拼音输入法 实验报告

致理书院 郭士尧 2022012406

目录

1 摘要

本实验实现了基于统计的中文拼音输入法，提供了基于字的二元模型、基于词的二元模型和基于词的三元模型。通过对平滑方法、语料库、超参数等的调节，最终在测试数据上达到了

80.04% 的行准确率和 96.51% 的字准确率。更完整的测例运行方法和准确率、运行时间等报告参见附录 小节 1。

2 实验环境

- 系统版本：Arch Linux, 6.13.7-arch1-1
- CPU：CPU 20 × 12th Gen Intel® Core™ i7-12700H
- 内存：32GB RAM
- 编译器：g++ 14.2.1 20250207 (GCC)

3 代码框架

代码使用 C++ 编写，提供三种模型，运行方式详见代码 README。

```
├── corpus
│   └── ...
├── data
│   └── ...
├── extra  // 额外的数据文件与词典
├── include
│   ├── cppjieba  // 分词库
│   │   └── ...
│   ├── limonp  // 分词库依赖
│   │   └── ...
│   ├── ime  // 输入法模型
│   │   ├── bigram.hpp
│   │   ├── ime.hpp
│   │   ├── word.hpp
│   │   └── word_tri.hpp
│   ├── aho_corasick.hpp
│   ├── common.hpp
│   ├── corpus.hpp
│   ├── encoding.hpp
│   ├── tables.hpp
│   └── utils.hpp
├── Makefile
├── README.pdf
└── src
    ├── bigram_ime.cpp
    ├── corpus.cpp
    ├── encoding.cpp
    ├── main.cpp
    ├── main_word.cpp
    ├── main_word_tri.cpp
    ├── tables.cpp
    ├── test_aho_corasick.cpp
    ├── utils.cpp
    ├── word_ime.cpp
    └── word_tri_ime.cpp
```

4 语料库与预处理

使用到了如下两种语料库：

- `sina_news_gbk`（`sina`）：随作业下发的新浪新闻数据，GBK 编码
  - 数据量：831MB
  - 预处理：取 `title`、`html` 两列
- `webtext2019zh`（`web`）：来自 https://github.com/brightmart/nlp_chinese_corpus) 的社区问答数据集，UTF-8 编码
  - 数据量：3.7GB
  - 预处理：取 `title`、`content` 两列

预处理的实现上，为了效率考虑没有使用 JSON 解析库而是手动按子串范围匹配。使用 `iconv` 做编码转换，并根据标点符号（`extra/punctuations.txt`）进行分句。详见 `include/corpus.hpp`、`src/corpus.cpp`。

5 基本思路

在本实验中我们需要实现中文拼音输入法。输入是空格分隔的拼音序列，要求模型基于给定的中文语料，推理出拼音对应的中文句子。

5.1 基于字的二元模型

记输入的拼音序列为 $s = s_1 s_2 \cdots s_n$，我们需要找到中文句子 $w = w_1 \cdots w_n$ 使得条件概率 $P(w|s)$ 最大化。由贝叶斯公式：

$$P(w|s) = \frac{P(w)P(s|w)}{P(s)}$$

其中 $P(s)$ 为常数，而 $P(s|w)$ 可近似为 1（因为由汉字推导拼音的准确率可以达到 99.9% [2]），因此我们的目标转换为最大化 $P(w)$。

由条件概率公式可知：

$$P(w) = P(w_1^n) = P(w_1) \cdot P(w_2|w_1) \cdot P(w_3|w_1^2) \cdot \cdots \cdot P(w_n|w_1^{n-1})$$

在我们基于字的二元模型中，$P\left(w_i|w_1^j\right), j \leq i-2$ 被忽略掉了，因此概率转换为：

$$P(w_1^n) = P(w_n|w_{n-1}) \cdot P(w_1^{n-1}) \tag{1}$$

在这里我们只需要记录 w_{n-1} 与 $P(w_1^{n-1})$ 即可计算 $P(w_1^n)$，因此我们基于这个公式实现动态规划，又名 Viterbi 算法。其时间复杂度为 $O(n|\Omega|^2)$，其中 $|\Omega|$ 为字的个数（若使用词模型则为词的个数）。

基于语料库的统计，我们可以得到 P 的估计：

$$P(w_1) = \frac{c(w_1)}{N}$$

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

其中 $c\left(w_i^j\right)$ 为 w_i^j 在语料库中出现的次数，N 为语料库中所有字的个数。

但实际计算中，有可能会遇到 $c(w_{i-1}w_i)$ 的情况，导致概率归零。为了防止这种情况，我们可以采用平滑技术。插值平滑一种简单的平滑方法，是引入超参数 λ 并令：

$$P'(w_i|w_{i-1}) = \lambda \cdot \frac{c(w_{i-1}^i)}{c(w_{i-1})} + (1-\lambda) \cdot \frac{c(w_i)}{N}$$

另外一种效果较好的平滑方法是 Modified Kneser-Ney Smoothing [1]，具体在 小节 6.2.1 介绍。

5.2 基于词的二元模型

实际上基于字的模型存在相当多的局限性。我们可以使用词作为基本单位，来构建基于词的二元模型。推导与上文类似，但需要预测的序列成为了不定长的词序列 $w = w_1 \cdots w_k$。计算方式也与上文相同。

在基于词的模型中，需要经常计算对指定结束位置，有哪些匹配拼音序列的词语。例如，`wo shi zhong guo ren` 中，后缀可能会匹配 `ren - 人`、`ren - 任`、`guo ren - 国人`、`zhong guo ren - 中国人` 等词语。我们可以使用 Aho-Corasick 自动机 [3] 来实现高效的多模式匹配，其本质是将 KMP 匹配算法迁移到 Trie 树上。由于我们这里字符集过于巨大（中文词语），因此不能像普通的实现一样建立完整的自动机，需要动态计算边转移。具体实现见 `include/aho_corasick.hpp`，测试见 `src/test_aho_corasick.cpp`。

5.3 基于词的三元模型

在基于词的二元模型中，我们只考虑了前一个词对当前词的影响，但实际上当前词也与前两个词有关系。我们可以使用三元模型来考虑这个问题。

在这里，式 1 更新为：

$$P(w_1^n) = P(w_n|w_{n-1}w_{n-2}) \cdot P\left(w_1^{n-1}\right)$$

这也意味这我们的算法复杂度变为 $O\left(n|\Omega|^3\right)$。实际计算中，使用平滑方法，引入超参数 α, β，P 的计算方式为：

$$P(w_1^n) = \beta \cdot \frac{c(w_{i-2}^i)}{c(w_{i-2}^{i-1})} + (1-\beta) \cdot \left(\alpha \cdot \frac{c(w_{i-1}^i)}{c(w_{i-1})} + (1-\alpha) \cdot \frac{c(w_i)}{N} \right)$$

6 实验过程

6.1 基于字的二元模型

按照上文的思路实现了基于字的二元模型。按照题目要求，程序需要自动从新浪新闻数据集中训练并给出预测。在此基础上，可以对超参数 λ 进行搜索，寻找最优的平滑参数。实验结果如下：

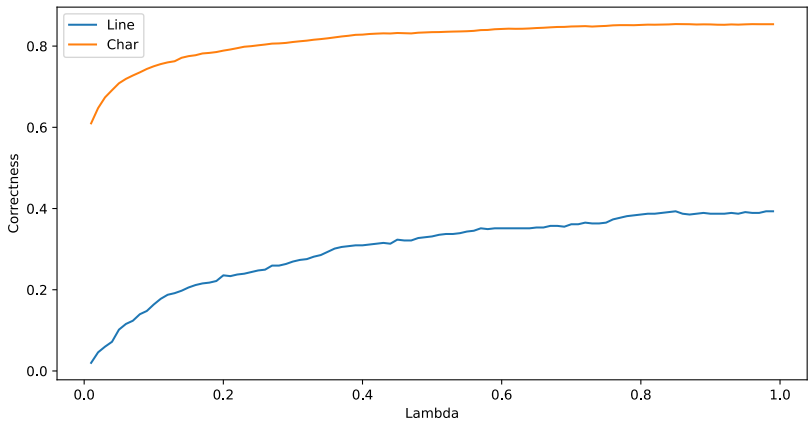

图 1　准确率与平滑参数关系

可以看到在 λ 取到 1 时有巨幅下降，因为此时概率有消失现象。在 $\lambda = 0.85$ 时效果最好，行准确率达到 39.32%，字准确率 85.42%。

6.1.1 首尾 Token 优化

可以使用首尾 Token 来标记句子开头和结尾（`<s>` 和 `</s>`），从而保留更多的语料信息。使用该方法后得到的结果如下：

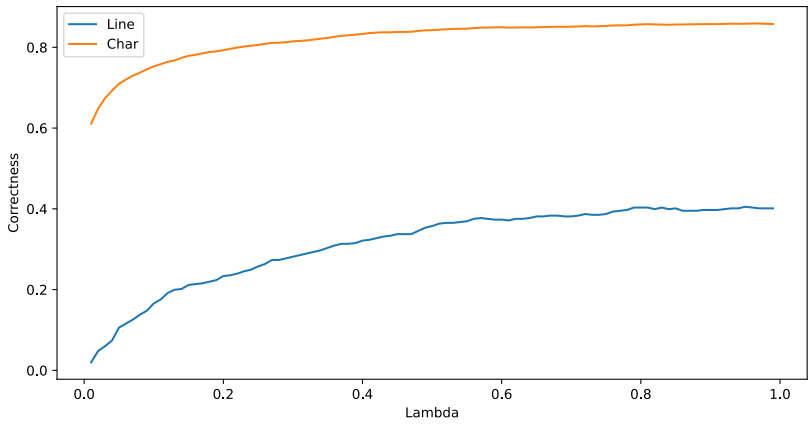

图 2　使用首尾 Token 后准确率与平滑参数关系

在 $\lambda = 0.95$ 时效果最好，行准确率达到 40.52%，字准确率 85.84%。可以看到在使用首尾 Token 后，准确率有了小幅提升。**后文实验中均默认使用该方法**。

6.2 基于词的二元模型

我使用了 cppjieba 库对语料进行中文分词，并使用来自 pypinyin 的拼音数据得到语料的拼音。为了避免重复计算，我们使用 `make-dict` 命令预处理语料，生成词典文件；然后使用 `run` 命令直接基于词典文件进行预测。

`make-dict [dataset]` 会生成 `extra/dict_[dataset].bin` 和 `extra/dict_words_[dataset].txt` 两个文件，前者为二进制编码的词典文件，后者为词表。

该方法下，注意到在 λ 十分靠近 1 时准确率最好，因此对 $\log(1-\lambda)$ 进行搜索，得到的结果如下：

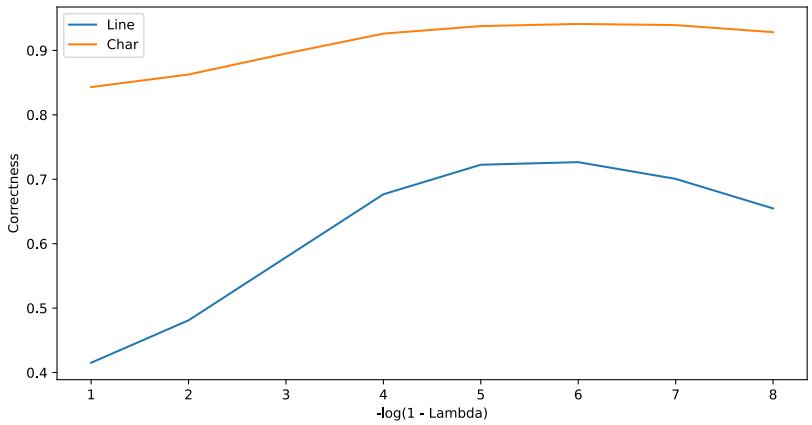

图 3　准确率与平滑参数关系

最后微调后，发现 $\lambda = 0.999998$ 时效果最好，行准确率达到 73.25%，字准确率 94.19%。

6.2.1 Modified Kneser-Ney Smoothing [1]

我们前面使用的插值平滑是相当朴素的。实际上，我在错误例子中注意到两个现象：

- $c(w_{i-1})$ 很小，但 $c(w_i)$ 很大。如 `井冈山 的`，`井冈山` 并不是常用词。此时 $P(w_i|w_{i-1})$ 会相当小，甚至导致任何包含 `井冈山` 的路径被排除。这是因为语料中可能极少出现 `井冈山` 这个词，而正好它后面不跟着 `的`，导致 $\frac{c(w_{i-1}^i)}{c(w_{i-1})}$ 为零。朴素的插值平滑无法对这种情况动态调整。
- $c(w_i)$ 很小，$c(w_{i-1})$ 很大但分布不同。考虑两个词：`的` 和 `清华`。二者词频可能都很大，但 `的` 后面可以跟的词语非常多，`清华` 后面跟的几乎只能是 `大学`。此时如果我们要对于生僻词 `制程` 比较 `的 制程` 和 `清华 制程` 的出现概率，假设二者都没有在语料中出现过，则应当有 `的 制程` 的概率要显著大于 `清华 制程`。然而插值平滑完全无法补偿这种情况。

在查阅相关资料后，我发现了 Modified Kneser-Ney Smoothing 这种方法。其核心思想是将出现频次高的 gram 中分出一部分频次给到低频的 gram；在此基础上，低频的 gram 按照自己前缀的可接续性进行平滑：例如上面 `的 制程` 和 `清华 制程` 的例子，`的` 的可接性远大于 `清华`，因此其会分配到更多的频次。这种方法很好地解决了上述两个问题。

具体而言，二元情况下其概率计算方式为：

$$P_{\text{mkn}}(w_i|w_{i-1}) = u(w_i|w_{i-1}) + b(w_{i-1}) \times P_{\text{mkn}}(w_i)$$

$$P_{\text{mkn}}(w_i) = u(w_n) + \frac{b(\varepsilon)}{|\Omega|}$$

其中 u 为划分后的概率，b 为回退概率，计算方式如下：

$$u(w_i|w_{i-1}) = \frac{c(w_{i-1}^i) - D_n(c(w_{i-1}^i))}{c(w_{i-1})}$$

$$b(w_{i-1}) = \frac{1}{c(w_{i-1})} \times \sum_{j=1}^{3} D_n(j) \cdot |\{x : c(w_{i-1}x) = j\}|$$

使用这种平滑方式后，行准确率达到 72.65%，有所下降；但字准确率达到 94.72%，有小幅提升。

6.2.2 更换语料

新浪新闻语料库的语料量相对较小，且时间较早。若使用 `web` 问答语料库，加上 Modified Kneser-Ney Smoothing，可以达到 79.04% 的行准确率和 96.32% 的字准确率。这也是我在使用二元词模型时达到的最高准确率。

6.3 基于词的三元模型

在基于词的二元模型的基础上，可以通过简单的修改实现基于词的三元模型。但为了避免时间复杂度过高，因此在推理过程中需要适当剪枝。我采用的剪枝策略是，对每一个 n，只保留概率不小于 $\max(P(w_1^n)) \times 10^{-3}$ 的路径，这样可以有效地减少计算量。

此外，如果要将语料中的所有三元组都加入到模型中，可能会造成大量内存开销。为此我只保留了出现频次大于等于 10 的词语，且只保留包含这些词语的二、三元组。最后 `sina` 数据集的词典大小为 181.2MB，`web` 数据集为 308.3MB。

三元模型实现 Modified Kneser-Ney Smoothing 比较复杂，因此我只使用了插值平滑。对插值平滑的超参数进行搜索，得到的结果如下：

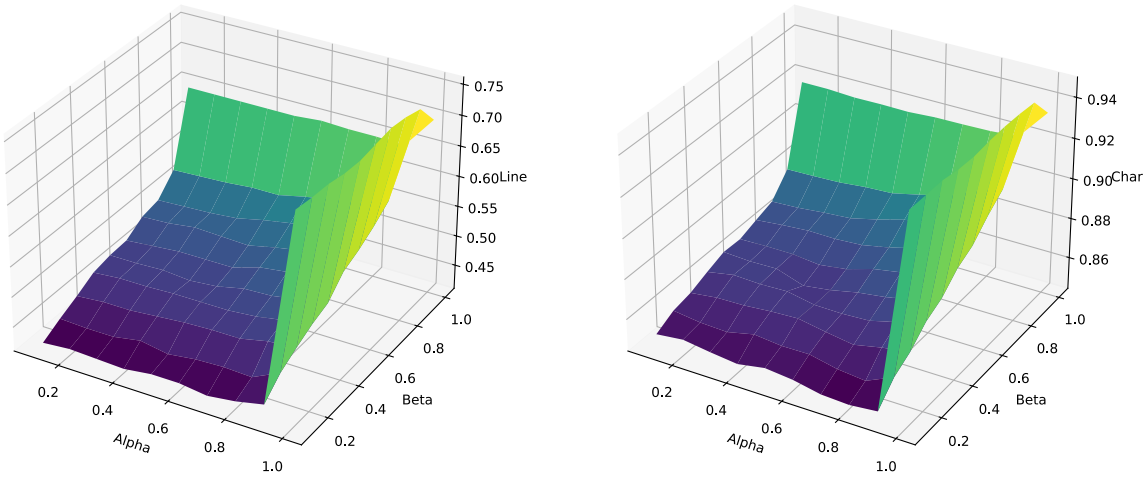

图 4　`sina` 数据集下，准确率与 α、β 的关系

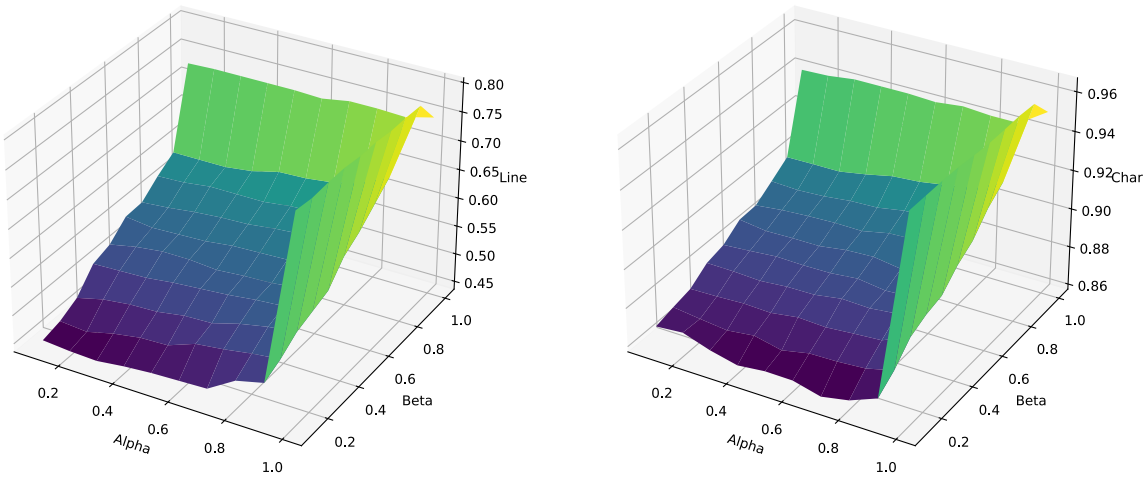

图 5　`web` 数据集下，准确率与 α、β 的关系

可以看到，在 α 接近 1 的情况下，准确率有大幅提升。下面固定 $\alpha = 0.999998$，对 β 进行搜索，得到的最好结果为：

- `sina` 数据集，$\beta = 0.15$，行准确率 75.05%，字准确率 94.75%；
- `web` 数据集，$\beta = 0.15$，行准确率 80.04%，字准确率 96.51%。这也是本次实验中达到的最高准确率。

7 总结

本次实验尝试了许多模型和方法，也让我对机器学习、统计学有了更深的理解。通过不断地修改模型和调整参数提高准确率真的是一件有成就感的事情。虽然最后的准确率并不高，结果中也还有许多离谱的错误，但相比于最初的实现已经有了质的提升。我觉得有几点比较大的收获：

- 语料库的选择真的很重要。基于统计的语言模型，质量好坏与语料质量强相关。质量差的语料，一方面可能导致严重的分布偏差，另一方面可能由于小样本数据过多导致噪声过大；
- 做东西前要进行充分的调研。比如我在研究小样本偏差的问题时，想过置信区间、特殊处理分布等方法，但都只能片面地解决问题，直到最后找到 Modified Kneser-Ney Smoothing 方法才提升了准确率；
- C++ 真的很难写。C++ 真的很难写。C++ 真的很难写。为了性能考虑以及避免每次加载很久导致实验流程过长我在 Python 与 C++ 中选择了后者，但编码转换、split、read_lines 、ULEB 全要手写，map 语法、字符串格式化、构建控制反人类，还有神秘的隐式转换坑了我半小时... 用 Rust 我都不知道我这几天会多轻松。支持 Rust 交作业真的很难吗，求求助教以后给 Rust 人一点基本的人权吧 😭😭😭

但实际上还有很多没有做的可以改进的方向，比如：

- 内存占用过大，实际上完全可以通过优化词典结构来避免推理时要加载整个词典到内存；
- 可以对词语进行词性标注，利用词性信息来协助推理（如特殊处理停词等）；
- 基于 skip-gram 学习词向量，在推理时倾向于选择与当前平均词向量相近的词。

参考文献

[1] K. Heafield, I. Pouzyrevsky, J. H. Clark, 和 P. Koehn, 《Scalable modified Kneser-Ney language model estimation》, 收入 *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, 2013, 页 690–696.

[2] S. Zhang 和 Y. Laprie, 《Text-to-pinyin conversion based on contextual knowledge and D-tree for Mandarin》, 收入 *IEEE International Conference on Natural Language Processing and Knowledge Engineering 2003-NLP-KE'2003.*, 2003, 页 6–p.

[3] A. V. Aho 和 M. J. Corasick, 《Efficient string matching: an aid to bibliographic search》, *Communications of the ACM*, 卷 18, 期 6, 页 333–340, 1975.

附 1 程序参数与性能

该部分包括不同参数配置下，程序的运行时间和内存占用，包括使用的命令。

```
# (1) 基于字的二元模型
make run

# (2.1) 基于词的二元模型，插值平滑，新浪数据集
make main_word
./main_word make-dict sina
./main_word run sina < data/input.txt > data/output.txt

# (2.2) 基于词的二元模型，MKN 平滑，新浪数据集
make main_word KN_SMOOTHING=1
./main_word make-dict sina
./main_word run sina < data/input.txt > data/output.txt

# (2.3) 基于词的二元模型，插值平滑，问答数据集
make main_word
./main_word make-dict web
./main_word run web < data/input.txt > data/output.txt

# (2.4) 基于词的二元模型，MKN 平滑，问答数据集
make main_word KN_SMOOTHING=1
./main_word make-dict web
./main_word run web < data/input.txt > data/output.txt

# (3.1) 基于词的三元模型，新浪数据集
make main_word_tri
./main_word_tri make-dict sina
./main_word_tri run sina < data/input.txt > data/output.txt
```

```
# (3.2) 基于词的三元模型，问答数据集
make main_word_tri
./main_word_tri make-dict web
./main_word_tri run web < data/input.txt > data/output.txt
```

由于推理时间较短，表中的数据都采样了 10 次取平均。

参数组	行准确率	字准确率	词典构建用时	加载用时	推理用时（501 句）	峰值内存
1	40.52%	85.84%	6.35s	-	77.6ms	338.91MB
2.1	73.25%	94.19%	204.71s	2.55s	240.78ms	901.30MB
2.2	72.65%	94.72%		2.90s	247.72ms	905.52MB
2.3	77.25%	95.94%	404.05s	3.24s	288.34ms	1.53GB
2.4	79.04%	96.32%		4.85s	302.82ms	1.54GB
3.1	75.05%	94.75%	289.88s	15.82s	484.42ms	3.33GB
3.2	80.04%	96.51%	694.42s	23.21s	411.12ms	6.37GB

表 1　不同参数组，程序的运行时间和内存占用

附 2 思考题

附 2.1 语料编码

GBK 编码是 GB-2312 字符集的扩展，与 ASCII 编码兼容，并使用 `[0x81, 0xFE]` × `[0x40, 0xA0] \ {0x7F}` 的二字节序列编码中文字符。其字符集只局限在中文字符上，且不支持 Unicode 编码。

UTF-8 与 ASCII 编码兼容，使用变长编码表示 Unicode 字符；其使用第一个字节的高位来表示字符长度。单字符对应字节长度在 1-4 之间。

还有许多其他的编码方式，包括被 Java 默认使用的 UTF-16 编码（现已被 JEP 254 Compact Strings 部分替代），日本有使用的 Shift-JIS 系列，以及为了能在 ASCII 环境中传输二进制数据的 Base64 编码，或者世纪初为了在 ASCII 邮件中传输中文而发明的 hz 编码等。

附 2.2 算法设计

记读音数目为 $|P|$，则期望单个拼音对应字数为 $V/|P|$。Viterbi 算法单次转译所需计算次数为 $(V/|P|)^2$，时间复杂度为 $O(n \cdot V^2)$。

空间复杂度上，对 $k \in [1, n]$ 需要记录前 k 字的方案中，以字符 c 结尾的方案信息。朴素的时间复杂度是 $O(nV)$ 的，如果使用滚动数组优化（只记录前一层状态）则空间复杂度为 $O(n + V)$。

附 2.3 输入输出

两种代码，一种即时将每行输入放到输出，一种将所有输入放到内存中，最后一起输出。二者就实现结果而言都是正确的；区别在于内存占用以及即时性。第二种会需要线性于输入的内存，且无法即时输出。在某些特殊的情况下（OJ 输入会等待程序输出）甚至可能会导致阻塞。

附 3 数据来源与许可协议

所有可下载数据统一放在清华网盘：https://cloud.tsinghua.edu.cn/d/11977280567f4609a3bf/

附 3.1 社区问答语料（`webtext2019zh`）

该语料只有需要生成对应社区语料词典时才需要下载。

- 来源：https://github.com/brightmart/nlp_chinese_corpus
- 清华网盘：`webtext2019zh.zip`
- 文件位置：`corpus/webtext2019zh`
- 许可协议：MIT License

附 3.2 `extra` 部分数据

该部分数据统一打包在清华网盘 `extra.zip`，下载后解压到 `extra` 目录下即可。

附 3.2.1 cppjieba 所需分词数据

- 来源：https://github.com/yanyiwu/cppjieba/tree/b11fd29697c0a6d8e5bef8eab62bae4221e0eda6/dict
- 文件位置：`extra/{jieba.dict.utf8, hmm_model.utf8}`
- 许可协议：MIT License

附 3.2.2 标点符号列表

- 来源：https://github.com/yanyiwu/cppjieba/blob/b11fd29697c0a6d8e5bef8eab62bae4221e0eda6/dict/stop_words.utf8，经过手工过滤
- 文件位置：`extra/punctuations.txt`
- 许可协议：MIT License

附 3.2.3 汉字词语拼音表

- 来源：https://github.com/wolfgitpr/cpp-pinyin/tree/b05278f14ace213d85777ffa55de6967585a6a93/res/dict/mandarin
- 文件位置：`extra/{word.txt, phrases_dict.txt, user_dict.txt, License.txt}`
- 许可协议：CC BY-SA 4.0 (https://creativecommons.org/licenses/by-sa/4.0/)

附 3.2.4 基础词语拼音表

- 来源：基于 汉字词语拼音表 生成，方法为 `python3 src/gen_words.txt`
- 文件位置：`extra/word.txt`

附 3.3 词典数据

该部分可以通过 `make-dict` 命令生成；但生成时间可能较长，可以选择直接下载。

生成方式：

```
# 二元模型词典
make main_word && ./main_word make-dict [dataset]

# 三元模型词典
make main_word_tri && ./main_word_tri make-dict [dataset]
```

当 `[dataset]` 为 `web` 时，需要先下载社区问答语料，见 附录 小节 3.1。

生成的词典文件会放在 `extra` 目录下，命名为 `dict_[dataset].bin` 和 `dict_[dataset]_words.txt` 。

- 二元模型词典（ `main_word` ）
  - `sina` 数据集： `dict_sina.zip`
  - `web` 数据集： `dict_web.zip`
- 三元模型词典（ `main_word_tri` ）
  - `sina` 数据集： `dict_tri_sina.zip`
  - `web` 数据集： `dict_tri_web.zip`