Lab2实验报告

马成 20307130112

Part1 HMM

基本原理

1. HMM假设了前一次的转移情况仅仅和上一次状态有关。HMM认为有一个发生概率矩阵用于处理每一个标签生成不同的词汇的概率以及每一个标签在下一次转化成另一个标签的概率。同时在生成一个初始标签概率。
2. 如果已知了这些概率就可以使用vetebi解码，通过目前已知的单词序列动态规划求出这个序列对应的隐状态序列的概率最大值以及这个序列。由此就完成了序列标注任务
3. 为了得到1.中提到的那些概率矩阵可以直接对训练集进行统计计算来求出概率。

代码大致框架以及代码讲解

1. 对输入数据的预处理：全部读入数据，将每一个单词转化为一个数字，每一个标签也转化为数字后传回，同时传回字典以及标签和编号的对应关系

```
def init(language='English', mode='train'):
    if language == "English":
        sort_labels = sorted_labels_eng
    else:
        sort_labels = sorted_labels_chn
    tag2id = {}
    for s in sort_labels:
        tag2id[s] = len(tag2id)
    f = open('./NER/' + language + '/' + mode + '.txt', 'r', encoding='utf-
8')
    word2id = {}
    sentences = []
    sentence = []
    tags = []
    tag_list = []
    for i in range(10000000):
        s = f.readline()
        if s == '':
            break
        s = s[:-1]
        if s != '':
            word, tag = s.split(' ')
            if word2id.get(word) is None:
                word2id[word] = len(word2id)
            sentence.append(word)
            if tag2id.get(tag) is None:
                print(tag)
            tags.append(tag2id[tag])
        elif len(sentence) != 0:
            sentences.append(sentence.copy())
```

```
            tag_list.append(tags.copy())
            sentence.clear()
            tags.clear()
    if len(sentence) != 0:
        sentences.append(sentence.copy())
        tag_list.append(tags.copy())
    return sort_labels, word2id, sentences, tag_list
```

2. 构建A,B,Pi矩阵，直接读入所有的数据，直接计算出发生和转移概率矩阵。最后为了不直接禁止没有出现过的发生和转移情况，给矩阵中所有为0的位置全部加上了一个很小的数字。最后为了防止viterbi的时候数字过大这里再取一个log方便运算。

```
def build(sort_labels, word2id, sentences, tag_list):
    A = np.zeros([len(sort_labels), len(sort_labels)])
    B = np.zeros([len(word2id), len(sort_labels)])
    Pi = np.zeros([len(sort_labels)])
    for tags in tag_list:
        Pi[tags[0]] += 1
        for idx in range(len(tags) - 1):
            A[tags[idx]][tags[idx + 1]] += 1
    for idx in range(len(sentences)):
        sentence = sentences[idx]
        tags = tag_list[idx]
        for idx2 in range(len(tags)):
            B[word2id[sentence[idx2]]][tags[idx2]] += 1
    A[A == 0] += 5e-2
    B[B == 0] += 5e-2
    Pi[Pi == 0] += 5e-2
    A = A / np.sum(A, axis=1, keepdims=True)
    B = B / np.sum(B, axis=0, keepdims=True)
    Pi = Pi / np.sum(Pi)
    A = np.log2(A)
    B = np.log2(B)
    Pi = np.log2(Pi)
    return A, B, Pi
```

3. 利用viterbi算法求解路径，就是一个简单的dp，但是尽量少用循环提升性能

```
def viterbi(word2id, A, B, Pi, sentence):
    sentence_len = len(sentence)
    tag_num = len(Pi)
    dp = np.zeros((tag_num, sentence_len))
    pre = np.zeros((tag_num, sentence_len))
    start = word2id.get(sentence[0])
    if start is None:
        now = np.ones(tag_num)
        now = now * np.log(1.0 / tag_num)
    else:
        now = B[start]
    dp[:, 0] = Pi + now
    pre[:, 0] = -1
    for idx in range(1, sentence_len):
        wordid = word2id.get(sentence[idx], None)
        if wordid is None:
```

```
            now = np.ones(tag_num)
            now = now * np.log(1.0 / tag_num)
        else:
            now = B[wordid]
        dp[:, idx] = [np.max(dp[:, idx - 1] + A[:, tag_id], 0)
                      for tag_id in range(tag_num)] + now
        pre[:, idx] = [np.argmax(dp[:, idx - 1] + A[:, tag_id], 0)
                       for tag_id in range(tag_num)]
    p = int(np.argmax(a=dp[:, sentence_len - 1], axis=0))
    path = [p]
    for idx in range(sentence_len - 1, 0, -1):
        p = int(pre[p, idx])
        path.append(p)
    path.reverse()
    return path
```

实验结果(Validation)截图

1. 英文数据集

	precision	recall	f1-score	support
B-PER	0.9614	0.6900	0.8034	1842
I-PER	0.9358	0.7689	0.8442	1307
B-ORG	0.8110	0.7554	0.7822	1341
I-ORG	0.8600	0.6791	0.7589	751
B-LOC	0.9187	0.8247	0.8692	1837
I-LOC	0.8710	0.7354	0.7975	257
B-MISC	0.9257	0.8113	0.8647	922
I-MISC	0.8014	0.6763	0.7335	346
micro avg	0.9002	0.7538	0.8205	8603
macro avg	0.8856	0.7426	0.8067	8603
weighted avg	0.9032	0.7538	0.8201	8603

2. 中文数据集

	precision	recall	f1-score	support
B-NAME	0.9434	0.9804	0.9615	102
M-NAME	0.8675	0.9600	0.9114	75
E-NAME	0.8679	0.9020	0.8846	102
S-NAME	0.8000	0.5000	0.6154	8
B-CONT	0.8462	1.0000	0.9167	33
M-CONT	0.8889	1.0000	0.9412	64
E-CONT	0.9167	1.0000	0.9565	33
S-CONT	0.0000	0.0000	0.0000	0
B-EDU	0.8655	0.9717	0.9156	106
M-EDU	0.8551	1.0000	0.9219	177
E-EDU	0.8814	0.9811	0.9286	106
S-EDU	0.0000	0.0000	0.0000	0
B-TITLE	0.8305	0.8607	0.8453	689
M-TITLE	0.8362	0.8871	0.8609	1479
E-TITLE	0.9383	0.9710	0.9544	689
S-TITLE	0.0000	0.0000	0.0000	0
B-ORG	0.8985	0.9157	0.9070	522
M-ORG	0.9222	0.9097	0.9159	3622
E-ORG	0.7936	0.8103	0.8019	522
S-ORG	0.0000	0.0000	0.0000	0
B-RACE	0.6667	1.0000	0.8000	14
M-RACE	0.0000	0.0000	0.0000	0
E-RACE	0.6667	1.0000	0.8000	14
S-RACE	0.0000	0.0000	0.0000	1
B-PRO	0.3000	0.6667	0.4138	18
M-PRO	0.2500	0.6061	0.3540	33
E-PRO	0.4250	0.9444	0.5862	18
S-PRO	0.0000	0.0000	0.0000	0
B-LOC	0.2222	1.0000	0.3636	2
M-LOC	0.1818	1.0000	0.3077	6
E-LOC	0.2500	1.0000	0.4000	2
S-LOC	0.0000	0.0000	0.0000	0
micro avg	0.8703	0.9054	0.8875	8437
macro avg	0.5286	0.6833	0.5708	8437
weighted avg	0.8805	0.9054	0.8914	8437

Part2 CRF

基本原理

1. CRF的核心思想是考虑观测序列和标签序列之间的条件概率分布，通过最大化条件概率来预测最可能的标签序列。
2. 通过对序列生成构造一些特征以及对应的特征函数进行概率分布的评估。可以考虑自身词汇以及很多的上下文关系以及和标签之间的关系来综合考虑。一般可以直接将特征函数设置为出现了该特征就是1否则为0，这样考虑较为简单。
3. 对产生的特征进行加权评估，这些权重是可以根据模型的训练动态调整的。

4. 一般loss函数是最大似然估计，

$$P(t \mid h, \bar{\alpha}) = \frac{e^{\sum_s \alpha_s \phi_s(h,t)}}{Z(h, \bar{\alpha})}$$

，公式图片是从论文中摘抄下来的，含义大致就是当前选择的最优路径的加权概率在左右合法路径的权值综合的占比，显然这个占比应该是越大越优秀的。

代码大致框架

使用了Sklearn框架编写。这个框架只需要传输给用户为每一个单词设计的若干特征即可。sklearn框架将会自己提取这些特征，并构建特征函数以及训练每一个特征函数所占据的比重。另外sklearn中会自己帮助训练转移矩阵以及发射矩阵。必须注意的是在构建特征的时候不应该出现和标签有关的任何信息。与标签有关的注意会在sklearn中全部内置，如果在构建的时候产生和标签有关的信息就会产生信息泄露问题。

构建的特征（除了PPT已经给出的）

1. 中文：
    - 当前词、前一个前两个和后一个后两个词是否是数字
2. 英文：
    - 当前词、前一个前两个和后一个后两个词是否是数字
    - 当前词、前一个前两个和后一个后两个词的小写版本
    - 当前词、前一个前两个和后一个后两个词是否是大写，是否首字母大写
    - 当前词、前一个前两个和后一个后两个词的前后缀

实验结果(Validation)截图

1. 英文：

	precision	recall	f1-score	support
B-PER	0.9136	0.8952	0.9043	1842
I-PER	0.9556	0.9549	0.9552	1307
B-ORG	0.8422	0.8240	0.8330	1341
I-ORG	0.8650	0.8535	0.8592	751
B-LOC	0.9186	0.9091	0.9138	1837
I-LOC	0.9383	0.8288	0.8802	257
B-MISC	0.8929	0.8406	0.8659	922
I-MISC	0.8610	0.7341	0.7925	346
micro avg	0.9024	0.8782	0.8901	8603
macro avg	0.8984	0.8550	0.8755	8603
weighted avg	0.9021	0.8782	0.8897	8603

2. 中文

	precision	recall	f1-score	support
B-NAME	0.9903	1.0000	0.9951	102
M-NAME	0.9615	1.0000	0.9804	75
E-NAME	0.9902	0.9902	0.9902	102
S-NAME	1.0000	1.0000	1.0000	8
B-CONT	1.0000	1.0000	1.0000	33
M-CONT	1.0000	1.0000	1.0000	64
E-CONT	1.0000	1.0000	1.0000	33
S-CONT	0.0000	0.0000	0.0000	0
B-EDU	0.9811	0.9811	0.9811	106
M-EDU	0.9888	1.0000	0.9944	177
E-EDU	0.9810	0.9717	0.9763	106
S-EDU	0.0000	0.0000	0.0000	0
B-TITLE	0.9239	0.9158	0.9198	689
M-TITLE	0.8922	0.9405	0.9157	1479
E-TITLE	0.9927	0.9855	0.9891	689
S-TITLE	0.0000	0.0000	0.0000	0
B-ORG	0.9493	0.9330	0.9411	522
M-ORG	0.9396	0.9542	0.9468	3622
E-ORG	0.8951	0.8831	0.8891	522
S-ORG	0.0000	0.0000	0.0000	0
B-RACE	1.0000	1.0000	1.0000	14
M-RACE	0.0000	0.0000	0.0000	0
E-RACE	1.0000	1.0000	1.0000	14
S-RACE	0.0000	0.0000	0.0000	1
B-PRO	0.8095	0.9444	0.8718	18
M-PRO	0.8205	0.9697	0.8889	33
E-PRO	0.9000	1.0000	0.9474	18
S-PRO	0.0000	0.0000	0.0000	0
B-LOC	0.5000	1.0000	0.6667	2
M-LOC	1.0000	1.0000	1.0000	6
E-LOC	0.6667	1.0000	0.8000	2
S-LOC	0.0000	0.0000	0.0000	0
micro avg	0.9353	0.9494	0.9423	8437
macro avg	0.6932	0.7334	0.7092	8437
weighted avg	0.9358	0.9494	0.9424	8437

Part3：BiLSTM+CRF

基本原理

1. CRF部分和上面的原理十分相似不再赘述
2. 上面的CRF种需要我们自行设计需要提取的特征，这里我们可以使用LSTM来帮助我们提取特征，直接通过LSTM计算出每一个词汇的隐状态是某个状态的权值分数。将这个信息传输给CRF部分，这一部分信息比较类似于HMM的发射概率。
3. 还是和CRF比较相似的利用真是路径的分数在所有合法路径中分数的占比作为评判指标（真实的Loss做了一些改动）最后预测的时候先通过LSTM得到概率之后用vetebi解码计算既可得到预测的隐状态序列

代码大致框架以及代码讲解

1. 网络搭建

```
def __init__(self, vocab_size, tag_to_ix, embedding_dim, hidden_dim):
    super(BiLSTM_CRF, self).__init__()
    self.embedding_dim = embedding_dim
    self.hidden_dim = hidden_dim
    self.vocab_size = vocab_size
    self.tag_to_ix = tag_to_ix
    self.tagset_size = len(tag_to_ix)
    self.device = 'cuda' if torch.cuda.is_available() else 'cpu'
    self.word_embeds = nn.Embedding(vocab_size, embedding_dim)
    self.lstm = nn.LSTM(embedding_dim, hidden_dim // 2,
                        num_layers=1, bidirectional=True, batch_first=True)
    self.hidden2tag = nn.Linear(hidden_dim, self.tagset_size)
    self.transitions = nn.Parameter(
        torch.randn(self.tagset_size, self.tagset_size))
    self.transitions.data[tag_to_ix[START_TAG], :] = -10000
    self.transitions.data[:, tag_to_ix[STOP_TAG]] = -10000
```

2. loss的计算

$$loss = -log(\frac{P_{real}}{\sum_{i=1}^{n} p_i}) = -(\sum_{i=1}^{len} x_{iy_i} + \sum_{i=1}^{len-1} t_{y_i y_{i+1}} - log(\sum_{i=1}^{n} e^{s_i}))$$

所以在计算的loss的时候先得到lstm的输出，随后调用函数_forward_alg来计算所有路径的分数exp总和的log，这里并不需要枚举路径的选择而是可以通过枚举每一个位置上的情况得到结果。而获取句子真是标签的分数较为简单，只需要遍历即可。

```
    def _get_lstm_features(self, sentence, sentence_len):
        embeds = self.word_embeds(sentence)
        embeds2 = nn.utils.rnn.pack_padded_sequence(embeds, sentence_len,
batch_first=True, enforce_sorted=False)
        lstm_out, self.hidden = self.lstm(embeds2)
        lstm_out, _ = nn.utils.rnn.pad_packed_sequence(lstm_out,
batch_first=True)
        lstm_feats = self.hidden2tag(lstm_out)
        return lstm_feats

    def _score_sentence(self, feats, sentence_len, tags):
        batch_size = feats.shape[0]
        score = torch.zeros(1, batch_size, device=self.device)
        tags = torch.cat([(torch.tensor([self.tag_to_ix[START_TAG]],
dtype=torch.long, device=self.device).repeat(batch_size, 1)), tags],
                         dim=1)
        for batch_id in range(batch_size):
            tag_next = tags[batch_id, 1:sentence_len[batch_id] + 1]
            tag_now = tags[batch_id, 0:sentence_len[batch_id]]
            score[0][batch_id] = torch.sum(self.transitions[tag_next,
tag_now])
            score[0][batch_id] += torch.sum(feats[batch_id,
range(sentence_len[batch_id]), tag_next])
```

```
            score[0][batch_id] += self.transitions[self.tag_to_ix[STOP_TAG],
tags[batch_id][sentence_len[batch_id]]]
        return score

    def _forward_alg(self, feats):
        batch_size = feats.shape[0]
        seq_len = feats.shape[1]
        init_alphas = torch.full((batch_size, self.tagset_size), -10000.,
device=self.device)
        init_alphas[:, self.tag_to_ix[START_TAG]] = 0.
        previous = init_alphas
        for iii in range(seq_len):
            obs = feats[:, iii, :]
            alphas_t = []
            for next_tag in range(self.tagset_size):
                emit_score = obs[:, next_tag].unsqueeze(1).repeat(1,
self.tagset_size)
                trans_score =
self.transitions[next_tag].unsqueeze(0).repeat(batch_size, 1)
                next_tag_var = previous + trans_score + emit_score
                alphas_t.append(torch.logsumexp(next_tag_var, dim=1).view(1,
-1))
            previous = torch.cat(alphas_t).t()
        terminal_var = previous +
self.transitions[self.tag_to_ix[STOP_TAG]].unsqueeze(0).repeat(batch_size,
1)
        scores = torch.logsumexp(terminal_var, dim=1).view(1, -1)
        return scores

    def neg_log_likelihood(self, data, tags):
        sentence, sentence_len = data
        feats = self._get_lstm_features(sentence, sentence_len)
        forward_score = self._forward_alg(feats)
        gold_score = self._score_sentence(feats, sentence_len, tags)
        return forward_score.mean() - gold_score.mean()
```

3. 自行搭建了runner用于训练和选取在validate集合上的最佳参数。evcaluate使用sklearn的内置函数获取f1_score进行比较即可

```
class Runner(object):
    def __init__(self, model, optimizer, loss_fn, tag_num, **kwargs):
        self.model = model
        self.optimizer = optimizer
        self.loss_fn = loss_fn
        self.tag_num = tag_num
        self.best_score = 0
        self.device = 'cuda' if torch.cuda.is_available() else 'cpu'
        self.model.to(self.device)

    def train(self, train_loader, dev_loader, num_epochs, log_steps,
eval_steps, eval_steps2, change_thread, save_path):
        self.model.train()
        num_training_steps = num_epochs * len(train_loader)
        global_step = 0
        self.best_score = self.evaluate(dev_loader)
```

```
        print('start')
        for epoch in range(num_epochs):
            total_loss = 0
            for data in train_loader:
                X, y = data
                X = (X[0].to(self.device), X[1].to(self.device))
                y = y.to(self.device)
                loss = self.loss_fn(X, y)
                total_loss += loss
                loss.backward()
                self.optimizer.step()
                self.optimizer.zero_grad()
                if global_step and (global_step % eval_steps == 0 or
global_step == (num_training_steps - 1)):
                    dev_score = self.evaluate(dev_loader)
                    if dev_score > change_thread:
                        eval_steps = eval_steps2
                    self.model.train()
                    if dev_score > self.best_score:
                        self.save_model(save_path)
                        print(
                            f"[Evaluate] best accuracy performence has been
updated: {self.best_score:.5f} --> {dev_score:.5f}")
                        self.best_score = dev_score
                global_step += 1
                if global_step and global_step % log_steps == 0:
                    print(
                        f"[Train] epoch: {epoch}/{num_epochs}, step:
{global_step}/{num_training_steps}")
        print("[Train] Training done!")

    @torch.no_grad()
    def evaluate(self, dev_loader):
        self.model.eval()
        my_tags = []
        real_tags = []
        for (sentence, sentence_len), dev_tags in dev_loader:
            sentence = sentence.to(self.device)
            sentence_len = sentence_len.to(self.device)
            now_tags = self.model((sentence, sentence_len))
            for tags in now_tags:
                my_tags += tags
            for tags, now_len in zip(dev_tags, sentence_len):
                real_tags += tags[:now_len].tolist()
        dev_score = metrics.f1_score(y_true=real_tags, y_pred=my_tags,
labels=range(2, self.tag_num), average='micro')
        return dev_score
```

实验结果截图(validation)

1. 英文

	precision	recall	f1-score	support
B-PER	0.9276	0.8350	0.8789	1842
I-PER	0.9575	0.8095	0.8773	1307
B-ORG	0.8864	0.7800	0.8298	1341
I-ORG	0.8911	0.7843	0.8343	751
B-LOC	0.9543	0.8405	0.8938	1837
I-LOC	0.9140	0.7860	0.8452	257
B-MISC	0.9313	0.7939	0.8571	922
I-MISC	0.9256	0.6474	0.7619	346
micro avg	0.9280	0.8059	0.8626	8603
macro avg	0.9235	0.7846	0.8473	8603
weighted avg	0.9281	0.8059	0.8622	8603

2. 中文

	precision	recall	f1-score	support
B-NAME	0.9806	0.9902	0.9854	102
M-NAME	0.9487	0.9867	0.9673	75
E-NAME	0.9709	0.9804	0.9756	102
S-NAME	1.0000	1.0000	1.0000	8
B-CONT	1.0000	0.9697	0.9846	33
M-CONT	1.0000	0.9688	0.9841	64
E-CONT	1.0000	0.9697	0.9846	33
S-CONT	0.0000	0.0000	0.0000	0
B-EDU	0.9722	0.9906	0.9813	106
M-EDU	0.9724	0.9944	0.9832	177
E-EDU	0.9720	0.9811	0.9765	106
S-EDU	0.0000	0.0000	0.0000	0
B-TITLE	0.9314	0.9260	0.9287	689
M-TITLE	0.9556	0.9175	0.9362	1479
E-TITLE	0.9956	0.9768	0.9861	689
S-TITLE	0.0000	0.0000	0.0000	0
B-ORG	0.9823	0.9559	0.9689	522
M-ORG	0.9689	0.9638	0.9664	3622
E-ORG	0.9214	0.8985	0.9098	522
S-ORG	0.0000	0.0000	0.0000	0
B-RACE	1.0000	1.0000	1.0000	14
M-RACE	0.0000	0.0000	0.0000	0
E-RACE	1.0000	1.0000	1.0000	14
S-RACE	0.0000	0.0000	0.0000	1
B-PRO	0.9000	1.0000	0.9474	18
M-PRO	0.9091	0.9091	0.9091	33
E-PRO	0.9474	1.0000	0.9730	18
S-PRO	0.0000	0.0000	0.0000	0
B-LOC	1.0000	1.0000	1.0000	2
M-LOC	1.0000	1.0000	1.0000	6
E-LOC	1.0000	1.0000	1.0000	2
S-LOC	0.0000	0.0000	0.0000	0
micro avg	0.9640	0.9512	0.9575	8437
macro avg	0.7290	0.7306	0.7296	8437
weighted avg	0.9638	0.9512	0.9574	8437