机器学习分词中基于最大熵模型的算法改进与应用

姓名：黄中天　　学号：1951476

一、最大熵算法 MEM 概要

1.模型概述

假设有 n 个特征函数 $f_i(x, y)$，就有 n 个约束函数

满足所有约束条件的模型集合为：

$$C \equiv \{P \in \rho \mid E_{\tilde{p}}(f_i) = E_p(f_i), i = 1,2,...,n\}$$

定义在条件概率分布 P(Y|X)上的条件熵为：

$$H(P) = -\sum_{x,y} \tilde{P}(x)P(y \mid x)\log P(y \mid x)$$

则模型集合 C 中条件熵 H(P)最大的模型称为最大熵模型

$$\begin{aligned} \min_{P \in C} \quad & \sum_{x,y} \widetilde{P}(x)P(y|x)logP(y|x) \\ s.t. \quad & E_p(f_i) = E_{\widetilde{P}}(f_i) \\ & \sum_y P(y|x) = 1 \end{aligned}$$

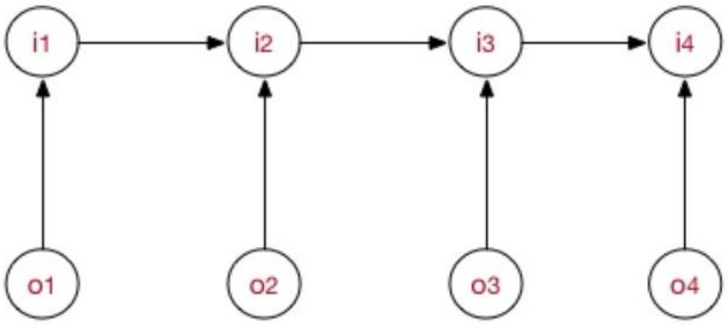
2. MEM 模型的缺点

约束函数数量会随着样本量的增大而增大，样本量很大的时候，对偶函数优化求解的迭代过程非常慢，实际应用比较难

二、最大熵隐马尔可夫模型 MEMM

1.模型概述

MEMM 的思想是找到一个满足马尔可夫奇次性假设、观测不独立且熵最大的模型解决序列标注问题，模型图结构如下：



MEMM 对条件概率直接建模，模型表示为

$$P(\boldsymbol{s}|\boldsymbol{o}) = \prod_t P(s_t|s_{t-1}, o_t)$$

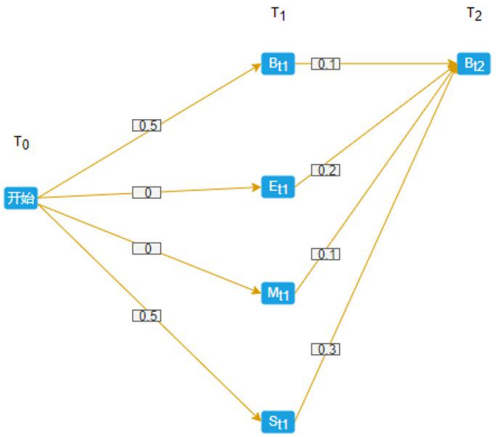
相比 HMM，MEMM 没有观测独立性假设。若不考虑整个序列式，时刻 t tt 的隐状态可看做一个分类问题，我们采用最大熵模型建模

$$P(s_t = i|s_{t-1}, o_t) = \frac{1}{Z(o_t, s_{t-1})} \exp\left(\sum_k \lambda_k f_k(o_t, s_t = i)\right), \quad Z(o_t, s_{t-1}) = \sum_i \exp\left(\sum_k \lambda_k f_k(o_t, s_t = i)\right)$$

2. 维比特算法的应用——动态规划思想优化预测模型

动态规划

如果最优路径在时刻 T_t 通过节点 T_t^i 。那么这一路径从节点 T_0 到节点 T_t^i 之间的路径一定是最优的。



- 定义一个变量 δ 来存放最优概率
  - $\delta_{B2} = 0.5 \times 0.3 = 0.15$
- 定义一个变量 tab=[{}] 来表示 δ 值
  - tab[T2]['B2']=0.15
    - T2 表示 T2 时刻
    - 'B2' 表示状态为 'B2'
- 定义一个变量 Ψ 来存放最优路径
- 定义一个变量 path={} 来表示 Ψ 值
  - path['B2']=['S1','B2']
    - 'S1' 表示上一个时刻经过的状态节点
    - 'B2' 表示当前状态节点

3. 改进算法 MEMM 的优点

最大熵马尔科夫模型把HMM模型和maximum-entropy模型的优点集合成一个产生式模型，这个模型允许状态转移概率依赖于序列中彼此之间非独立的特征上，从而将上下文信息引入到模型的学习和识别过程中，提高了识别的精确度，召回率也大大的提高。

三、算法实现主要函数

1.get_tag() 语料库标记函数

```
def get_tag(word):
    tags = []  # 创建一个空列表用来存放标注数据
    word_len = len(word)
    if word_len == 1:  # 如果是单字成词，标记为 S
        tags = ['S']
    elif word_len == 2:  # 如果该词仅有两个字，则标记为 B 和 E
        tags = ['B', 'E']
    else:
        tags.append('B')  # 第一个字标记为 B
        tags.extend(['M']*(len(word)-2))  # 中间标记为 M ，
        tags.append('E')  # 最后一个标记为 E
    return tags
```

测试结果：

```
get_tag('同济大学')

✓ 0.5s

['B', 'M', 'M', 'E']
```

2.pre_data() 数据集预处理函数

```
def pre_data(data):
    X = []  # 创建一个空列表来存放每个中文句子
    y = []  # 创建一个空列表来存放每个句子标注结果
    word_dict = []  # 创建一个空列表来存放每个句子的正确分词结果
    for sentence in data:
        sentence = sentence.strip()
        if not sentence:
            continue
```

```
        # 将句子按空格进行切分，得到词
        words = sentence.split("  ")
        word_dict.append(words)
        sent = []  # 用于临时存放一个中文句子
        tags = []  # 用于临时存放一个句子对应的标注
        for word in words:
            sent.extend(list(word))
            tags.extend(get_tag(word))  # 获得标注结果
        X.append(sent)
        y.append(tags)
    return X, y, word_dict


train_data = sentence_list[:-60]
test_data = sentence_list[-60:]
```

测试结果：

```
train_X, train_y, train_word_dict = pre_data(train_data)
test_X, test_y, test_word_dict = pre_data(test_data)

print(train_X[0])
print('-'*100)
print(train_y[0])
print('-'*100)
print(train_word_dict[0])
```

✓ 1.7s

```
['迈', '向', '充', '满', '希', '望', '的', '新', '世', '纪', '—', '—', '一', '九', '九', '八', '年', '新', '年', '讲', '话', '（', '附', '图
----------------------------------------------------------------------------------------------------
['B', 'E', 'B', 'E', 'B', 'E', 'S', 'S', 'B', 'E', 'B', 'E', 'B', 'M', 'M', 'M', 'E', 'B', 'E', 'B', 'E', 'S', 'S', 'B', 'E', 'S', 'S',
----------------------------------------------------------------------------------------------------
['迈向', '充满', '希望', '的', '新', '世纪', '—', '一九九八年', '新年', '讲话', '（', '附', '图片', '1', '张', '）']
```

3. para_init 矩阵初始化函数

```
states = {'B', 'M', 'E', 'S'}


def para_init():
    init_mat = {}  # 初始状态矩阵
    emit_mat = {}  # 发射矩阵
    tran_mat = {}  # 转移状态矩阵
    state_count = {}  # 用于统计每个隐藏状态（即 B,M,E,S）出现的次数
    for state in states:
        tran_mat[state] = {}
        for state1 in states:
            tran_mat[state][state1] = 0.0  # 初始化转移状态矩阵
        emit_mat[state] = {}  # 初始化发射矩阵
        init_mat[state] = 0.0  # 初始化初始状态矩阵
        state_count[state] = 0.0  # 初始化状态计数变量
```

```
    return init_mat, emit_mat, tran_mat, state_count
```

测设结果：

```
    import pandas as pd
    init_mat, emit_mat, tran_mat, state_count = para_init()
    print(pd.DataFrame(init_mat, index=['init']))
    print('-'*100)
    print(pd.DataFrame(tran_mat).T)
    print('-'*100)
    print((pd.DataFrame(emit_mat)).T)
```

✓ 1.4s

```
        E    B    S    M
init  0.0  0.0  0.0  0.0
----------------------------------------------------------------------------------------------------
     E    B    S    M
E  0.0  0.0  0.0  0.0
B  0.0  0.0  0.0  0.0
S  0.0  0.0  0.0  0.0
M  0.0  0.0  0.0  0.0
----------------------------------------------------------------------------------------------------
Empty DataFrame
Columns: []
Index: [E, B, S, M]
```

4. count（）统计函数

```
def count(train_X, train_y):
    """
    train_X: 中文句子
    train_Y: 句子对应的标注
    """
    # 初始化三个矩阵
    init_mat, emit_mat, tran_mat, state_count = para_init()
    sent_count = 0
    for j in range(len(train_X)):
        # 每次取一个句子进行统计
        sentence = train_X[j]
        sent_state = train_y[j]
        for i in range(len(sent_state)):
            if i == 0:
                # 统计每个状态（即 B,M,E,S）在每个句子对应的标注序列中第一个位置的次数
                init_mat[sent_state[i]] += 1
                # 统计每个隐藏状态（即 B,M,E,S）在整个训练样本中出现的次数
                state_count[sent_state[i]] += 1
                # 统计有多少个句子。
                sent_count += 1
            else:
                # 统计两个相邻时刻的不同状态组合同时出现的次数
                tran_mat[sent_state[i-1]][sent_state[i]] += 1
                state_count[sent_state[i]] += 1
                # 统计每个状态对应于每个文字的次数
```

```
            if sentence[i] not in emit_mat[sent_state[i]]:
                emit_mat[sent_state[i]][sentence[i]] = 1
            else:
                emit_mat[sent_state[i]][sentence[i]] += 1
    return init_mat, emit_mat, tran_mat, state_count, sent_count
```

测试结果：

```
init_mat, emit_mat, tran_mat, state_count, sent_count = count(train_X, train_y)
print(pd.DataFrame(init_mat, index=['init']))
print('-'*100)
print(pd.DataFrame(tran_mat).T)
print('-'*100)
# 随机取 6 个观测字
print((pd.DataFrame(emit_mat)).iloc[94:100, :].T)
```

✓ 1.6s

```
        E        B        S    M
init  0.0  26411.0  17017.0  0.0
----------------------------------------------------------------------------------------------------
          E         B         S        M
E       0.0  282122.0  270028.0      0.0
B  499035.0       0.0       0.0  85889.0
S       0.0  276391.0  201616.0      0.0
M   85889.0       0.0       0.0  45366.0
----------------------------------------------------------------------------------------------------
        得       成       活       步       善       术
E  1477.0  1981.0   927.0  1244.0  512.0  1813.0
B   584.0  2808.0  1265.0   231.0   86.0    12.0
S   840.0   482.0    77.0   122.0   13.0     5.0
M   146.0   177.0    62.0    46.0   13.0   286.0
```

5.predict（）最大熵优化概率函数

```
def predict(sentence, tran_prob_mat, emit_prob_mat, init_prob_mat):
    tab = [{}]  # 用于存放对应节点的
    path = {}   # 用于存放对应节点所经过的最优路径
    # 求出 T0 时刻的
    for state in states:
        tab[0][state] = init_prob_mat.get(
            state) * emit_prob_mat[state].get(sentence[0], 0.000000001)
        path[state] = [state]
    for t in range(1, len(sentence)):
        tab.append({})  # 创建一个元组来存放
        new_path = {}
        for state1 in states:
            # state1 为后一个时刻的状态
            items = []
            for state2 in states:
                # state2 为前一个时刻的状态
                if tab[t - 1][state2] == 0:
```

```
                    continue
                # 计算上一个时刻状态 state2 到当前时刻的状态 state1 的转移概率值
                tr_prob = tran_prob_mat[state2].get(
                    state1, 0.000000001) * emit_prob_mat[state1].get(sentence[t], 0.0000001)
                # 计算当前的状态为 state1 时经过上一个时刻状态 state2 的概率值
                prob = tab[t - 1][state2] * tr_prob
                items.append((prob, state2))
            if not items:
                items.append((0.000000001, 'S'))
            # 求出某个时刻的每个状态节点对应的最优路径
            best = max(items)  # best: (prob, state)
            tab[t][state1] = best[0]
            new_path[state1] = path[best[1]] + [state1]
        path = new_path
    # 寻找最后一个时刻最大的
    prob, state = max([(tab[len(sentence) - 1][state], state)
                       for state in states])
    return path[state]  # 返回最大
```

测试结果：

```
sentence = "我喜欢中文信息处理"
tag = predict(sentence, tran_prob_mat, emit_prob_mat, init_prob_mat)
' '.join(tag)
```

✓ 0.5s

```
'S B E B E B E B E'
```

6.main 函数输出

```
while True:
  sentence = input("请输入分词语句：\n")
  result = word_seg(sentence, tran_prob_mat, emit_prob_mat, init_prob_mat)
  a=' | '.join(result)

  print(a)
```

调试结果：

```
问题  输出  调试控制台  终端  JUPYTER: VARIABLES

    code = compile(f.read(), fname, 'exec')
  File "e:\中文语言处理\中文语言处理\作业\homework2.py", line 312
    sentence = input("请输入分词语句：\n")
    ^
IndentationError: expected an indented block
PS E:\中文语言处理\中文语言处理\作业> e:; cd 'e:\中文语言处理\中文语言处理\作业'; & 'E:\Anaconda\python.exe' 'c:\Users\admin\.vscode\extensions\ms-python.python-2022.2
.1924087327\pythonFiles\lib\python\debugpy\launcher' '1968' '--' 'e:\中文语言处理\中文语言处理\作业\homework2.py'
请输入分词语句：
西方人的国家政府基本都是服务型政府，公务员服务的不好，马上就被选票干掉，再换一批公务员。政府用于养老的投入比较透明。税收基本实现用之于民。本国公民没有后顾之忧，所以西方
人生老病死都有政府部门的基本保障，所以西方百姓不存钱，该吃喝玩乐的地方一点都不吝啬。不会担心没钱看不起病，更不操心将来以后谁来养老的问题。
西方 | 人 | 的 | 国家 | 政府 | 基本 | 都 | 是 | 服务型 | 政府 | ， | 公务员 | 服务 | 的 | 不好 | ， | 马上 | 就 | 被 | 选票 | 干掉 | ， | 再换 | 一 | 批 | 公务员 | 。 |
 政府 | 用于 | 养老 | 的 | 投入 | 比 | 较 | 透明 | 。 | 税收 | 基本 | 实现 | 用 | 之于 | 民。 | 本国 | 公民 | 没有 | 后 | 顾 | 之忧 | ， | 所 | 以 | 西方 | 人生 | 老病
| 死 | 都 | 有 | 政府 | 部门 | 的 | 基本 | 保障 | ， | 所 | 以 | 西方 | 百姓 | 不存 | 钱 | ， | 该 | 不存钱，该吃 | 喝 | 玩乐 | 的 | 地方 | 一点 | 都 | 不 | 吝啬 | 。 |
 不会 | 担心 | 没钱 | 看 | 不 | 起病 | ， | 更 | 不 | 操心 | 将 | 来 | 以后 | 谁 | 来养 | 老 | 的 | 问题 | 。
请输入分词语句：
```

四、语料资源和测试用例

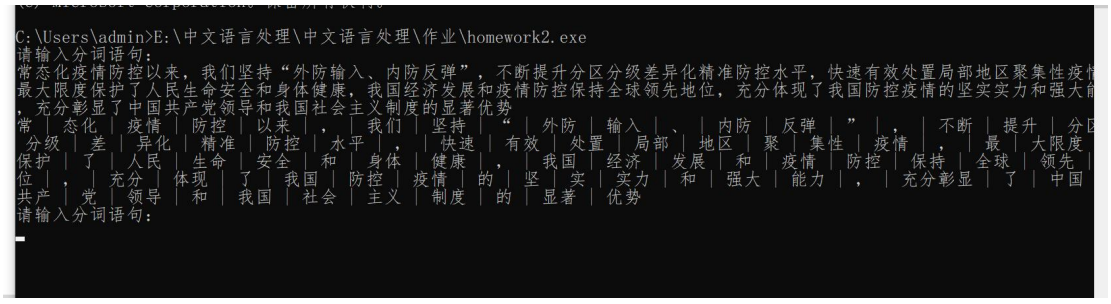
1. 语料资源

《人民日报》199801-199806 数据集

2. 测试结果

将我编写的算法实现结果和哈工大分词系统[1]的结果进行对比，其中样例文字资料来自与网络。

A：[2]



B:

3. 评分和准确率

我在“我爱自然语言处理”网站下载了测试集和测试程序，用命令行进行了如下操作：

pku_training_words.txt < ../testing/pku_test.txt > pku_test_seg.txt

其中第一个参数需提供一个词表文件 pku_training_word.txt，输入为 pku_test.txt，输出为 pku_test_seg.txt。利用 score 评分的命令如下：

pku_training_words.txt >pku_test_gold.txt pku_test_seg.txt > score.txt

其中前三个参数已介绍，而 score.txt 则包含了详细的评分结果，不仅有总的评分结果，还包括每一句的对比结果。总评结果：

```
INSERTIONS:         1
DELETIONS:          2
SUBSTITUTIONS:      9
NCHANGE:            12
NTRUTH: 27
NTEST:    26
TRUE WORDS RECALL:          0.593
TEST WORDS PRECISION:       0.615
=== SUMMARY:
=== TOTAL INSERTIONS:       4877
=== TOTAL DELETIONS:        6615
=== TOTAL SUBSTITUTIONS:            15838
=== TOTAL NCHANGE:          27330
=== TOTAL TRUE WORD COUNT:          104372
=== TOTAL TEST WORD COUNT:          102634
=== TOTAL TRUE WORDS RECALL:        0.785
=== TOTAL TEST WORDS PRECISION: 0.798
=== F MEASURE:    0.791
=== OOV Rate:     0.058
=== OOV Recall Rate:          0.326
=== IV Recall Rate:0.813
###         pku_test_seg.txt    4877    6615    15838   27330   104372  102634  0.785   0.798   0.791   0.058   0.326   0.813
```

五、遇到的一些问题和心得体会

我的专业是电子信息工程，不是所谓科班出身，这次的作业可以说是被迫捡起了高程和算法的知识，可谓收获还是很大的。写完程序之后，用训练集测试结果不错，达到了 80%以上可以说还是很让人满意的。

当然也遇到了一些困难比如 jupyter 经常报错 kernel error，觉得应该是一个 win32security 的内核有问题，install 之后还是报错，后来重装了 python 内核才解决。还有 predict（）函数的编写，因为是创新的点，所以编写难度大了些。我查了一些资料，结合数据结构的内容，做了一些运行速度上的优化。

六、系统的优缺点以及分析改进

HMM

HMM 是最早提出来的动机是为了类似解决序列标注问题的一个理想模型，也是一个基于 P(X|Y)建模的概率生成模型。之所以说理想，是因为它的核心思想是建立在两个假设上面：一阶齐次马尔可夫假设、观测独立假设。

显然这两个假设都是离实际偏差比较大，因此它的优缺点和适用场景也很明显：

优点：

有了这两个假设，假设成立的场景，可以大大简化概率 P(X|Y)的计算。

缺点：

- 不适用于一般场景，应用范围比较窄。
- 因为是生成模型，因为不是判别标注类别，不如概率判别模型计算量小。

MEMM

MEMM 是最大熵马尔可夫模型，向较于 HMM，它出现的动机主要是打破了 HMM 的观测独立假设，拓宽了实际应用的场景范围。同时，它还是一个基于 P(Y|X)的概率判别模型。所以它的核心思想是：改造 HMM，打破 HMM 的观测独立假设。

它的优缺点和适用场景也是相对于 HMM 来说：

优点：

• 没有观测独立假设，很明显它比 HMM 有更宽的适用场景

• 因为是概率判别模型，直接对 P(Y|X)计算，不需要算联合概率等中间步骤，计算量小。

缺点：

• MEMM 有一个致命的问题，即标注偏差问题，导致这个问题的原因是局部归一化，因为这个问题,限制了它的实际使用，实际用途不大。

算法改进方向——双向 MEMM

事实上，相比 CRF，MEMM 明显的一个不够漂亮的地方就是它的不对称性——它是从左往右进行概率分解的。笔者的实验表明，如果能解决这个不对称性，能够稍微提高 MEMM 的效果。笔者的思路是：将 MEMM 也从右往左做一遍，这时候对应的概率分布是

$$P(\boldsymbol{y}|\boldsymbol{x}) = \frac{e^{f(y_1;\boldsymbol{x})+g(y_1,y_2)+\cdots+g(y_{n-1},y_n)+f(y_n;\boldsymbol{x})}}{\left(\sum\limits_{y_n} e^{f(y_n;\boldsymbol{x})}\right)\left(\sum\limits_{y_{n-1}} e^{g(y_n,y_{n-1})+f(y_{n-1};\boldsymbol{x})}\right)\cdots\left(\sum\limits_{y_1} e^{g(y_2,y_1)+f(y_1;\boldsymbol{x})}\right)}$$

然后也算一个交叉熵，跟从左往右的式的交叉熵平均 s。这样一来，模型同时考虑了从左往右和从右往左两个方向，并且也不用增加参数，弥补了不对称性的缺陷。

七、引用

[1] 哈工大语言技术平台云： https://www.ltp-cloud.com/

[2] 习近平将应约于北京时间 3 月 18 日晚同美国总统拜登就中美关系和双方共同关心的问题交换意见《 人民日报 》（ 2022 年 03 月 18 日 第 01 版）