HMM-命名实体识别 算法说明

学号：20373594 姓名：魏少杭

〇、整体思路和步骤

第一，学习过程。

在学习过程中，通过./train.json中的所有text中他们的label种类的统计，训练得到各隐状态之间的概率转移矩阵A、每一个label状态下得到各观测值的概率矩阵B、初始概率矩阵（即第一个字符的label类型的分布）pi。

第二，推理过程。

编写Viterbi算法对输入的dev.json观测序列进行解码，得到预测状态序列。

第三，评估。

在步骤二种，得到的状态序列与dev.json的真实标签进行对比，计算出精确率Precision、召回率Recall和F1值。

一、学习过程

定义隐状态概率转移矩阵A，表示i到j状态转移的概率；定义每一个状态下的观测值矩阵B，表示从某状态观测到某字符的概率；定义pi列表为初始的概率分布。为了统计出以上三者，利用似然估计方法，通过频数的比值代替概率值。

```
A = {}
B = {}
pi = {}
# 为了能够统计出pi，我们定义count_state，分别对每一个状态进行统计出现次数
count_state = {}
# 统计各个状态下，出现各观测值的个数
count_s_o = {}
# 统计从i状态到j状态的个数
count_sij = {}
# 统计所有的状态
count_l_s = []
```

然后对./train.json文件按行解析出text和label两个部分的字典。第一，遍历所有label标签，并用`hidden_states`存储一行text每一个字符的隐状态。

```
        # 下面扫描所有的label
```

```
        # k: e.g. 'address'
        for k in label_now.keys():
            index1 = labelDict[k]   # e.g. address-> 1
            index2 = 2 * index1-1     # e.g. B-ADDRESS 的index就是adress的2倍-1, 2*1-1=1
            # obs:观测字典，如{"汉堡": [[3, 4]], "汉诺威": [[16, 18]]}
            obs = label_now[k]

            #locations 如： [[3,4], [6,7], [10,11]]
            for locations in obs.values():
                list_location_ob_word = locations     # list_location_ob_word是定位的列表，可能有多次出现的位置，如[[3,4],[7,8]]
                for item in list_location_ob_word:
                    hidden_states[item[0]] = index2         # B-xxx

                    for i1 in range(item[0]+1, item[1]+1):
                        hidden_states[i1] = index2 + 1      # I-xxx
```

扫描整个text的 `hidden_states`，获得各个频数的更新：

```
        # 下面进行counter更新
        for obs_index in range(len(text_now)):
            # if hidden_states[obs_index] == 0:
            count_state[hidden_states[obs_index]] += 1      # 各状态出现次数计数
            # 如果之前没有统计过这个 状态-观测值的出现次数，现在就令其为 1
            if text_now[obs_index] not in count_s_o[hidden_states[obs_index]].keys():
                # 状态-观测值概率
                count_s_o[hidden_states[obs_index]][text_now[obs_index]] = 1
            else:   # 否则，出现次数需要加1
                count_s_o[hidden_states[obs_index]][text_now[obs_index]] += 1

        for obs_index in range(1, len(text_now)):
            # 状态i到状态j的转移频数
            count_sij[hidden_states[obs_index-1]][hidden_states[obs_index]] += 1
```

用似然估计方法对A、B、pi进行估计。理论依据是：

$$A[i][j]=\frac{P(S_iS_j)}{P(S_i)}=\frac{N(S_iS_j)}{N(S_i)},\quad B[i][O]=\frac{P(S_iO)}{P(S_i)}=\frac{N(S_iO)}{N(S_i)},\quad \pi_i=P(S_i|t=1)=\frac{N(S_i)}{\sum_j N(S_j)}$$

```
sum_1_s = sum(count_1_s)
for i in range(n):
    for j in range(n):
        # 状态转移概率矩阵估计（最大似然估计）
        A[i][j] = count_sij[i][j] / count_state[i]
    for ob in count_s_o[i].keys():
        # 状态-观测值 估计
        B[i][ob] = count_s_o[i][ob] / count_state[i]
    # 初始状态的概率分布 估计
    pi[i] = count_1_s[i] / sum_1_s
```

二、推理过程

针对./dev.json下的文件中的每一个数据进行处理，通过观测各行字符串，根据之前求取的A，B和pi，推测出各行字符串的各字符的隐状态label。

1.维特比算法说明

维特比算法首先定义了Vertebi变量$\delta_t(i)$

定义：**Viterbi** 变量 $\delta_t(i)$是在时间 t 时，模型沿着某一条路径到达 S_i，输出观察序列 $O=O_1O_2\ldots O_t$ 的最大概率为：

$$\delta_t(i)=\max_{q_1,q_2,\cdots,q_{t-1}} p(q_1,q_2,\cdots,q_t=S_i,O_1O_2\cdots O_t\mid\mu) \quad \ldots (6.22)$$

我们只需要求得每一个时间t下使得变量$\delta_t(i)$最大的那个状态i，就可以得到最可能生成当前text的一组隐藏序列。

递归计算：$\delta_{t+1}(i)=\max_j[\delta_t(j)\cdot a_{ji}]\cdot b_i(O_{t+1}) \qquad \ldots (6.23)$

- **算法6.3**：**Viterbi 算法描述**

(1) 初始化：$\delta_1(i)=\pi_i b_i(O_1)$， $1\le i\le N$

概率最大的路径变量：$\psi_1(i)=0$

(2) 递推计算：

$$\delta_t(j)=\max_{1\le i\le N}[\delta_{t-1}(i)\cdot a_{ij}]\cdot b_j(O_t),\quad 2\le t\le T,\quad 1\le j\le N$$

$$\psi_t(j)=\arg\max_{1\le i\le N}[\delta_{t-1}(i)\cdot a_{ij}]\cdot b_j(O_t),\ 2\le t\le T,\ 1\le i\le N$$

(3) 结束：

$$\hat{Q}_T = \underset{1 \le i \le N}{\operatorname{argmax}}[\delta_T(i)], \qquad \hat{p}(\hat{Q}_T) = \max_{1 \le i \le N} \delta_T(i)$$

(4) 通过回溯得到路径（状态序列）：

$$\hat{q}_t = \psi_{t+1}(\hat{q}_{t+1}), \quad t = T-1,\ T-2,\ \cdots,\ 1$$

算法的时间复杂度： $O(N^2 T)$

而需要主要到，在HMM模型中使用动态规划思想求解该问题的前提必须是两个基本假设：

1.马尔科夫的，则当前状态的转移概率只与前一步的状态有关

2.观测值是独立产生的，也就是只与当前的状态i有关，与其他均无关

以下是初始化、动态规划迭代部分：

```
        # 初始化derta
        derta = {}
        phi = {}
        for t in range(length):
            derta[t] = {}
            phi[t] = {}
            if t == 0:
                for i in range(n):
                    # 对于t时刻下的观测值，其是由i状态产生的概率是这样的
                    if seq[t] in B[i].keys():
                        derta[t][i] = pi[i] * B[i][seq[t]]
                    else:   # 如果之前从没有在i状态下观测到过B
                        # derta[t][i] = 0
                        # if i == 0:
                        derta[t][i] = pi[i] * (1 / count_state[i]) # 尝试用(1 / count_state[j])代替B[j][seq[t]]
                        # else:
                            # derta[t][i] = 0
                    # 这里实际上直接应用到了HMM的两个基本假设
                    # 1.马尔科夫的，则当前状态的转移概率只与前一步的状态有关
                    # 2.观测值是独立产生的，也就是只与当前的状态i有关，与其他均无关
                    phi[t][i] = 0

        # 更新迭代
        for t in range(1, length):
            for j in range(n):
                max_derta_1 = -100000
                max_phi = 0
                for i in range(n):  # 遍历上一个时刻，状态i
```

```
                    # Attention!  temp没有乘以B[j][O_t]
                    temp = derta[t-1][i] * A[i][j]
                    if max_derta_1 < temp:
                        max_derta_1 = temp
                        max_phi = i
                if seq[t] in B[j].keys():
                    derta[t][j] = max_derta_1 * B[j][seq[t]]
                else:    # 如果在j状态下从没观察到过当前的观测值
                    # derta[t][j] = 0
                    # if j == 0:     # TODO:当完全得到了模型评价之后，再来看如果没出现过当前的观测值应该怎么处理
                    derta[t][j] = max_derta_1 * (1 / count_state[j]) # 尝试用(1 / count_state[j])代替B[j][seq[t]]
                    # else:
                        # derta[t][j] = 0
                # 更新当前的phi值
                # derta[t][0] = 1
                phi[t][j] = max_phi
```

最后，遍历最终得到的δ数组中的最大值，以及最大值对应的状态q：

```
        # 结束遍历
        #首先定义最优序列q
        q = np.zeros(length)
        p_t_max = max(derta[length-1].values())
        for i in range(n):
            # 这里需要思考，如果出现了derta相同的情况，选择最大值如何选取，我是直接选取的从前到后出现的第一个最大的
            if derta[length-1][i] == p_t_max:
                q[length-1] = i
                break    # 这里我就直接让从前到后遍历到的第一个最大值为q

        # 回溯以得到路径
        for t in range(length-2 , -1, -1):
            q[t] = phi[t+1][q[t+1]]
```

最终得到了一行字符串中的最有可能的隐藏状态序列（在q列表中）

三、评估

思路：首先，遍历统计所有的预测标签和真实标签。分别用TP FP FN TN存储在每一个标签维度下的预测/真实 | 正确/错误的个数列表,index就是各类标签

对每一个label进行考察，设置Positive表示“当前label”，Negative表示"其他label"，所以可以得到如下的表

<table>
<tr><td colspan="2" rowspan="2">混淆矩阵</td><td colspan="2">真实值</td></tr>
<tr><td>P（是当前label）</td><td>N（不是当前label）</td></tr>
<tr><td rowspan="2">预测值</td><td>P'（是当前label）</td><td>TP</td><td>NP</td></tr>
<tr><td>N'（不是当前label）</td><td>FN</td><td>TN</td></tr>
</table>

```python
# 定义和初始化评估参数
TP = []
FP = []
# TN = []
FN = []

for i in range(21):
    TP.append(0)
    FP.append(0)
    # TN.append(0)
    FN.append(0)
```

针对每一行的数据：

```python
        # 在dev.json每一行根据当前的句子情况，更新TP, FP, FN
        for i in range(length):
            hid_state = hidden_states_dev[i]
            pred_state = pred_states_dev[i]
            if hid_state == pred_state:
                TP[hid_state] += 1
            else:
                FP[int(pred_state)] += 1
                FN[hid_state] += 1
                # 不必计算TN
```

有了TP,FP,FN后计算precision，recall和F1 score

```python
# 下面进行一些precision、recall和f1-score的计算
TP = np.array(TP)
FP = np.array(FP)
FN = np.array(FN)
precision = np.zeros(n)
recall = np.zeros(n)
F1 = np.zeros(n)

precision = TP / (TP + FP)
recall = TP/ (TP + FN)
F1 = 2 * precision * recall / (precision + recall)
```

评价结果：

见`output版本1.txt`，结果如下

```
Label           Precision       Recall          F1-score
O               0.961471        0.870602        0.913783
B-ADDRESS       0.510836        0.442359        0.474138
I-ADDRESS       0.513346        0.622272        0.562585
B-BOOK          0.505952        0.551948        0.527950
I-BOOK          0.468317        0.539339        0.501325
B-COM           0.611702        0.608466        0.610080
I-COM           0.559494        0.672243        0.610708
B-GAME          0.626087        0.732203        0.675000
I-GAME          0.624623        0.759912        0.685658
B-GOV           0.433702        0.635628        0.515599
I-GOV           0.469476        0.813670        0.595409
```

```
B-MOVIE         0.507937        0.635762        0.564706
I-MOVIE         0.471240        0.762332        0.582441
B-NAME          0.685466        0.679570        0.682505
I-NAME          0.526074        0.671890        0.590108
B-ORG           0.561680        0.583106        0.572193
I-ORG           0.500000        0.528979        0.514081
B-POS           0.538899        0.655889        0.591667
I-POS           0.538618        0.690104        0.605023
B-SCENE         0.360153        0.449761        0.400000
I-SCENE         0.432702        0.601108        0.503188

```

四、其他说明

在"三、评估"部分所给出的结果是我认为较为合理的其中一种方案。之所以称之为较为合理，是因为我观察到训练数据的时候，我们的状态-观测值矩阵中，可能没有包含dev.json等除了训练数据之外的可能性。所以，在进行动态规划算法实现的时候需要合理地处理这一部分新的状态-观测值的矩阵。

在以上"二、推理过程"用维特比算法进行初始化和迭代的代码部分

```
        # 初始化derta
        derta = {}
        phi = {}
        for t in range(length):
            derta[t] = {}
            phi[t] = {}
            if t == 0:
                for i in range(n):
                    # 对于t时刻下的观测值，其是由i状态产生的概率是这样的
                    if seq[t] in B[i].keys():
                        derta[t][i] = pi[i] * B[i][seq[t]]
                    else:   # 如果之前从没有在i状态下观测到过B
                        # derta[t][i] = 0
                        # if i == 0:
                        derta[t][i] = pi[i] * (1 / count_state[i]) # 尝试用(1 / count_state[j])代替B[j][seq[t]]
                        # else:
                            # derta[t][i] = 0
                    # 这里实际上直接应用到了HMM的两个基本假设
                    # 1.马尔科夫的，则当前状态的转移概率只与前一步的状态有关
                    # 2.观测值是独立产生的，也就是只与当前的状态i有关，与其他均无关
                    phi[t][i] = 0

        # 更新迭代
        for t in range(1, length):
            for j in range(n):
                max_derta_1 = -100000
                max_phi = 0
                for i in range(n):  # 遍历上一个时刻，状态i
                    # Attention!  temp没有乘以B[j][O_t]
                    temp = derta[t-1][i] * A[i][j]
                    if max_derta_1 < temp:
                        max_derta_1 = temp
                        max_phi = i
                if seq[t] in B[j].keys():
                    derta[t][j] = max_derta_1 * B[j][seq[t]]
                else:   # 如果在j状态下从没观察到过当前的观测值
```

```
37                      # derta[t][j] = 0
38                      # if j == 0:    # TODO:当完全得到了模型评价之后，再来看如果没出现过当前的观测值应该怎么处理
39                      derta[t][j] = max_derta_1 * (1 / count_state[j]) # 尝试用(1 / count_state[j])代替B[j][seq[t]]
40                      # else:
41                          # derta[t][j] = 0
42                  # 更新当前的phi值
43                  # derta[t][0] = 1
44                  phi[t][j] = max_phi
```

这段代码中10-15行、34-39行代码就是在针对从未在某个状态下出现过当前观测值的情况，对数据进行合理处理的部分。尤其是在第15、39行代码，在这种特殊情况下计算$\delta(S_i, O_t)$的时候，由于$B(S_i, O_t) = 0$，正常情况下$\delta(S_i, O_t)$也应当为0，但是这可能会导致由于训练数据的不完备性导致该状态S_i本身合理但被直接忽略掉，进而导致验证/应用效果降低。因此，我用$\frac{1}{\text{隐藏状态}S_i\text{在训练数据中的总数}}$来代替$B(S_i, O_t)$，这个比值能够约等于$\frac{N(S_i\text{导致了当前新出现的训练数据})}{N(\text{隐藏状态}S_i\text{在训练数据中的总数})}$，所以这个处理方法也具有数值上的可行性。

另外一种处理方式，较为简单粗暴，即，如果当前状态S_i下对当前观测值没有发射矩阵的值与之对应，则先判断S_i是否为其他类型O，如果是，则直接$\frac{1}{\text{隐藏状态}S_i\text{在训练数据中的总数}}$来代替$B(S_i, O_t)$；如果不是$O$状态，则令该状态下的$\delta(S_i, O_t)$为0，在遍历过程中，后续的从该时间t以状态$S_i$开始的概率变为0。

具体实现方式

```
        # 初始化derta
        derta = {}
        phi = {}
        for t in range(length):
            derta[t] = {}
            phi[t] = {}
            if t == 0:
                for i in range(n):
                    # 对于t时刻下的观测值，其是由i状态产生的概率是这样的
                    if seq[t] in B[i].keys():
                        derta[t][i] = pi[i] * B[i][seq[t]]
                    else:   # 如果之前从没有在i状态下观测到过B
                        # derta[t][i] = 0
                        if i == 0:
                            derta[t][i] = pi[i] * (1 / count_state[i]) # 尝试用(1 / count_state[j])代替B[j][seq[t]]
                        else:
                            derta[t][i] = 0
                    # 这里实际上直接应用到了HMM的两个基本假设
                    # 1.马尔科夫的，则当前状态的转移概率只与前一步的状态有关
                    # 2.观测值是独立产生的，也就是只与当前的状态i有关，与其他均无关
                    phi[t][i] = 0

        # 更新迭代
        for t in range(1, length):
            for j in range(n):
                max_derta_1 = -100000
                max_phi = 0
                for i in range(n):  # 遍历上一个时刻，状态i
                    # Attention!  temp没有乘以B[j][O_t]
                    temp = derta[t-1][i] * A[i][j]
```

```
                    if max_derta_1 < temp:
                        max_derta_1 = temp
                        max_phi = i
                if seq[t] in B[j].keys():
                    derta[t][j] = max_derta_1 * B[j][seq[t]]
                else:   # 如果在j状态下从没观察到过当前的观测值
                    # derta[t][j] = 0
                    if j == 0:     # TODO:当完全得到了模型评价之后，再来看如果没出现过当前的观测值应该怎么处理
                        derta[t][j] = max_derta_1 * (1 / count_state[j]) # 尝试用(1 / count_state[j])代替B[j][seq[t]]
                    else:
                        derta[t][j] = 0
                # 更新当前的phi值
                # derta[t][0] = 1
                phi[t][j] = max_phi
```

在这个情况下，可以得到模型评价结果如下，同时可见`output版本2.txt`文件

```
Label          Precision      Recall         F1-score
O              0.943051       0.917054       0.929871
B-ADDRESS      0.511905       0.461126       0.485190
I-ADDRESS      0.580622       0.604214       0.592183
B-BOOK         0.618644       0.474026       0.536765
I-BOOK         0.622977       0.438997       0.515050
B-COM          0.629333       0.624339       0.626826
I-COM          0.574553       0.659316       0.614023
B-GAME         0.649860       0.786441       0.711656
I-GAME         0.678295       0.770925       0.721649
B-GOV          0.492163       0.635628       0.554770
I-GOV          0.567531       0.779026       0.656669
B-MOVIE        0.582822       0.629139       0.605096
I-MOVIE        0.637014       0.698430       0.666310
B-NAME         0.684989       0.696774       0.690832
I-NAME         0.546851       0.697356       0.613000
B-ORG          0.611260       0.621253       0.616216
I-ORG          0.558681       0.529899       0.543909
B-POS          0.595833       0.660508       0.626506
I-POS          0.611628       0.684896       0.646192
B-SCENE        0.480000       0.459330       0.469438
I-SCENE        0.602826       0.531856       0.565121
```

这个细节的两个处理方法评价结果差异并不大（对比output版本1.txt（即第三部分）和output版本2.txt（第二种处理方式）），原因可能是O状态占了大多数情况，而其他状态均较少。当我们认为在验证数据中的某个状态下出现新的某个字符，将这个字符出现的原因只归因为“其他标签”也是十分合理的。这与将该字符出现归因到所有标签状态下均分概率，差别并不大。