NLP Lab02: Word Segementation by HMM or BEP

中文分词：HMM

基本原理与具体实现

中文分词问题可以转变为类似词性标注的问题，此时每个字的标签为"B"和"I"，分别代表在该字之后断字、不断字。使用该方法而不是四个标签（词头、词中、词尾、单独成词）是为了避免解码阶段的限制带来的麻烦。

记语料库中某一句子为 $\boldsymbol{v}=(v_1,v_2,\dots v_N)$（ v_i 代表第 i 个字），字 v_i 对应的标签为 h_i ，且

$$h_i=\begin{cases}0, & v_i\text{后不断字，即标签为}B\\ 1, & v_i\text{后断字，即标签为}I\end{cases}$$

则分词问题转化为求解 $\boldsymbol{h}=(h_1,h_2,\dots h_N)$ s.t.

$$\arg\max_{\boldsymbol{h}} P(\boldsymbol{h}|\boldsymbol{v})=\arg\max_{\boldsymbol{h}}\prod_{j=1}^{N}\left[P\left(v_j|h_j\right)\right]P(h_1)\prod_{i=1}^{N-1}P(h_{i+1}|h_i)$$

这一形式中使用了HMM模型，即认为当前状态 h_i 只和前一个状态 h_{i-1} 有关。

求解该问题使用 Viterbi 算法（本质其实是动态规划），具体计算 $P(\boldsymbol{h}|\boldsymbol{v})$ 概率值使用前向算法或后向算法（也使用动态规划进行实现）。

已知参数在程序 `hmm.py` 中对应的变量分别如下：

- $P(h_1)$ 为 start_prob[h_1]（状态 h_1 的起始概率）；
- $P(h_{i+1}|h_i)$ 为 trans_mat[h_i][h_{i+1}]（从状态 h_i 转移到状态 h_{i+1} 的转移概率）；
- $P(v_j|h_j)$ 为 emission_mat[h_j][v_j]（状态 h_j 下出现字 v_j 的发射概率）。

此外，由于本任务仅在短句子上进行效果测试，因此对概率的计算可直接进行连乘。在实践中，常先对概率取对数，将连乘变为加法来计算，以避免出现数值溢出的情况。

Viterbi算法

程序中，我们使用 $dp[state][i]$ 计算第 $(i+1)$ 个字符标注为 $state$ 时对应的最大概率，同时使用 $path[state][i]$ 记录为达到该最大概率值时，第 i 个字符的标注（隐状态）。

则动态规划过程如下：

首先计算前一个状态分别为0和1的情况下，

$$prob[k] = dp[k][i-1] \cdot P^*(v_i|h_i = state) \cdot P(h_i = state|h_{i-1} = k), \quad k \in \{0, 1\}.$$

$$dp[state][i] = \max\{prob[0], prob[1]\}$$

$$path[state][i] = \begin{cases} 0, & \text{if } prob[0] \geq prob[1] \\ 1, & \text{if } prob[0] < prob[1] \end{cases}$$

初始值设置如下：

$$dp[state][0] = P(h_1 = state)P(v_1|h_1 = state), \quad path[state][0] = 0.$$

特别地，为了应对未登录字（不在已有词表 $\mathcal{V}$ 中的字符）的情况，在程序中我们进行如下处理：

$$P^*(v_j|h_j) = \begin{cases} 1, & \text{if } v_j \notin \mathcal{V} \\ P(v_j|h_j), & \text{if } v_j \in \mathcal{V} \end{cases}$$

对本程序中的情况，$\mathcal{V}$ 为 ord<65536 的汉字。因此，我们可以简单地通过 ord<65535 的方式判断当前字是否在词表中。

记最终的结果（能取得最大 $P(\boldsymbol{h}|\boldsymbol{v})$ 概率的 $\boldsymbol{h}$）为 $\boldsymbol{h}^* = (h_1^*, h_2^*, \dots h_N^*)$。$\boldsymbol{h}^*$ 可通过 $path[state][i]$ 得到。具体而言，当 $dp[0][i] \geq dp[1][i]$ 且当前状态为0的情况下记录的上一个状态确实符合目前的最优解（即 $path[i][0] = h_{i-1}^*$）时，h_i^* 标注为0（不断字）；其余情况下标注为1（断字）。

特别地，由于姓名的长度已知（且大概率不在语料库中），我们将姓名部分的字符标注默认为B（不断字）。

综上，记 $\boldsymbol{h}^* = (h_1^*, h_2^*, \dots h_N^*)$，则有

$$h_i^* = \begin{cases} 0, & \text{if } v_i \in \text{name} \\ 0, & \text{if } h_{i-1}^* == path[i][0] \text{ and } dp[0][i] \geq dp[1][i] \\ 1, & \text{otherwise} \end{cases}$$

随后，程序中根据 $\boldsymbol{h}^*$ 来划分原始的语句并返回划分后的句子。

前向算法

我们用前向算法计算 $P(\boldsymbol{v})$ 的概率值。前向算法的整体过程类似于Viterbi算法，唯一的区别是将取最大值的过程变为求和的过程。

动态规划过程如下：（其中的 $dp[state][i]$ 即为HMM前向算法推导中的 $\alpha_{i+1}\left[state\right]$）

$$dp[state][i] = \sum_{k=0}^{1} dp[k][i-1] \cdot P^*(v_i|h_i = state) \cdot P(h_i = state|h_{i-1} = k)$$

其中，为了也能对未登录字进行处理，对发射概率的特殊处理与Viterbi算法中一致。

初始化如下：

$$dp[state][0] = P(h_1 = state)P(v_1|h_1 = state), \quad state \in \{0, 1\}$$

最终的结果为

$$P(\boldsymbol{v}) = \sum_{j=0}^{1} dp[j][N-1].$$

后向算法

后向算法也用于计算 $P(\boldsymbol{v})$ 的概率值，理论上其结果应与前向算法完全一致。其过程类似前向算法，也通过动态规划实现。

动态规划过程如下：（其中的 $dp[state][i]$ 即为HMM后向算法推导中的 $\beta_{i+1}\left[state\right]$）

$$dp[state][i] = \sum_{k=0}^{1} dp[k][i+1] \cdot P^*(v_{i+1}|h_{i+1} = k) \cdot P(h_{i+1} = k|h_i = state)$$

其中，为了也能对未登录字进行处理，对发射概率的特殊处理与Viterbi算法中一致。

初始化如下：

$$dp[state][N-1] = 1$$

最终的结果为

$$P(\boldsymbol{v}) = \sum_{j=0}^{1} P(h_1 = j)P(v_1|h_1 = j) \cdot dp[j][0]$$

代码运行结果

使用命令 `python hmm.py` 即可运行代码，代码实现思路已经在前几小节中叙述，代码中亦有注释。运行结果如下：

```
viterbi算法分词结果： 邱一航/是/一名/优秀/的/学生/
前向算法概率： 5.44758263376502e-32
后向算法概率： 5.447582633765018e-32
```

英文分词：BPE

基本原理与具体实现

简单而言，BPE的过程如下。

首先对语料库进行正则化处理（字母转小写、数字转为N、删除标点符号），根据语料库构建初始状态词典，使用空白字符拆分原始预料得到“词”，并将每个单词拆分为字母，在最后加上“</w>”结束符，统计每个词在语料库中出现的频数。

随后每一轮训练过程中，进行如下处理：

- 对字典中所有的词统计bigram及其出现的频数。注意此处的bigram是指两个“字”（包括合并后成为一个“字的多个字母）的词组，而非单纯指两个字母组成的词组。
- 找到其中频数最高的bigram。将该bigram合并为一个“字”，即将字典中所有包含该bigram的词组中的bigram合并为一个“字”，删除中间用于分隔的空格。

在代码中，`build_bpe_vocab` 函数用于根据语料库直接构建初始状态词典，对每个词进行字母的拆分、“</w>”结束符的插入和空格的插入，同时统计词频。

`get_bigram_fre` 函数生成bigram的字典，同时统计每个bigram的频数。其中bigram的“字”是根据空格拆分得到的。

`refresh_bpe_vocab_by_merging_bigram` 函数将给定的 bigram 合并为一个字，具体而言即将词典中所有出现bigram的地方替换为删去空格的bigram（空格用于分隔两个相邻的字）。

`get_bpe_tokens` 对词典中的字（分词）的频数进行统计，返回统计得到的字典。**注意最后对该词典按照分词的长度进行了降序排序。**

函数 `print_bpe_tokenize` 中的 `bpe_tokenize` 函数通过递归的方式对句子进行分词。其实现如下：

- 若输入的句子长度为0，则返回“”（空）
- 否则，扫描分词字典中的所有分词（由于已经排序，所以是按照长度降序扫描的），若该分词在句子中有出现，则分别对前后的部分进行递归分词，将返回的分词结果拼接后，与当前的分词通过空格分隔并连接。
- 若所有分词都没有在当前（不为空的）句子中出现，则意味着有未登录分词的出现。此时返回"<unknown>"。

代码运行结果

使用命令 `python bpe.py` 即可运行代码，代码实现思路已经在前一小节中叙述，代码中亦有注释。运行结果如下：

Loaded training corpus.
naturallanguageprocessing 的分词结果为：
n atur al lan gu age pro ce s sing</w>