Project 2 实验报告

Part 1. 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认为有一个发射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。同时存在一个初始标签概率。
2. 如果已知了这些概率就可以使用viterbi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求出频率，近似认为概率。

代码基本框架

1. `dataprocess` 模块负责对中英文数据进行预处理的，包括整理保存为 `.npz` 格式、生成 `word2id` 、生成 `tag2id` 等。
2. `HMM` 模块是HMM模型的核心代码，其中的 `HMM_model` 类包含了初始状态概率矩阵、状态转移概率矩阵、发射概率矩阵，类的 `train` 方法用于学习参数，`viterbi` 方法用于解码。
3. `train.ipynb` 是训练相关的代码。
4. `test.ipynb` 是测试相关的代码。

模型设计

将原始 `train.txt` 文件中的一个个单词（汉字）和一个id对应起来，构建出 `vocab` 。将 `tag.txt` 中的tag提取出来，逐一和一个id对应，构建出 `tag2id` 。通过这两步操作将字符转为量化特征，便于输入模型。

我们对训练集数据进行直接统计，得到HMM的关键参数矩阵：`A`、`B`、`Pi`：

```
for sentence, labels in tqdm(train_data):
    for j in range(len(sentence)):
        cur_word = sentence[j]
        cur_tag = labels[j]
        self.B[self.tag2id[cur_tag]][self.vocab[cur_word]] += 1
        if j == 0:  # 统计初始概率
            self.Pi[self.tag2id[cur_tag]] += 1
            continue
        pre_tag = labels[j - 1]
        self.A[self.tag2id[pre_tag]][self.tag2id[cur_tag]] += 1
```

同时为了防止viterbi解码过程中概率不断相乘发生数据下溢，对参数取对数：

```
        # 对数据进行对数归一化，并最后取log防止连乘浮点数下溢
        self.Pi = self.Pi / self.Pi.sum()
        self.Pi[self.Pi == 0.0] = self.epsilon
        self.Pi = np.log10(self.Pi)

        row_sums = self.A.sum(axis=1)
        zero_rows = row_sums == 0
```

```
        self.A[~zero_rows] /= row_sums[~zero_rows, None]
        self.A[zero_rows] = 0
        self.A[self.A == 0] = self.epsilon
        self.A = np.log10(self.A)

        row_sums = self.B.sum(axis=1)
        zero_rows = row_sums == 0
        self.B[~zero_rows] /= row_sums[~zero_rows, None]
        self.B[zero_rows] = 0
        self.B[self.B == 0] = self.epsilon
        self.B = np.log10(self.B)
```

解码直接使用viterbi算法，只不过由于参数取了对数，viterbi中概率的相乘在这里变成相加。viterbi本质就是一个DP，最后回溯找到最优路径：

```
    def viterbi(self, O):
        N = len(self.Pi)
        T = len(O)
        delta = np.zeros((N, T))
        psi = np.zeros((N, T), dtype=int)

        delta[:, 0] = self.Pi + self.B[:, self.vocab.get(O[0], 0)]
        for t in range(1, T):
            O_t = self.vocab.get(O[t], 0)
            for j in range(N):
                # delta[j, t] = max(delta[i, t-1] * A[i, j] * B[j, O[t]]) 1<=i<=N
                # 参数已经取过log了，所以这里得用加法
                delta[j, t] = np.max(delta[:, t - 1] + self.A[:, j]) + self.B[j,
O_t]
                psi[j, t] = np.argmax(delta[:, t - 1] + self.A[:, j])

        best_path = np.zeros(T, dtype=int)
        best_path[-1] = np.argmax(delta[:, -1])
        for t in range(T - 2, -1, -1):
            best_path[t] = psi[best_path[t + 1], t + 1]

        return [self.idx2tag[id] for id in best_path]
```

实验结果

最后在验证集上进行验证，中文数据集上micro avg f1-score为0.8734，英文数据集上micro avg f1-score为0.8173。可以看到，中文数据集上的表现要优于英文数据集上的表现。

Part 2. 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. 一般使用最大似然作为loss函数：

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

以上公式来自参考论文，大致含义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。

代码基本框架

1. `dataprocess` 直接调用 Part 1. 的模块，功能相同。
2. `sklearn_crf` 模块中是CRF的核心代码。由于使用了 `sklearn_crfsuite` 框架，这里只需要实现库接口要求中的特征函数即可。`sent2features` 对每个单词（汉字）调用 `word2features` 函数，转化为量化的特征字典。
3. `train.ipynb` 是训练相关的代码。
4. `test.ipynb` 是测试相关的代码。

模型设计

使用了 `sklearn_crfsuite` 框架编写。这个框架只需要传入为每一个单词设计的若干特征即可，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权重。另外框架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框架中全部内置，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。

模型采用了 `sklearn_crfsuite.CRF` 进行训练，参数选择如下：

```
crf_model = CRF(
    algorithm="ap",
    max_iterations=300,
    all_possible_transitions=True,
    verbose=True,
)
```

其中 `algorithm="ap"` 参数指定了使用Averaged Perceptron方法进行学习，`max_iterations=300` 参数指定了最大迭代300次，`all_possible_transitions` （所有可能的转移）指定是否允许CRF模型学习所有可能的标签转移。设置为 `True` ，模型将学习所有可能的标签转移。`verbose` （详细程度）指定训练过程中的详细程度。设置为1，不断输出训练过程的表现，利于观察模型的能力。

此外还需要设计提取的特征：

```
def word2features_0(sent, language, i):
    word = sent[i]
    prev_word = "<start>" if i == 0 else sent[i - 1]  # START_TAG
    next_word = "<end>" if i == (len(sent) - 1) else sent[i + 1]  # STOP_TAG
    prev_word2 = "<start>" if i <= 1 else sent[i - 2]  # START_TAG
    next_word2 = "<end>" if i >= (len(sent) - 2) else sent[i + 2]  # STOP_TAG
    features = {
        "w": word,
        "w-1": prev_word,
        "w+1": next_word,
        "w-1:w": prev_word + word,
        "w:w+1": word + next_word,
        "w-1:w:w+1": prev_word + word + next_word,  # add
```

```
        "w-2:w": prev_word2 + word,  # add
        "w:w+2": word + next_word2,  # add
        "bias": 1,
        "word.isdigit": word.isdigit(),
    }
    return features

def en2features(sent, i):
    word = sent[i]
    features = {
        "bias": 1.0,
        "word.lower()": word.lower(),
        "word[-3:]": word[-3:],
        "word[-2:]": word[-2:],
        "word[:3]": word[:3],
        "word[:2]": word[:2],
        "word.isupper()": word.isupper(),
        "word.istitle()": word.istitle(),
        "word.isdigit()": word.isdigit(),
        "word": word,
        "word.length()": len(word),
        "word.isalnum()": word.isalnum(),
        "word.has_hyphen()": "-" in word,
        "word.has_digit()": any(char.isdigit() for char in word),
    }
    if i > 0:
        prev_word = sent[i - 1]
        features["prev_word.lower()"] = prev_word.lower()
        features["prev_word.isupper()"] = prev_word.isupper()
        features["prev_word[-3:]"] = prev_word[-3:]
        features["prev_word[-2:]"] = prev_word[-2:]
        features["prev_word[:3]"] = prev_word[:3]
        features["prev_word[:2]"] = prev_word[:2]
    else:
        features["BOS"] = True

    if i < len(sent) - 1:
        next_word = sent[i + 1]
        features["next_word.lower()"] = next_word.lower()
        features["next_word.isupper()"] = next_word.isupper()
        features["next_word[-3:]"] = next_word[-3:]
        features["next_word[-2:]"] = next_word[-2:]
        features["next_word[:3]"] = next_word[:3]
        features["next_word[:2]"] = next_word[:2]
    else:
        features["EOS"] = True

    return features
```

面对中文数据集，主要考察特征：当前字、前一个字、后一个字、汉字是否为数字，以及这些字的相加后的值等。面对英文数据集，主要考察特征：当前单词、前一个单词、后一个单词、单词是否为数字、大小写、单词的前后缀等。利用这两个函数，分别对中文和英文句子进行特征化处理后，再输入模型。

实验结果

最后在验证集上进行验证，中文数据集上micro avg f1-score为0.9457，英文数据集上micro avg f1-score为0.8921。可以看到，中文数据集上的表现要优于英文数据集上的表现。

Part 3. BiLSTM+CRF

基本原理

1. CRF部分的原理和 Part 2. 比较类似，这里不赘述。
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比较相似的利用真实路径的分数在所有合法路径中分数的占比作为评判指标（实际代码中的loss有些许不同）最后预测的时候先通过LSTM得到概率，然后再用viterbi解码计算既可得到预测的隐状态序列。

代码基本框架

1. `data_process` 模块负责对数据进行预处理，具体的函数和 Part 1. 中的 `dataprocess` 比较接近，不过在构建 `tag2id` 和 `word2id` 的时候加入了 `"UNK"` 表示未知单词（汉字），`"PAD"` 表示填充的占位字等。
2. `dataset` 模块中是自定义的数据集 `MyDataset` ，其读入原始文本数据，利用 `tag2id` 和 `word2id` 将其转化为量化张量，同时还定义了 `collate_fn` 作为自定义的批打包函数，完成填充句子等步骤。
3. `model` 模块是BiLSTM+CRF的核心代码。其中的 `CRF` 类主要包含一个可训练的状态转换矩阵。`BiLSTM_CRF` 类则包含一个 `nn.LSTM` 和 `CRF` ，前向传播时先通过 BiLSTM ，然后用 `CRF._viterbi_decode` 进行解码，得到标签序列。
4. `runner` 模块中定义了 `Runner` 类，能够对传入的模型进行训练、评估、保存加载等。
5. `test.ipynb` 是测试相关的代码。
6. `train.ipynb` 是训练相关的代码。

模型设计

在数据预处理上，对于 `word2id` 的构建，需要添加PAD与UNK，PAD用于在构建batch的时候填充句子，将句子填充为同样的长度。UNK是在识别到不在字典中出现的词时的标签：

```
def build_word2id(filename):
    word2id = {"PAD": 0, "UNK": 1}
    with open(filename, "r", encoding="utf-8") as file:
        lines = file.readlines()
    for line in lines:
        line = line.strip()
        if line:
            word, _ = line.split()
            if word not in word2id:
                word2id[word] = len(word2id)
    id2word = {v: k for k, v in word2id.items()}
    return word2id, id2word
```

类似地，由于使用LSTM，所以使用了START_TAG和STOP_TAG用于标志句子的起始与结束：

```
def build_tag2id(filename):
    tags = []
    with open(filename, "r", encoding="utf-8") as f:
        content = f.read()
        tags = re.findall(r"[B|M|E|S|I]-[A-Z]+", content)
    if filename == "../NER/Chinese/tag.txt":
        tags = tags[4:]
    else:
        tags = tags[2:]
    tag2id = {}
    tag2id["O"] = 0
    for tag in tags:
        tag2id[tag] = len(tag2id)
    tag2id["<START>"] = len(tag2id)
    tag2id["<STOP>"] = len(tag2id)
    id2tag = {v: k for k, v in tag2id.items()}
    return tag2id, id2tag
```

在构建数据集的时候，除了利用相关字典将句子转化成整数张量，还会在 `collate_fn` 进行长度填充：

```
    def collate_fn(self, batch):
        text = [t for t, _ in batch]
        label = [l for _, l in batch]
        seq_len = [len(i) for i in text]
        max_len = max(seq_len)

        text = [t + [self.word2id["PAD"]] * (max_len - len(t)) for t in text]
        label = [l + [self.tag2id["O"]] * (max_len - len(l)) for l in label]

        text = torch.tensor(text, dtype=torch.long)
        label = torch.tensor(label, dtype=torch.long)
        seq_len = torch.tensor(seq_len, dtype=torch.long)

        return text, label, seq_len
```

首先统计一个batch内部所有可能的seq的长度，并将最大的长度记录下来，同时将所有其他小于最大长度的sentence与label长度扩充到与最长的seq一致的长度。扩充的方法是对sentence用PAD进行扩充，对label用O进行扩充。扩充处理完成之后返回对应的text，label，seq_len（注：seq_len应为每个句子原本的长度，即扩充之前的长度，因为在之后对loss进行计算的时候，不应该将超出句子长度的部分即用padding填充的部分进行计算)。

网络的搭建利用了 `torch` 框架：

```
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  # 隐层维度
        self.vocab_size = len(vocab)  # 词表大小
        self.tagset_size = len(label_map)  # 标签个数
        self.device = device
```

```
        # 记录状态，'train'、'eval'、'pred'对应三种不同的操作
        self.state = "train"  # 'train'、'eval'、'pred'

        self.word_embeds = nn.Embedding(self.vocab_size, embedding_dim)
        self.lstm = nn.LSTM(
            embedding_dim,
            hidden_dim // 2,
            num_layers=2,
            bidirectional=True,
            batch_first=True,
        )

        self.hidden2tag = nn.Linear(hidden_dim, self.tagset_size, bias=True)
        self.crf = CRF(label_map, device)
        self.dropout = nn.Dropout(p=0.5, inplace=True)
        self.layer_norm = nn.LayerNorm(self.hidden_dim)
```

模型的网络结构主要由三个部分组成：词向量嵌入层、双向LSTM和CRF层。

- 在输入句子时，每个词语经过embedding层后被转化为一个大小为embedding_dim的向量。
- 接着，整个句子通过双向LSTM进行处理，每个时刻输出一个大小为hidden_dim的向量，包括正向和反向LSTM的输出。
- 最后，通过CRF层对标签序列建模，解决标签间的依存关系。

在训练过程中，该模型采用负对数似然函数计算交叉熵损失。同时在模型的训练和评估过程中，为了避免过拟合，该模型还采用了dropout和layer normalization等技术来提高模型的泛化能力和鲁棒性。

同时模型还提供了eval与pred的状态，eval状态会每次调用crf中的viterbi算法，将生成的最优预测序列加入到tag中并返回，pred状态类似。

```
    def forward(self, sentence, seq_len, tags=""):
        feats = self._get_lstm_features(sentence, seq_len)
        # 根据 state 判断哪种状态，从而选择计算损失还是维特比得到预测序列
        if self.state == "train":
            loss = self.crf.neg_log_likelihood(feats, tags, seq_len)
            return loss
        elif self.state == "eval":
            all_tag: list[list[int]] = []
            for i, feat in enumerate(feats):
                # path_score, best_path =
self.crf._viterbi_decode(feat[:seq_len[i]])
                all_tag.append(self.crf._viterbi_decode(feat[: seq_len[i]])[1])
            return all_tag
        elif self.state == "pred":
            return self.crf._viterbi_decode(feats[0])[1]
```

CRF部分通过给每个可能的标签分配一个得分，以便在标记序列中选择最佳标签序列。模型使用了前向传播算法来计算给定序列下每个标签的得分和后向传播算法来计算每个标签的后验得分，以便计算负对数似然损失。利用模型中的LSTM模块生成特征序列，并将序列传递给CRF模块进行训练和推断。CRF主要提供了三个部分的功能，分别是_forward_alg提供所有路径的得分，_score_sentence提供了标签路径的得分以及_viterbi_decode得到最优路径。

在train的过程中得到所有路径得分与标签路径得分后返回batch的分数平均值，evaluate的过程则直接调用viterbi解码部分获取到最优路径。

实验结果

最后在验证集上进行验证，中文数据集上micro avg f1-score为0.9556，英文数据集上micro avg f1-score为0.8780。可以看到，中文数据集上的表现要优于英文数据集上的表现。