中文分词

班级：2016211303　　姓名：黄若鹏　　学号：2016212901

1.题目要求

- Chinese word segmentation：30 points

-This task used PKU_training set and PKU_testing set as input set.After training,set output.utf8 as a input to score compare with gold_data.

-evaluation metrics:

* precision = (number of words correctly segmented)/(number of words segmented)*100%

*recall = (number of words correctly segmented)/(number of words in the reference)*100%

* F measure=2*P*R/（P+R）

这道题是我本次课程做的第一道题，由于之前分词都是直接用 jieba 库，所以本以为，这道题是比较简单的，可是后来查阅资料发现，中文分词，竟然有这么多算法。因此完成这第一道题，也花了不少功夫，查看了许多开源的博客和资料。

2.概述

中文分词(Chinese Word Segmentation) 指的是将一个汉字序列切分成一个一个单独的词。分词就是将连续的字序列按照一定的规范重新组合成词序列的过程。

中文分词方法

现有的**分词方法**可分为三大类：**基于字符串匹配的分词方法**、**基于理解的分词方法**和**基于统计的分词方法**。

本道题，尝试使用了**基于字符串匹配的分词方法**和**基于统计的分词方法**。

基于字符串匹配的分词方法

（1）**正向最大匹配法**（从左到右的方向）；

（2）**逆向最大匹配法**（从右到左的方向）；

基于统计的分词方法

隐马尔可夫模型（Hidden Markov Model ， HMM）

3.运行环境

系统：windows10

软件工具：VsCode

4.模型建立

4.1 输入输出

输入

为网络上大量使用的中文词汇训练集 icwb2-data 下载地址为 https://github.com/yuikns/icwb2-data

```
"icwb2-data/gold/pku_training_words.txt"
"icwb2-data/testing/pku_test.txt"
```

输出

```
"Outputresult.txt"
```

测试

```
"icwb2-data/gold/pku_test_gold.txt"
```

4.2 模型算法

正向匹配算法

0	1	2	3	4	5	6	7	8	9
我	毕	业	于	北	京	邮	电	大	学

pos	remain characters	start character	max matching
0	我毕业于北京邮电大学	我	我
1	毕业于北京邮电大学	毕	毕业
3	于北京邮电大学	于	于
4	北京邮电大学	北	北京邮电大学

正向最大匹配法，顾名思义，对于输入的一段文本从左至右、以贪心的方式切分出当前位置上长度最大的词。正向最大匹配法是基于词典的分词方法，其分词原理是：单词的颗粒度越大，所能表示的含义越确切。

实现代码

```
#正向匹配
def positive_match(words, word_dict):
    maxLen = len(max(word_dict, key = lambda x: len(x)))
    length = len(words)
    start, end = 0, 0
    newWords = ""
    while start < length:
        end = min(length, start + maxLen)
        while start + 1 < end and (not words[start: end] in word_dict):
            end -= 1
        newWords += " " + words[start: end]
        start = end
    return newWords.strip()
```

逆向最大匹配算法

和正向类似，只不过是从后往前的顺序

实现代码

```
#逆向匹配
def reverse_match(words, word_dict):
    maxLen = len(max(word_dict, key = lambda x: len(x)))
    length = len(words)
    start, end = length, length
```

```
    newWords = ""
    while end > 0:
        start = max(0, end - maxLen)
        while start + 1 < end and (not words[start: end] in word_dict):
            start += 1
        newWords = words[start: end] +  " " + newWords
        end = start
    return newWords.strip()
```

基于 HMM 的分词方法

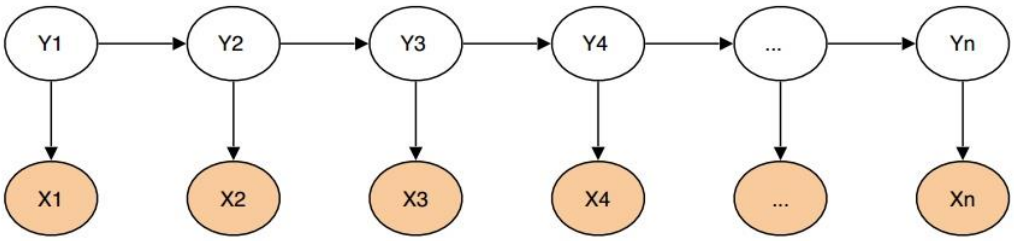
规定每个字有 4 个词位：

- 词首 B
- 词中 M
- 词尾 E
- 单字成词 S

X	我	毕	业	于	北	京	邮	电	大	学
Y	S	B	E	S	B	M	M	M	M	E

由于 HMM 是一个生成式模型，X 为观测序列，Y 为隐序列。

$$P(X,Y)=\prod_{t=1}^{T}P(y_t|y_{t-1})*P(x_t|y_t)$$



HMM 有三类基本问题：

- 预测(filter)：已知模型参数和某一特定输出序列，求最后时刻各个隐含状态的概率分布，即求 $P(x(t) \mid y(1),\cdots,y(t))$。通常使用前向算法解决.
- 平滑(smoothing)：已知模型参数和某一特定输出序列，求中间时刻各个隐含状态的概率分布，即求 $P(x(k) \mid y(1),\cdots,y(t)),k<t$。通常使用 forward-backward 算法解决.

- 解码(most likely explanation): 已知模型参数，寻找最可能的能产生某一特定输出序列的隐含状态的序列. 即求 $P([x(1)\cdots x(t)] \mid [y(1)\cdots,y(t)])$, 通常使用 Viterbi 算法解决.

分词就对应着 HMM 的解码问题，模型参数(转移矩阵，发射矩阵)可以使用统计方法计算得到，原始文本为输出序列，词位是隐状态序列，使用 Viterbi 算法求解即可。

实现代码

```
class HMM(object):
    """
    HMM 模型
    """
    def __init__(self, status, outputs):
        self.status = {s: i for i, s in enumerate(status)}
        self.outputs = {o: i for i, o in enumerate(outputs)}

        self.status_dict = {i: s for i, s in enumerate(status)}

        self.status_num = len(status) #状态个数
        self.outputs_num = len(outputs) #输出个数

        self.init_prob = [1/self.status_num] * self.status_num # 初始状态概率
        self.status_matrix = [[1/self.status_num] * self.status_num for _ in range(self.status_num)] #状态转移矩阵
        self.out_matrix = [[1/self.outputs_num] * self.outputs_num for _ in range(self.status_num)] #输出概率矩阵

    #计算概率，没使用 EM 算法，所以没用到
    def forward(self, outputs):
        """
        前向算法，给定输出 O 与模型，计算输出 O 的概率
        """
        if not outputs: return
        T = len(outputs)
        #将 output 转换为下标
        out = [self.outputs[o] for o in outputs]
        dp = [[0] * self.status_num for _ in range(T)]
        #初始化
        for i in range(self.status_num):
            dp[0][i] = self.init_prob[i] * self.out_matrix[i][out[0]]
        #迭代
        for t in range(1, T):
```

```
            for i in range(self.status_num):
                dp[t][i] = sum((dp[t-1][j] * self.status_matrix[j][i] for
j in range(self.status_num))) * self.out_matrix[i][out[t]]
        return dp

    def backward(self, outputs):
        """
        后向算法，给定输出 O 与模型，计算输出 O 的概率
        """
        if not outputs: return
        T = len(outputs)
        #将 output 转换为下标
        out = [self.outputs[o] for o in outputs]
        dp = [[0] * self.status_num for _ in range(T)]
        #初始化
        for i in range(self.status_num):
            dp[T-1][i] = 1
        #迭代
        for t in range(T-2, -1, -1):
            for i in range(self.status_num):
                dp[t][i] = sum(self.status_matrix[i][j] *
self.out_matrix[j][out[t+1]] * dp[t+1][j] for j in
range(self.status_num))
        return dp

    #学习算法,最大似然算法
    def maximum_likehood(self, words, tags):
        #统计单个句子
        def count(word, tag):
            if not tag: return
            #转为数字下标
            tag = [self.status[t] for t in tag]
            word = [self.outputs[w] for w in word]
            #统计
            init[tag[0]] += 1
            out_tran[tag[0]][word[0]] += 1
            for i in range(1, len(tag)):
                status_tran[tag[i-1]][tag[i]] += 1
                out_tran[tag[i]][word[i]] += 1

        init = [0] * self.status_num # 初始状态概率
        status_tran = [[0] * self.status_num for _ in
range(self.status_num)] #状态转移矩阵
```

```
        out_tran = [[0] * self.outputs_num for _ in
range(self.status_num)] #输出概率矩阵

        list(map(count, words, tags))
        #更新 init_prob
        init_sum = sum(init)
        for i in range(self.status_num):
            self.init_prob[i] = init[i]/init_sum
        #status_matrix
        for i in range(self.status_num):
            status_sum = sum(status_tran[i])
            for j in range(self.status_num):
                self.status_matrix[i][j] = status_tran[i][j]/status_sum
        #out_matrix
        for i in range(self.status_num):
            out_sum = sum(out_tran[i])
            for j in range(self.outputs_num):
                self.out_matrix[i][j] = out_tran[i][j]/out_sum


    #预测算法
    def predict(self, words):
        if not words: return
        T = len(words)
        #将 output 转换为下标
        out = [self.outputs[o] for o in words]
        dp = [[0] * self.status_num for _ in range(T)]
        path = [[0] * self.status_num for _ in range(T)]
        #初始化
        for i in range(self.status_num):
            dp[0][i] = self.init_prob[i] * self.out_matrix[i][out[0]]
            path[0][i] = 0
        #迭代
        for t in range(1, T):
            for i in range(self.status_num):
                tmp = [dp[t-1][j] * self.status_matrix[j][i] for j in
range(self.status_num)]
                dp[t][i] = max(tmp) * self.out_matrix[i][out[t]]
                path[t][i] = tmp.index(max(tmp))
        #得到最优解
        res_path = [0] * T
        P = max(dp[T-1])
        res_path[T-1] = dp[T-1].index(P)
        #路径回溯
        for t in range(T - 2, -1, -1):
```

```
            res_path[t] = path[t+1][res_path[t+1]]
        return "".join([self.status_dict[s] for s in res_path])
```

5．结果分析

HMM

```
PS C:\Users\Rocair> cd 'c:\Users\Rocair\Desktop\Chines
vscode\extensions\ms-python.python-2018.12.1\pythonFil
p\ChineseWordSegment\HMM.py'
P is:  0.7637007269205508
R is:  0.7690686516299717
F1 is:  0.7663752897478147
PS C:\Users\Rocair\Desktop\ChineseWordSegment>
```

最大匹配

```
'python' 'c:\Users\Rocair\.vscode\extensions\ms-python.python-2018.12
30' 'c:\Users\Rocair\Desktop\ChineseWordSegment\Maximum_matching.py'
P is:  0.8712015323290366
R is:  0.846199293534965
F1 is:  0.858518419324189
```

结果一开始有点令我不解，一度怀疑是不是，代码有问题。总觉得方法高级一点的，应该效果会好一点，很明显，HMM 的方法要比最大匹配的方法高端，但是结果上看，反而，最大匹配得出来的结果要更好。

后来也到网上查了相关博客，用不同的语料做训练集，效果会有很大的不同
表是盗过来的

	正向最大匹配	反向最大匹配	隐马尔科夫
as	0.284	0.285	0.064
	0.269	0.270	0.071
	0.276	0.277	0.067
ciytu	0.908	0.910	0.554
	0.838	0.840	0.406
	0.872	0.874	0.467
msr	0.957	0.955	0.775
	0.917	0.915	0.740
	0.937	0.935	0.757
pku	0.907	0.909	0.591
	0.843	0.845	0.731
	0.874	0.876	0.654

从上表中可以看出，

正向最大匹配和反向最大匹配对于词典有很严重的依赖，词典的好坏直接决定了分词的效果；

用隐马尔科夫模型在 as 数据集上测试时，由于汉字字库文件中不包含繁体字，即存在很多的未登录词，所以分词结果效果很差。

由于正向最大匹配和反向最大匹配使用的词典都是各个数据集对应的训练文件，故分词效果比 HMM 高出许多。

另外，在性能上，数据集比较小时正向和反向最大匹配比 HMM 稍快，但数据集大时 HMM 开始超过正向和反向最大匹配，这些测试中最长用时为 3.5 秒。

6.个人总结：

虽然课上时，可能由于没有相关扎实的知识储备，有些东西确实没有听懂，但是做作业，查看许多博客，确实学到了一些知识。