i = 1 \end{cases} \quad (3)$$

$$\underset{w \in W}{argmax}\, P(w_1, \cdots, w_n) = \underset{k}{argmax}\, P(T_{n,k})$$

设平均情况下，每个拼音对应 c 个汉字，则上述算法时间复杂度为 $O(cm)$，对于 c 和 m 都是线性的，效率极高。然而，它只能在二元模型下高效地计算出最优解，在三元模型下的优化见下文。

2 成果展示

本代码支持训练任意 n 元字模型。由于开发环境内存所限，在本机上只训练到三元字模型为止。使用和复现方法见readme.

在 2016 年 11 月的新闻语料中随机抽取了 1000 句话，近万字作为验证集。

以全模型在同学们贡献的测试集 (500 多句, 逾 5000 字) 上测试, 结果与验证集相差无几, 可见模型泛化性能的优秀。

表 1: 新闻语料测试结果

模型	γ	band width	字准确率	句准确率	运行时间 (s)
3-gram[1]	200	6	95.22 %	73.20 %	185.9
3-gram with low bias[2]	1	6	97.76 %	85.70 %	195.0
3-gram with low bias	200	6	97.24 %	83.80 %	195.5

[1] 用于调参的模型，训练语料为 11 月前的新闻。

[2] 全模型，训练集除新闻外，包括了来自微信公众号的训练语料。

表 2: 多样化语料测试结果

模型	γ	band width	字准确率	句准确率	运行时间 (s)
3-gram	200	6	94.21 %	74.07 %	92.5

3 优化细节

3.1 训练语料预处理

原始语料中含有大量字母、数字、标点等非汉字字符，先对它们进行预处理。

- 根据断句标点（如逗号，句号，冒号等）将文章分成多句。
- 将每句话中的阿拉伯数字转换成中文数字，去掉剩下的所有非汉字字符。
- 在句首句尾添加 ## 标记。

3.2 句首尾概率的修正

用”##” 对句首句尾作标记，对句首句尾字出现的概率做修正。

3.3 多音字的区分

简单测试之后，我发现许多错误是由字音误判导致的。比如，算法会将”yin xing” 转换成银行。为了解决这个问题，我在预处理中用 pypinyin 对原始语料进行了注音。将一个字的不同读音用数字区分开，如行 (xing) 标记为” 行 0”，行 (hang) 标记为” 行 1”。当然训练和预测的算法也要在细节上做相应的修改。

至此，预处理前后语料文本的变化举例如下：

```
// 预处理前
4月26日第二代乐视超级手机京东首售
// 预处理后
##四0月0二0十0六0日0第0二0代0乐0视0超0级0手0机0京0东0首0售0##
```

当然，pypinyin 注音并不完美，比如它会将“添砖加瓦”标注为“tian zhuan jie wa”. 我觉得手工修复这些错误太繁琐了，并不值得。很遗憾，对于这个问题，我还没有找到解决办法。

3.4 对线性平滑算法的改进

在三元模型中，由于语料的规模限制，有一些三元组在语料中可能没有出现过，或者出现的次数很少，如果直接取 $P(w_i|w_{i-2}w_{i-1})$ 为概率估计值，会导致整句概率过小甚至为 0。一个简单的解决的办法是线性地组合模型：

$$P(w_i|w_{i-2}w_{i-1})_{smooth} = \lambda_1 P(w_i|w_{i-2}w_{i-1}) + \lambda_2 P(w_i|w_{i-1}) + (1-\lambda_1-\lambda_2)P(w_i)$$

但是这就引入了对两个超参数的调节，对其简单改进如下：

$$\lambda_1 = \frac{\#(w_{i-2}w_{i-1})}{\#(w_{i-2}w_{i-1})+\gamma}$$
$$\lambda_2 = (1-\lambda_1)\frac{\#w_{i-1}}{\#w_{i-1}+\gamma}$$

这样就将需要调节的超参数减少到了一个。对该方法的直观解释是，$w_{i-1}w_i$ 出现的次数越多，λ_1 就会越趋近于 1，三元模型就会占主导地位，而二元、一元模型的权重就会减少。

3.5 对 Viterbi 算法的改进

上文提到，3式仅在二元模型下能找到最优解，在三元模型中，Viterbi 算法应该变为：

$$P(T_{i,j}) = \begin{cases} \max\limits_{k,h}\{P(T_{i-2,k})P(w_{i-1,h}|T_{i-2,k})P(w_{i,j}|T_{i-1,h})\} & i \geq 3 \\ \max\limits_{k} P(w_{i,j}|T_{i-1,k}) & i = 2 \\ P(w_{i,j}) & i = 1 \end{cases} \tag{4}$$

现在算法的时间复杂度数量级变为 $O(c^2m)$，实测表明程序的运行效率降低了十几倍，这几乎是不可接受的。

一个更好的选择是选用近似算法。对3式略作改动，对于第 i 个候选字，维护前 B(band width) 个概率最大的序列组合，而不只是最大者。算法时间复杂度 $O(Bcm)$.

$$P(T_{i,j}) = \begin{cases} \underset{k}{topB}\{P(T_{i-1,k})P(w_{i,j}|T_{i-1,k})\} & i \geq 2 \\ P(w_{i,j}) & i = 1 \end{cases} \tag{5}$$

现在 $P(T_{i,j})$ 表示由 B 个候选序列组成的集合，这就大大减少了因为3的贪心剪枝而得不到最优解的情况。

注意到，当 $B=1$ 时，5式退化为3式；$B=c$ 时，5式近似为4式。该近似算法在性能和效率上做了平衡，是非常重要的优化。

最后还有一点：取对数加速概率计算，防止数值下溢。

$$\begin{aligned}\underset{w\in W}{argmax}\, P(w_1,\cdots,w_n) &= \underset{w\in W}{argmax}\, P(w_1)\prod_{i=2}^{n} P(w_i|w_{i-1}) \\ &= \underset{w\in W}{argmax}\, \log P(w_1) + \sum_{i=2}^{n} \log P(w_i|w_{i-1})\end{aligned}$$

3.6 测试效果

综上所述，在交叉验证集上测试，各种优化的效果对比如下：

表 3: 模型优化测试结果 ($\gamma=10$)

ID	模型	首尾修正	多音字	band width	字准确率	句准确率	运行时间 (s)
1	2-gram	有	有	1	90.21%	52.10%	25.5
2	3-gram	无	无	1	92.02%	62.20%	32.5
3	3-gram	有	无	1	92.93%	66.20%	32.1
4	3-gram	有	有	1	93.57%	67.90%	34.2
5	3-gram	有	有	5	94.79%	71.50%	158.6

表 4: 模型预测效果对比

Answer	<1>	<2>	<3>	<4>	<5>
一旦复发常造成终身残疾乃至死亡	一旦复发**场**造成**中省**残疾乃至死亡	一旦复发**肠**造成终生残疾乃至死亡	一旦复发**肠**造成终生残疾乃至死亡	一旦复发**肠**造成终生残疾乃至死亡	一旦复发**畅**造成终生残疾乃至死亡
这大概是我听过的最舒心的音乐会	**浙**大概是我听过的最舒心的音乐会	**着**大概是我听过的最舒心的音乐会	**浙**大概是我听过的最舒心的音乐会	**浙**大概是我听过的最舒心的音乐会	这大概是我听过的最舒心的音乐会
夏山苍翠而如滴	**下山**苍翠而入低	**下山**苍翠而如滴	**下**山苍翠而**入的**	**下山**苍翠而如滴	夏山苍翠而如滴
到后来祭祀人格化的神灵	到后来祭祀人格化的**申领**	到后来**其四**人格化的神灵	到后来**其四**人格化的神灵	到后来**急死**人格化的神灵	到后来**寄私**人格化的神灵

二元模型 <1> 只能考虑相邻一个词的关系，而汉语中频繁出现的二字词又太多，这就容易出现奇怪的结果；不考虑多音字的模型 <2, 3> 容易混淆常见字的读音；剪枝过早的模型 <4> 在长句中的表现受限。但即便是经过了大量优化

的 <5>，在相邻字之间没有明显承接关系时也表现不佳，这是 n-gram 模型本身的问题。

4 参数调节

4.1 gamma 的选取

在区分多音字的三元模型 (band width = 1) 下测试。

表 5: gamma 的选取

gamma	0.1	1	10	100	150	200	250	300	500
字准确率	93.15%	93.34%	93.57%	93.99%	94.12%	94.19%	94.11%	94.15%	94.04%
句准确率	66.50%	67.00%	67.90%	68.50%	68.20 %	68.20%	68.20%	68.40%	67.80%

测试中发现，在一定范围内，增大 γ 能提高模型的泛化能力，即提高模型在训练集中没有的测试语料上的预测性能。见1。

这个现象很容易理解：γ 越大，模型对三元字的依赖就越小，而汉语中的三元组实在太多了，其中大部分在预测语料中没有出现过。可见，在这个场合下，平滑也起到了防止过拟合的作用。

4.2 band width 的选取

在区分多音字的三元模型 ($\gamma = 200$) 下测试。

表 6: band width 的选取

band width	1	3	5	6	7	10
字准确率	94.19 %	95.03 %	95.19 %	95.22 %	95.22 %	95.23 %
句准确率	68.20 %	72.20 %	73.00 %	73.20 %	73.30 %	73.30 %
运行时间 (s)	33.5	92.5	151.1	185.9	216.9	301.4

$5 \sim 7$ 是 band width 的合理选择。

5 总结反思

n-gram 的原理很简单，但是真正实现起来要考虑的细节实在是太多了，这让我深刻体会到了 NLP 的高度复杂。关于如何进一步改进程序，我的想法如下：

- 降低 n 元字模型的内存消耗。在本项目中，我选用 Python 内置的字典 (dict, 基于 hash) 存储状态转移矩阵，这实际上是一种以空间换时间的策略。
- 将 n 元字模型推广到 n 元词模型。词模型应比相应的字模型有更好的预测性能，但消耗的计算资源也更大。
- 设计 GUI。
- 改用深度学习。拼音转汉字有几乎无限的训练语料，使用端到端的深度学习模型（如 BERT）应能取得很好的效果，且不需要太多的特征工程。