实验报告

杜国豪 2018011346

1. 实验环境

- **操作系统**：Ubuntu 22.04
- **编程语言**：Python 3.8
- **主要库和框架**：无
- **硬件配置**：CPU i7-12700H, 64GB RAM

2. 语料库和数据预处理

- **语料库介绍**：使用的是新浪新闻2016年的新闻语料库，包含多个txt文件，每个文件中存储了多条新闻数据，数据格式为JSON。
- **数据预处理方法**：
  - 将语料库中的sina_news_gbk文件夹放在data文件夹的同级目录
  - 逐行读取，消去所有非汉字字符
  - 统计所有汉字的出现次数，生成1_word.txt
  - 统计所有两字词出现次数，生成2_word.txt
  - 将汉字转换为拼音序列，用于模型训练和测试。

3. 模型设计和实现

a. 基本思路和公式推导

二元模型（Bi-gram Model）是自然语言处理（NLP）中的一种统计语言模型，用于文本中信息的处理和理解。这种模型通过考虑文字或词汇序列中相邻元素之间的关系来预测序列中的下一个元素。在二元模型中，当前词的出现仅依赖于它前面的一个词。这种假设被称为马尔可夫性质。

在二元模型中，给定一个词序列，模型将尝试计算这个序列出现的概率，这通常通过将序列分解为多个二元对（即每个词与它前面的词）并计算这些二元对概率的乘积来实现。对于一个由词$w_1, w_2, ..., w_n$组成的序列，序列出现的概率可以表示为：

$$ P(w_1, w_2, ..., w_n) \approx \prod_{i=2}^{n} P(w_i | w_{i-1}) $$

其中，$P(w_i | w_{i-1})$是在给定前一个词w_{i-1}的条件下，词w_i出现的条件概率。

选择原因：

1. **简单高效**：二元模型在实现上相对简单，计算量适中，对于处理大量文本数据，尤其是实时系统（如拼音输入法）来说，这种高效性非常重要。

2. **上下文相关性**：尽管二元模型只考虑了相邻的一个词的上下文，但这在很多情况下已足够捕捉语言的局部相关性，特别是在汉语中，许多词语的含义和用法紧密依赖于其相邻的词汇，这使得二元模型成为一个实用的选择。

3. **数据需求适中**：相比于更高阶的模型（如三元模型或四元模型），二元模型需要的训练数据较少，这减轻了数据稀疏性的问题。在资源有限的情况下，二元模型能够更好地从有限的数据中学习。

4. **实用性**：在拼音到汉字的转换任务中，二元模型能够有效利用拼音之间的序列信息，预测出最可能的汉字序列。例如，在拼音输入法中，用户输入的拼音序列通常反映了他们想要输入的句子结构，二元模型通过考虑这种结构，能够有效地提高汉字预测的准确性。

公式推导： 如何计算条件概率 $(P(\text{w}*n*|*text{w}*{n-1}))$，并根据它选择最可能的汉字序列。

条件概率 $(P(w_n|w_{n-1}))$ 在二元模型中是核心概念，表示给定前一个字 w_{n-1} 的情况下，下一个字 w_n 出现的概率。这个概率可以根据历史数据中的出现频率来估计。

给定一个字序列，条件概率 $P(w_n|w_{n-1})$可以通过下面的公式计算：

$$ P(w_n|w_{n-1}) = \frac{Count(w_{n-1}, w_n)}{Count(w_{n-1})} $$

其中：

- $Count(w_{n-1}, w_n)$ 表示在训练数据中字对 w_{n-1}, w_n 连续出现的次数。
- $Count(w_{n-1})$ 表示字 w_{n-1} 在训练数据中出现的总次数。

选择最可能的汉字序列

为了根据给定的拼音序列选择最可能的汉字序列，我们需要计算整个序列的概率，并选择具有最高概率的序列作为输出。对于一个由拼音序列对应的多个可能汉字序列，我们计算每个序列的概率，选择概率最高的序列：

$$ P(w_1, w_2, \ldots, w_n) \approx \prod_{i=2}^{n} P(w_i | w_{i-1}) $$

其中，序列的总概率是通过计算序列中每一对连续汉字的条件概率的乘积得到的。

为了实现这一点，通过遍历所有可能的汉字序列组合，计算每个组合的总概率，并保留概率最高的序列。然而，这种方法在实际应用中可能会因为组合数量庞大而变得不切实际。因此，通常会使用如动态规划等更高效的算法来优化这个过程，例如在程序中使用到的Viterbi算法，它能够有效地找到最可能的汉字序列，同时避免了对所有可能序列的显式枚举。

通过这种方法，可以有效地从拼音输入预测出最可能的汉字序列，从而实现拼音到汉字的转换。

4. 实验结果和分析

a. 实验效果

- **准确率**：报告字准确率和句准确率，以及训练时间和测试时间。 字准确率是：0.9488061127029609 句准确率是：0.4151696606786427

训练时间为三小时 程序运行时间: 6.1546783447265625 秒

- **时间分析：** 生成一句话的平均时间为0.01228478711522268秒，就用户体验来说，在可接受范围内

b. 案例分析

- 展示一些预测效果好的例子和效果差的例子，分析可能的原因 效果好的例子： 你一定能够成为你想要去成为的人 我们在世界的远方擦肩而过 希望大家都健康平安

效果不好的例子： 阳光开朗大南海 阳光开朗大男孩

加入在夜间不到你 假如再也见不到你

华化运动员与省戒线 花滑运动员羽生结弦

c. 性能分析

- 对比不同参数设置（如不同的隐藏层维度）对模型性能的影响。 对于不同的k值，搜索宽度，准确率如下

语言	状态	分数	时间
python3	Wrong Answer	34.1 99	2024-04-11 00:37:51
python3	Wrong Answer	34 21	2024-04-11 00:36:43
python3	Wrong Answer	33.6 13	2024-04-11 00:34:51
python3	Wrong Answer	32.6 9	2024-04-11 00:33:31
python3	Wrong Answer	26.1 5	2024-04-11 00:32:47
python3	Wrong Answer	19.4 3	2024-04-11 00:30:17
python3	Runtime Error	0	2024-04-11 00:08:05
python3	Runtime Error	0	2024-04-11 00:04:36

- 时间和空间复杂度分析

时间复杂度

1. **读取数据**：读取和解析 JSON 文件的时间复杂度依赖于文件的大小。假设 `1_word.txt` 和 `2_word.txt` 分别有 (N) 和 (M) 条记录，读取这两个文件的时间复杂度是 (O(N + M))。

2. **计算二元组概率**：`calculate_probability` 函数的时间复杂度是 (O(1))，因为它只执行了几次字典查找和基本的算术操作。

3. **Viterbi 算法**：

    - 对于输入的拼音序列长度为 (L)，每个拼音平均对应 (P) 个汉字。在最坏的情况下，对于每个拼音，算法需要考虑所有 (P) 个汉字对于之前步骤的每个汉字的概率计算。这个过程大致需要 (O(P^2)) 的时间。
    - 在每一步中，算法还需要对概率进行排序并选择前 (k) 个最大概率的节点。在最坏的情况下，这需要 (O(P \log P)) 的时间（如果使用高效的排序算法）。
    - 因此，对于整个拼音序列，Viterbi 算法的时间复杂度大约是 $(O(L \cdot P^2 + L \cdot P \log P))$。

综上所述，整个程序的时间复杂度主要由 Viterbi 算法决定，大致为 $(O(L \cdot P^2 + L \cdot P \log P + N + M))$。

空间复杂度

1. **存储数据**：两个 JSON 文件被加载到内存中，分别占据 (O(N)) 和 (O(M)) 的空间。

2. **Viterbi 算法**：

    - `dp` 和 `path` 字典用于存储每一步的概率和路径。在最坏的情况下，每个拼音对应的每个汉字都需要存储一个概率值和一个路径列表，因此空间复杂度约为 (O(L \cdot P))。这里假设路径列表的空间消耗相对较小，因为它们共享大部分路径。
    - `all_probabilities` 和 `best_probabilities` 列表在每一步中被重用，所以它们的空间消耗可以忽略不计。

因此，整个程序的空间复杂度大致为 $(O(N + M + L \cdot P))$，这里 (N) 和 (M) 分别是两个 JSON 文件中记录的数量，(L) 是拼音序列的长度，(P) 是每个拼音平均对应的汉字数量。

这个分析提供了程序的大致时间和空间复杂度。实际的复杂度可能会根据数据的具体特性和 Python 解释器的内部优化而有所不同。