Project2 HMM、CRF、Transformer+CRF

具体运行的完整代码在 `Code` 中，运行的 `JyputerNotebook` 可见 `run.pdf`

Part1 - HMM实现命名实体识别（NER）任务

由于我们的数据中既有隐藏状态序列，又有观察序列，则只要做统计就能得到估计的初始概率，转移概率矩阵和发射矩阵。为此，简单统计：

1. 每个序列第一个词的状态出现次数
2. 状态之间转移的次数
3. 同一状态发射到不同单词的次数

具体来说：

```
def _build_matrix(self,epsilon=1e-8):
        self.observation_num = len(self.word_index)
        self.state_num = len(self.label_index)
        self.pi = np.zeros(self.state_num)
        self.transfer = np.zeros((self.state_num,self.state_num))
        self.emission = np.zeros((self.state_num,self.observation_num))
        for seq in self.sequences:
            previous_label = None
            for idx,word in enumerate(seq["word"]):
                label = seq["label"][idx]
                self.emission[self.label_index[label],self.word_index[word]] += 1
                if idx == 0:
                    self.pi[self.label_index[label]] += 1
                else:
                    self.transfer[self.label_index[previous_label],self.label_index[label]] += 1
                previous_label = label
        # Avoid overflow
        self.pi[self.pi == 0] = epsilon
        self.pi /= self.pi.sum()
        self.transfer[self.transfer == 0] = epsilon
        self.transfer /= self.transfer.sum(axis=1, keepdims=True)
        self.emission[self.emission == 0] = epsilon
        self.emission /= self.emission.sum(axis=1, keepdims=True)
```

在统计完次数后，做 `softmax` 归一化变为概率即可。由于词空间巨大，很多数值为 `0` ，为了防止溢出，可以将所有的 `0` 加上 `1e-8` 。

`Viterbi` 算法是一种允许线性复杂度进行推理的动态规划算法。伪代码如下：

```
function VITERBI( O, S, π, Y, A, B ) : X
    for each state s_i do
        T_1[i,1] ← π_i · B_{iy_1}
        T_2[i,1] ← 0
    end for
    for i ← 2,3,...,T do
        for each state s_j do
            T_1[j,i] ← max_k (T_1[k,i−1] · A_kj · B_{jy_i})
            T_2[j,i] ← arg max_k (T_1[k,i−1] · A_kj · B_{jy_i})
        end for
    end for
    z_T ← arg max_k (T_1[k,T])
    x_T ← s_{z_T}
    for i ← T,T-1,...,2 do
        z_{i-1} ← T_2[z_i,i]
        x_{i-1} ← s_{z_{i-1}}
    end for
    return X
end function
```

具体来说，

```python
def _viterbi(self,seq):
        length = len(seq["word"])
        T1_table = np.zeros([length, self.state_num])
        T2_table = np.zeros([length, self.state_num])
        initial_state = self._get_state(seq["word"][0])
        T1_table[0, :] = self.pi + initial_state
        T2_table[0, :] = np.nan
        for i in range(1, length):
            state = self._get_state(seq["word"][i])
            state = np.expand_dims(state, axis=0)
            prev_score = np.expand_dims(T1_table[i-1, :], axis=-1)
            score = prev_score + self.transfer + state
            T1_table[i, :] = np.max(score, axis=0)
            T2_table[i, :] = np.argmax(score, axis=0)
        best_label = int(np.argmax(T1_table[-1, :]))
        best_labels = [best_label]
        for i in range(length-1, 0, -1):
            best_label = int(T2_table[i, best_label])
            best_labels.append(best_label)
```

```
        return list(reversed(best_labels))
```

最后进行简单测试，发现确实收敛：

```
              precision    recall  f1-score   support

   micro avg     0.8941    0.7476    0.8143      8603
   macro avg     0.8831    0.7318    0.7987      8603
weighted avg     0.8974    0.7476    0.8137      8603
```

English

```
...
              precision    recall  f1-score   support

   micro avg     0.8652    0.8882    0.8765      8437
   macro avg     0.5843    0.7061    0.6238      8437
weighted avg     0.8694    0.8882    0.8782      8437
```

Chinese

Part2 - CRF 实现命名实体识别（NER）任务

条件随机场作为一种特殊的随机场，其假定只有前后两个状态之间，通过人工挑选的特征生成分数，利用类似 `Logistic` 回归的方式得到各个标签的概率，**刻画观察与隐藏状态之间的关系**；辅以概率转移矩阵，**刻画隐藏状态之间的序列关系**。

我们这里直接选取 `pytorch` 默认的表征传入即可。

由于特征为人工挑选，可训练的参数只在概率转移矩阵中。

我们利用 `pycrfsuite` 库，可以看到其利用 `L-BFGS` 这一常见优化器优化概率转移矩阵。

推理仍然可以用 `Viterbi` 算法来进行推理。

具体来说，只要设置 `trainer` 即可：

```
for xseq, yseq in zip(X_train, y_train):
    trainer.append(xseq, yseq)

trainer.set_params({
    'c1': 1.0,
    'c2': 1e-3,
    'max_iterations': 100000,
})

```

```
trainer.train('crf_eng.model')
tagger = pycrfsuite.Tagger()
tagger.open('crf_eng.model')
y_pred_eng = [tagger.tag(xseq) for xseq in X_test]
tagger.close()
```

训练略微花费时间，但总体还比较快。结果确实收敛：

```
              precision    recall  f1-score   support

   micro avg     0.8816    0.8445    0.8626      8603
   macro avg     0.8763    0.8168    0.8443      8603
weighted avg     0.8821    0.8445    0.8622      8603
```

English

```
...
              precision    recall  f1-score   support

   micro avg     0.9346    0.9444    0.9395      8437
   macro avg     0.7233    0.7290    0.7257      8437
weighted avg     0.9349    0.9444    0.9395      8437
```

Chinese

Part3 - Transformer+CRF 实现命名实体识别 （NER）任务

在传统的 `CRF` 模型中，人为挑选的特征对于模型的性能十分重要。 `Transformer` 的介入相当于将“人为挑选特征得到分数- `Logistic回归` -得到概率”的模式改为“ `Transformer` 得到表征- `Feed Forward` 得到概率”。概率转移矩阵部分不变。

由于 `Transformer` 部分和概率转移矩阵要一同训练，不能使用较为高效的 `L-BFGS` 算法，这里就采用简单的 `SGD` 进行优化。

推理也可以用 `Viterbi` 算法来进行推理。

我们利用 `pytorch` 框架中的 `Transformer Encoder` 完成 `Transformer` 部分

```
self.transformer = nn.TransformerEncoder(nn.TransformerEncoderLayer(d_model=self.embedding_dim,nhead=nhead,dim_feedforward=self.hidden_dim
        ), num_layers=num_layers)
```

再额外训练一个转移概率矩阵：

```
self.transitions = nn.Parameter(torch.randn(self.num_label, self.num_label))
```

这只需要在 `forward` 中对 `Transformer` 获得的表征做即可。前向时仍然利用 `Viterbi` 算法。

```
def forward(self, sentence):
        feats = self._get_transformer_features(sentence)
        score, tag_seq = self._viterbi_decode(feats)
        return score, tag_seq
```

而在学习时，我们既可以像在 `HMM` 模型中那样直接统计获得概率转移矩阵，也可以跟随 `Transformer` 一起利用 `SGD` 训练。这里就简单利用 `SGD` 训练。

训练比较花费时间，下面证明其收敛：

```
   micro avg     0.5264    0.6076    0.5641      8603
   macro avg     0.5699    0.5839    0.5586      8603
weighted avg     0.5807    0.6076    0.5765      8603
```

English

```
...
   micro avg     0.8130    0.8649    0.8382      8437
   macro avg     0.5978    0.5866    0.5843      8437
weighted avg     0.8451    0.8649    0.8447      8437
```

Chinese

其中 `English` 在训练时epoch进行的较多，可以看到已经出现了明显的过拟合现象；相反，`Chinese` 只训练了1个epoch，准确率达到 `0.83` ，相当不错。

备注

具体运行结果都在 `run.pdf` 中可以看到。