AI-PJ2-report

柳世纯 20307130111

1. 实验目标

本次实验需要对完成命名实体识别（NER）任务。NER任务要求标注句子中哪些词属于一个实体，包括名词、国籍、头衔等。本次任务需要用HMM、CRF、BiLSTM-CRF三种方式来实现。

2. 数据处理

实验数据集中每一行是一个字(词)和它对应的实体类别，二者之间用空格隔开。一句话中间用空行隔开。数据处理时将一句话处理为如下格式。

```
[['现', '任', '上', '海', '大', '盛', '资', '产', '有', '限', '公', '司', '财', '务', '部', '总', '经', '理', '。'], ['O', 'O', 'B-ORG', 'M-ORG', 'M-ORG', 'M-ORG', 'M-ORG', 'M-ORG', 'M-ORG', 'M-ORG', 'E-ORG', 'B-TITLE', 'M-TITLE', 'M-TITLE', 'M-TITLE', 'E-TITLE', 'O']]
```

相关代码

```python
def DataProcess(path: str) -> (list, list, list):
    data = []
    sentence = []
    tag = []
    s = []
    t = []
    with open(path, "r", encoding="utf-8") as file:
        for line in file:
            if line != "\n":
                line = line.rstrip().split()
                sentence.append(line[0])
```

```
                tag.append(line[1])
            else:
                data.append([sentence, tag])
                s.append(sentence)
                t.append(tag)
                sentence = []
                tag = []
    return data, s, t
```

下面分别介绍HMM、CRF、BiLSTM-CRF三种实现方式。

3. HMM

3.1 原理

参考文章：https://zhuanlan.zhihu.com/p/110989180

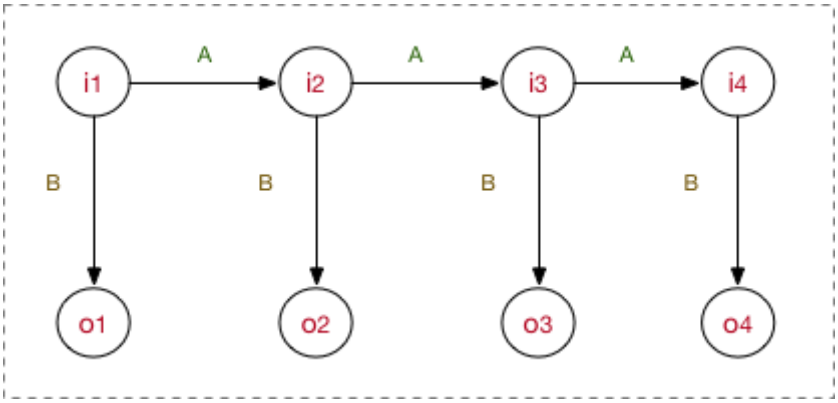
隐马尔科夫模型（HMM）描述由隐藏的马尔科夫链随机生成观测序列的过程，属于**生成模型**。

HMM模型是关于**时序**的**概率模型**，描述由一个**隐藏的马尔科夫链随机生成不可观测的状态随机序列**，再由各个状态生成一个观测从而产生观测随机序列的过程，隐藏的马尔科夫链随机生成的状态的序列，称为**状态序列**；每个状态生成一个规则，而由此产生的观测的随机序列称为**观测序列**。序列的每一个位置又可以看作是一个时刻。

HMM模型由初始概率分布、状态转移概率分布以及观测概率分布三者共同决定。HMM的定义简历在两个基本假设之上：

（1）齐次马尔科夫性假设，即假设隐藏的马尔科夫链在任意时刻t的状态**只依赖于前一时刻的状态**，与其他时刻的状态及观测无关，其余时刻无关。换句话说，隐马尔科夫链中的每个状态都是由前一个时刻的状态决定的。

（2）观测独立性假设，即假设任意时刻的观测**只依赖于该时刻的马尔科夫链的状态**，与其他观测以及状态无关。



根据这两个假设，给出HMM模型包含：

- **隐藏状态集**N：$N=\{q_1,\cdots,q_N\}$，也就是说隐藏节点只能限定取包含在隐藏状态集中的符号。
- **观测集**M：$M=\{v_1,\cdots,v_M\}$，观测节点的状态也只能限定取包含在观测状态集中的符号。
- **状态转移概率矩阵**A：$A=[a_{ij}]_{N\times N}$（N为隐藏状态集元素个数），表示由一种隐藏状态到另一种隐藏状态的概率。
- **观测概率矩阵**B：$B=[b_{ij}]_{N\times M}$，也称为**发射概率**，表示对于某一种隐藏状态，到各个观测状态的概率。
- **初始概率**π：$\pi=\{p_1,\cdots,p_N\}$，表示第一个隐状态节点属于不同状态的概率。

此次实验中，给定 (A, B, π) 和观测序列 O，求最有可能的对应的状态序列。在这个问题中我们认为每一个tag代表一种状态，而字(词)则是当前状态下的可能观测结果。首先求出模型参数 A, B, π，可以直接从训练数据当中**统计**得到。

3.2 核心代码

利用统计方法，得到模型参数 (A, B, π)；由于 A, B, π 中各个位置的数之间差距可能非常大，因此需要取对数。先将所有的0加上一个1e-8，再取对数。

```
def train(data, A, B, pi, tag2idx, char2idx ):
    for i in tqdm(range(len(data))):  # 几组数据
        for j in range(len(data[i][0])):  # 每组数据中几个字符
            cur_char = data[i][0][j]  # 取出当前字符
            cur_tag = data[i][1][j]  # 取出当前标签
            B[tag2idx[cur_tag]][char2idx[cur_char]] += 1  # 对B矩阵中标签->字符的位
置加一
            if j == 0:
                # 若是文本段的第一个字符，统计pi矩阵
                pi[tag2idx[cur_tag]] += 1
                continue
            pre_tag = data[i][1][j - 1]  # 记录前一个字符的标签
            # 对A矩阵中前一个标签->当前标签的位置加一
            A[tag2idx[pre_tag]][tag2idx[cur_tag]] += 1
    # 先将0加上一个1e-8，再取对数
    A[A == 0] = 1e-8
    A = np.log(A) - np.log(np.sum(A, axis=1, keepdims=True))
    B[B == 0] = 1e-8
    B = np.log(B) - np.log(np.sum(B, axis=1, keepdims=True))
    pi[pi == 0] = 1e-8
    pi = np.log(pi) - np.log(np.sum(pi))

    return A, B, pi
```

求出模型参数之后，通过viterbi算法就可以求出一个观测序列对应的最大可能状态序列。

viterbi算法的基本思想是动态规划：假设t+1时刻的路径是当前最优路径，那么这一条路径上到t时刻的路径也是0-t的最优路径。因此，假设在t时刻已经保存了从0到t时刻的最优路径，那么t+1时刻只需要计算从t到t+1的最优就可以了。每次只需要保存到当前位置最优路径，之后循环向后走。到结束时，从最后一个时刻的最优值回溯到开始位置，回溯完成后，这个从开始到结束的路径就是最优的。

```
def viterbi(A, B, pi, s, word2idx, idx2tag):
    delta = pi + B[:, word2idx.setdefault(s[0], -1)]
    # 前向传播记录路径
    path = []
    for i in range(1, len(s)):
        # 广播机制，重复加到A矩阵每一列
        tmp = delta.reshape(-1, 1) + A
        # 取最大值作为节点值，并加上B矩阵
        delta = np.max(tmp, axis=0) + B[:, word2idx.setdefault(s[i], -1)]
        # 记录当前层每一个节点的最大值来自前一层哪个节点
        path.append(np.argmax(tmp, axis=0))

    # 回溯，先找到最后一层概率最大的索引
    index = np.argmax(delta)
    best_path = [idx2tag[index]]
    # 逐层回溯，沿着path找到起点
```

```
    while path:
        tmp = path.pop()
        index = tmp[index]
        best_path.append(idx2tag[index])
    # 序列翻转
    best_path.reverse()
    return best_path
```

3.3 结果

```
中文结果:
              precision    recall  f1-score   support

      B-NAME     0.8922    0.8922    0.8922       102
       ...
       S-LOC     0.0000    0.0000    0.0000         0

   micro avg     0.8610    0.8859    0.8732      8437
   macro avg     0.5773    0.7051    0.6148      8437
weighted avg     0.8705    0.8859    0.8775      8437

英文结果:
              precision    recall  f1-score   support

       B-PER     0.9572    0.6916    0.8030      1842
       ...
      I-MISC     0.3059    0.7197    0.4293       346

   micro avg     0.7114    0.7694    0.7393      8603
   macro avg     0.6656    0.7630    0.6908      8603
weighted avg     0.7740    0.7694    0.7570      8603
```

4. CRF

4.1 原理

概率无向图模型(probabilisticundirected graphical model)，又称为马尔可夫随机场(Markov random field)， 是一个可以由无向图表示的联合概率分布。

条件随机场(conditional random field)是给定随机变量X条件下， 随机变量Y的马尔可夫随机场。**这里主要介绍定义在线性链上的特殊的条件随机场，称为线性链条件随机场(Linear chain conditional random field)。线性链条件随机场可以用于标注等问题。**

这时，**在条件概率模型$P(Y|X)$中，Y是输出变量，表示标记序列，X是输入变量，表示需要标注的观测序列。**也把标记序列称为状态序列(参见隐马尔可夫模型)。学习时，利用训练数据集通过**极大似然估计或正则化的极大似然估计**得到条件概率模型 $P(Y|X)$ ；预测时，对于给定的输入序列x, 求出条件概率 $P(y|x)$ 最大的输出序列y。

在本次任务中我们需要使用线性链条件随机场，引入特征函数，定义条件概率：

$$P(Y \mid X)=\frac{1}{Z} \exp \left(\sum_{j=1}^{K_1} \sum_{i=1}^{n-1} \lambda_j t_j\left(Y_i, Y_{i+1}, X, i\right)+\sum_{k=1}^{K_2} \sum_{i=1}^{n} \mu_k s_k\left(Y_i, X, i\right)\right)$$

- $t_j(Y_i, Y_{i+1}, X, i)$：已知观测序列的情况下，两个相邻标记位置上的转移特征函数；

- $s_k(Y_i, X, i)$：在已知观测序列的情况下，标记位置 i 上的状态特征函数;
- Z 表示对所有可能的标记序列进行求和

4.2 核心代码

确定好所需要训练的数据集之后，最重要的步骤就是定义特征模板。本次任务中主要使用了词的上下文信息来构造特征模板，构造方式参考了官方示例。
中文特征：中文数据特征模板主要判断每个字的前后一个字的内容和它们是否为数字

```
def word2features(sent, i):
    word = sent[i]
    features = {
        "bias": 1.0,
        "word": word,
        "word.isdigit()": word.isdigit(),
    }

    if i > 0:
        word = sent[i - 1]
        features.update(
            {
                "-1:word": word,
                "-1:word.isdigit()": word.isdigit(),
            }
        )
    else:
        features["BOS"] = True
    if i < len(sent) - 1:
        word = sent[i + 1]
        features.update(
            {
                "+1:word": word,
                "+1:word.isdigit()": word.isdigit(),
            }
        )
    else:
        features["EOS"] = True
    return features
```

英文特征：中英文数据特征模板主要判断每个词的前后一个词的内容和它们是否为数字、是否为标题、是否为大写。

定义sent2features和sent2labels函数，从数据集中提取特征

```
def sent2features(sent):
    return [word2features(sent, i) for i in range(len(sent))]


def sent2labels(sent):
    return [label for label in sent]
```

模型的定义与训练

```
crf_model = sklearn_crfsuite.CRF(
    algorithm="lbfgs",
```

```
    c1=0.1,
    c2=0.1,
    max_iterations=50,
    all_possible_transitions=True,
    verbose=True,
)
print("开始训练数据：")
crf_model.fit(X_train, y_train)
print("训练结束")

print("开始预测！")
labels = list(crf_model.classes_)
y_pred = crf_model.predict(X_dev)
```

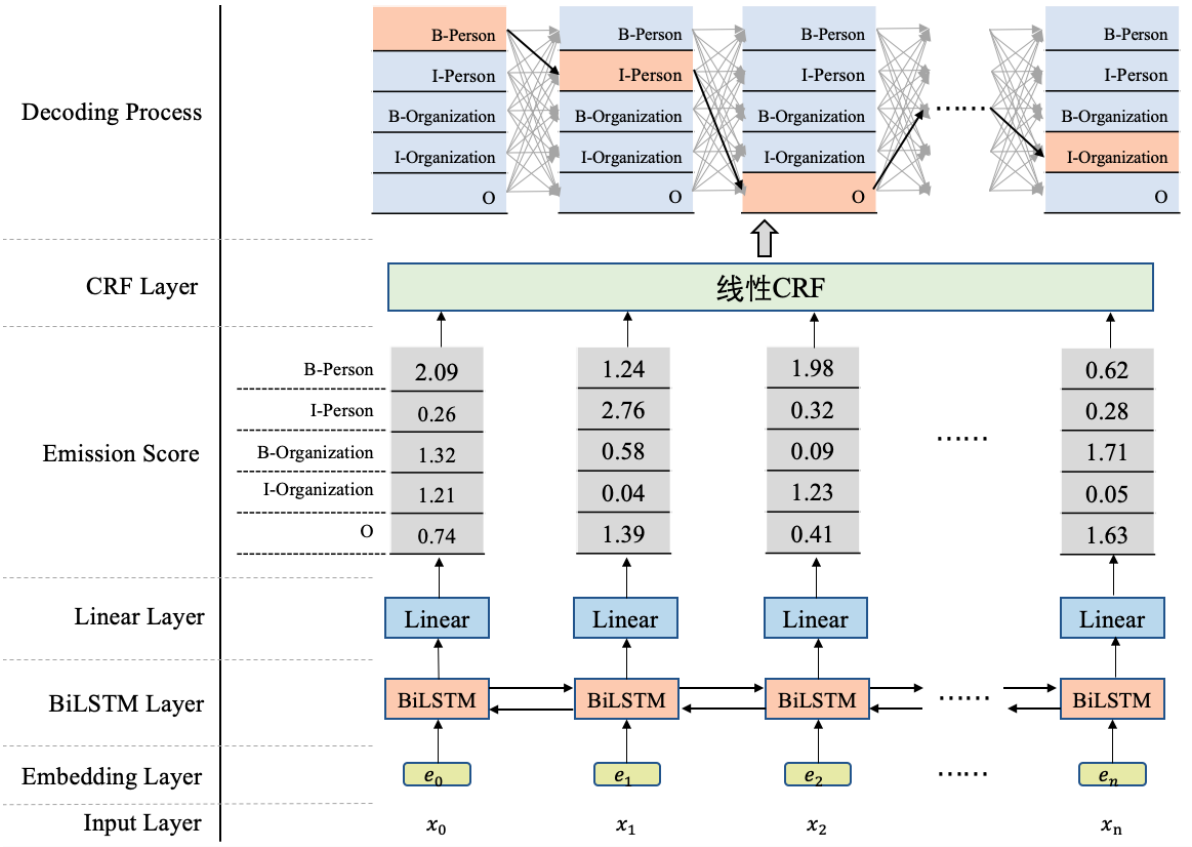
4.3 实验结果

```
中文结果:
              precision    recall  f1-score   support

      B-NAME     0.9901    0.9804    0.9852       102
      ...
       S-LOC     0.0000    0.0000    0.0000         0

   micro avg     0.9254    0.9423    0.9338      8437
   macro avg     0.7177    0.7233    0.7204      8437
weighted avg     0.9259    0.9423    0.9338      8437

英文结果:
              precision    recall  f1-score   support

       B-PER     0.9015    0.8990    0.9002      1842
       ...
      I-MISC     0.8759    0.7341    0.7987       346

   micro avg     0.9039    0.8696    0.8864      8603
   macro avg     0.9035    0.8458    0.8726      8603
weighted avg     0.9042    0.8696    0.8860      8603
```

5. BiLSTM-CRF

5.1 原理

BiLSTM-CRF模型双向长短时记忆网络（BiLSTM)）和条件随机场（CRF）组成，模型输入是字符特征，输出是每个字符对应的预测标签。



BiLSTM-CRF模型结构大致如上图所示，文本输入经过embedding层后转化为向量形式输入双向LSTM层。LSTM层的结果经过一个线性层后的得到一个大小为状态数 * 文本长度的发射矩阵。矩阵每一个位置对应文本中的字由一个状态产生的分数，该分数称为发射分数。矩阵最后经过CRF层解码得到最大概率路径。

5.2 核心代码

参考：https://pytorch.org/tutorials/beginner/nlp/advanced_tutorial.html

https://blog.csdn.net/misite_J/article/details/109036725

整体架构，BiLSTM层获得发射矩阵（状态特征函数），CRF计算得到概率分布。

```
class BiLSTM_CRF(nn.Module):
    def __init__(self, embedding_dim, hidden_dim, vocab, label_map,
device='cpu'):
        super(BiLSTM_CRF, self).__init__()
        self.embedding_dim = embedding_dim  # 词向量维度
        self.hidden_dim = hidden_dim
        self.vocab_size = len(vocab)  # 词表大小
        self.tag_size = len(label_map)  # 标签个数
        self.device = device
        self.state = 'train'  # 模型有'train'、'eval'、'pred'三种状态

        self.word_embeds = nn.Embedding(self.vocab_size, embedding_dim)
        self.dropout = nn.Dropout(p=0.5, inplace=True)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim // 2, num_layers=2,
bidirectional=True, batch_first=True)
        self.hidden2tag = nn.Linear(hidden_dim, self.tag_size, bias=True)
        self.crf = CRF(label_map, device)
        self.layer_norm = nn.LayerNorm(self.hidden_dim)
```

CRF部分，学习转移特征函数，通过维特比算法得到最优路径。

计算归一化因子Z

```
def _forward(self, feats, seq_len):
        init_alphas = torch.full((self.tag_size,), -10000.)
        init_alphas[self.label_map[self.START_TAG]] = 0.

        forward = torch.zeros(feats.shape[0], feats.shape[1] + 1,
feats.shape[2], dtype=torch.float32,
                                  device=self.device)
        forward[:, 0, :] = init_alphas

        transitions = self.transitions.unsqueeze(0).repeat(feats.shape[0], 1, 1)
        for seq_i in range(feats.shape[1]):
            emit_score = feats[:, seq_i, :]
            tag_var = (
                    forward[:, seq_i, :].unsqueeze(1).repeat(1, feats.shape[2],
1)  # (batch_size, tag_size, tag_size)
                    + transitions
                    + emit_score.unsqueeze(2).repeat(1, 1, feats.shape[2])
            )
            cloned = forward.clone()
            cloned[:, seq_i + 1, :] = log_sum_exp(tag_var)
            forward = cloned

        forward = forward[range(feats.shape[0]), seq_len, :]
        last = forward +
self.transitions[self.label_map[self.STOP_TAG]].unsqueeze(0).repeat(feats.shape[
0], 1)
        alpha = log_sum_exp(last)
        return alpha
```

路径分数

```
def _score(self, feats, tags, seq_len):
        score = torch.zeros(feats.shape[0], device=self.device)
        start = torch.tensor([self.label_map[self.START_TAG]],
device=self.device).unsqueeze(0).repeat(feats.shape[0], 1)
        tags = torch.cat([start, tags], dim=1)
        for batch_i in range(feats.shape[0]):
            score[batch_i] = torch.sum(
                self.transitions[tags[batch_i, 1:seq_len[batch_i] + 1],
tags[batch_i, :seq_len[batch_i]]]) \
                             + torch.sum(feats[batch_i, range(seq_len[batch_i]),
tags[batch_i][1:seq_len[batch_i] + 1]])
            score[batch_i] += self.transitions[self.label_map[self.STOP_TAG],
tags[batch_i][seq_len[batch_i]]]
        return score
```

5.3 实验结果

中文结果:

	precision	recall	f1-score	support
B-NAME	0.9902	0.9902	0.9902	102
...				
S-LOC	0.0000	0.0000	0.0000	0
micro avg	0.9467	0.9599	0.9533	8437
macro avg	0.7092	0.7674	0.7316	8437
weighted avg	0.9475	0.9599	0.9535	8437

英文结果:

	precision	recall	f1-score	support
B-PER	0.9120	0.8105	0.8583	1842
...				
I-MISC	0.8764	0.6561	0.7504	346
micro avg	0.9261	0.7994	0.8581	8603
macro avg	0.9243	0.7784	0.8444	8603
weighted avg	0.9258	0.7994	0.8575	8603