PJ2实验报告

实验内容

分别使用HMM模型、CRF模型、Bilstm模型完成命名实体识别任务

HMM模型

使用HMM模型来完成此任务，将标签作为隐马尔可夫模型的状态，文字作为观测值。

模型的训练过程：

```
def initialize_parameters(tags):
    n_tags = len(tags)
    tag_to_index = {tag: i for i, tag in enumerate(tags)}

    transition_matrix = np.zeros((n_tags, n_tags))
    emission_matrix = {}
    initial_probabilities = np.zeros(n_tags)

    return transition_matrix, emission_matrix, initial_probabilities, tag_to_index

def train_hmm(file_path, tags):
    transition_matrix, emission_matrix, initial_probabilities, tag_to_index =
initialize_parameters(tags)

    with open(file_path, "r", encoding='utf-8') as f:
        lines = f.readlines()
    previous_tag = None;
    for line in lines:
        line = line.strip()

        if line:
            word, tag = line.split()
            tag_index = tag_to_index[tag]

            if word not in emission_matrix:
                emission_matrix[word] = np.zeros(len(tag_to_index))
            emission_matrix[word][tag_index] += 1

            if previous_tag is None:
                initial_probabilities[tag_index] += 1
            else:
                previous_tag_index = tag_to_index[previous_tag]
                transition_matrix[previous_tag_index][tag_index] += 1

            previous_tag = tag
        else:
            # Reset previous_tag to None for the next sentence
```

```
                previous_tag = None

    # Normalize initial probabilities
    initial_probabilities = initial_probabilities / np.sum(initial_probabilities)

    # Normalize transition matrix
    transition_matrix = np.divide(transition_matrix, np.sum(transition_matrix, axis=1,
keepdims=True), out=np.zeros_like(transition_matrix), where=np.sum(transition_matrix,
axis=1, keepdims=True)!=0)

    # Normalize emission matrix
    for word in emission_matrix:
        emission_matrix[word] = emission_matrix[word] / np.sum(emission_matrix[word])

    return transition_matrix, emission_matrix, initial_probabilities, tag_to_index
```

transition_matrix：转移概率矩阵，表示各个状态之间转移的概率。

emission_matrix：发射概率矩阵，使用字典表示，便于灵活处理单词和标签的组合，同时防止因为未在训练集中出现的单词在测试集中报错

initial_probabilities：为初始概率，表示各标签作为句子第一个标签的概率。

tag_index:用来对标签对索引进行映射。

训练过程：

- 当出现某个单词和标签的组合时，增加emission_matrix对应位置的计数。如果该单词是第一次出现，添加该单词和各个标签的映射关系的一元组。
- 对于标签转移的组合进行计数，作为transition_matrix的对应位置的值
- 当位于句首时，对initial_probabilities对应位置增加。 如果出现空行，则表示该句子已经结束，重置previous_tag为None。

归一化过程： 分别对initial_probabilities、transition_matrix、emission_matrix进行归一化，其中对于transition_matrix的归一化需要注意到全都为0的行，将其设为全为0.

解码过程

```
def decode_and_save_results(input_file, output_file, tags, transition_matrix,
emission_matrix, initial_probabilities):
    with open(input_file, "r", encoding='utf-8') as f:
        lines = f.readlines()

    observations = []
    original_lines = []
    results = []
    for line in lines:
        line = line.strip()
        if line:
            word, _ = line.split()
```

```
            observations.append(word)
            original_lines.append(line)
        else:
            if observations:
                predicted_tags = viterbi(observations, tags, transition_matrix,
emission_matrix, initial_probabilities)
                for line, tag in zip(original_lines, predicted_tags):
                    word, _ = line.split()
                    results.append(f"{word} {tag}\n")
                results.append("\n")
            observations = []
            original_lines = []

    if observations:##处理文件的最后一句
        predicted_tags = viterbi(observations, tags, transition_matrix, emission_matrix,
initial_probabilities)
        for line, tag in zip(original_lines, predicted_tags):
            word, _ = line.split()
            results.append(f"{word} {tag}\n")

    os.makedirs(os.path.dirname(output_file), exist_ok=True)
    with open(output_file, "w", encoding='utf-8') as f:
        f.writelines(results)

def viterbi(observation_sequence, tags, transition_matrix, emission_matrix,
initial_probabilities):
    sequence_length = len(observation_sequence)
    num_tags = len(tags)

    dp = np.zeros((num_tags, sequence_length))
    path = np.zeros((num_tags, sequence_length), dtype=int)

    for i in range(num_tags):
        dp[i][0] = initial_probabilities[i] * emission_matrix.get(observation_sequence[0],
np.ones(num_tags) * 1e-6)[i]

    for t in range(1, sequence_length):
        for i in range(num_tags):
            probability = dp[:, t-1] * transition_matrix[:, i] *
emission_matrix.get(observation_sequence[t], np.ones(num_tags) * 1e-6)[i]
            dp[i][t] = np.max(probability)
            path[i][t] = np.argmax(probability)

    tags_sequence = np.zeros(sequence_length, dtype=int)
    tags_sequence[sequence_length-1] = np.argmax(dp[:, sequence_length-1])

    for t in range(sequence_length-2, -1, -1):
        tags_sequence[t] = path[tags_sequence[t+1], t+1]

    return [tags[i] for i in tags_sequence]
```

decode_and_save_results函数主要对整个文件操作，其中分割成句子分别进行viterbi解码。

viterbi解码：

- 首先对于dp的第一列用initial_probabilities进行计算，如果在emission_matrix中找不到对应的单词，则将其置为接近0的值。
- 再对每一列进行循环操作，计算每一个状态下的概率值列表，dp[i][t]取其中最大的，同时更新path[i][t]。
- 最后找到最后一列中的最大值，向前回溯找到对应的tag即可。

CRF模型

```
import sklearn_crfsuite
from sklearn_crfsuite import metrics

# 读取数据函数
def read_data(file_path):
    sentences = []
    sentence = []
    with open(file_path, 'r', encoding='utf-8') as f:
        for line in f:
            line = line.strip()
            if not line:
                if sentence:
                    sentences.append(sentence)
                    sentence = []
            else:
                char, label = line.split()
                sentence.append((char, label))
        if sentence:
            sentences.append(sentence)
    return sentences

# 特征提取函数
def simple_features(sent):
    return [{'char': char} for char, _ in sent]

# 标签提取函数
def extract_labels(sent):
    return [label for _, label in sent]

# 训练模型函数
def train_crf(X_train, y_train):
    crf = sklearn_crfsuite.CRF(
        algorithm='lbfgs',
        c1=0.1,
        c2=0.1,
        max_iterations=100,
        all_possible_transitions=False
    )
    crf.fit(X_train, y_train)
    return crf
```

```
# 预测并保存结果函数
def predict_and_save_results(crf, test_sents, X_test, output_file):
    y_pred = crf.predict(X_test)
    with open(output_file, 'w', encoding='utf-8') as f:
        for sent, preds in zip(test_sents, y_pred):
            for (char, _), pred_label in zip(sent, preds):
                f.write(f"{char} {pred_label}\n")
            f.write("\n")

# 主函数
def main(language):
    if language == "English":
        train_file = "English/train.txt"
        validation_file = "English/validation.txt"
        output_file = "example_data/english_my_result.txt"
    else:
        train_file = "Chinese/train.txt"
        validation_file = "Chinese/validation.txt"
        output_file = "example_data/chinese_my_result.txt"

    # 读取训练和测试数据
    train_sents = read_data(train_file)
    test_sents = read_data(validation_file)

    # 提取特征和标签
    X_train = [simple_features(s) for s in train_sents]
    y_train = [extract_labels(s) for s in train_sents]
    X_test = [simple_features(s) for s in test_sents]
    y_test = [extract_labels(s) for s in test_sents]

    # 训练模型
    crf = train_crf(X_train, y_train)

    # 预测并保存结果
    predict_and_save_results(crf, test_sents, X_test, output_file)
    print(f"Prediction completed and results written to {output_file}")

if __name__ == "__main__":
    language = "English"
    main(language)
```

代码解释

1. **数据读取函数** `read_data`：
    - 读取训练和测试数据，并将每个句子存储为字符和标签的元组列表。
2. **简单特征提取函数** `simple_features`：
    - 使用一个简单的特征提取函数，只将字符本身作为特征。这里的特征是一个包含字符的字典。
3. **标签提取函数** `extract_labels`：

- 提取句子的标签列表。

4. **训练模型函数** `train_crf`:
   - 使用 `sklearn-crfsuite` 来训练 `CRF` 模型。
5. **预测并保存结果函数** `predict_and_save_results`:
   - 使用训练好的 `CRF` 模型进行预测，并将结果保存到文件中。
6. **主函数** `main`:
   - 根据指定的语言（中文或英文）选择对应的训练和验证数据文件。
   - 读取数据，提取特征和标签，训练模型，并保存预测结果。

代码只以字符本身作为特征进行提取，

训练过程中的计算

1. **特征提取**:
   - 对每个训练样本（句子）提取特征，生成特征向量。
2. **特征函数计算**:
   - 对于每个特征函数 $(f_k(Y, X))$，计算其在训练样本中的值。
3. **计算对数似然**:
   - 使用提取的特征和计算出的特征函数值来计算对数似然函数 $(L(\lambda))$。
4. **优化模型参数**:
   - 通过优化算法（如L-BFGS）最大化对数似然函数，更新特征权重 (λ_k)。

CRF模型的基本思想是：给定一个观察序列（如一个句子），通过最大化条件概率来预测相应的标签序列（如词性标签）。具体来说，CRF定义了给定观察序列的标签序列的条件概率分布。

1. **全局特征函数**： CRF通过全局特征函数来建模，这些特征函数可以依赖于整个输入序列，能够捕捉到更复杂的依赖关系和上下文信息。
2. **条件概率**： CRF直接建模标签序列的条件概率 $(P(Y|X))$，而不是联合概率 $(P(X, Y))$。而不需要对词语的独立性进行假设。
3. **对数线性模型**： CRF是一个对数线性模型，条件概率表示为： $[P(Y|X) = \frac{1}{Z(X)} \exp \left(\sum_{k} \lambda_k f_k(Y, X) \right)]$ 其中， $(f_k(Y, X))$ 是特征函数， (λ_k) 是特征权重， $(Z(X))$ 是归一化因子，确保概率的和为1。
4. **归一化因子**： 归一化因子 $(Z(X))$ 定义为： $[Z(X) = \sum_{Y} \exp \left(\sum_{k} \lambda_k f_k(Y, X) \right)]$ 它对所有可能的标签序列进行求和，用来将非标准化的概率值转化为标准的概率值。
5. **训练**： 训练CRF模型通常采用对数似然函数最大化的方法，需要使用梯度下降、拟牛顿法或其他优化算法来优化参数 (λ_k)。

6. **推断**： 给定训练好的CRF模型，预测新数据的标签序列时，需要使用动态规划算法（如前向-后向算法、维特比算法）来高效地计算最可能的标签序列。

Bilstm_CRF模型

1. **输入层**:
    - 输入是一个句子，表示为单词的索引序列。
2. **嵌入层（Embedding Layer）**：
    - 将单词索引转换为对应的词向量。这一步使用了一个嵌入矩阵nn.Embedding。
    - 输出是形状为（句子长度，嵌入维度）的张量。
3. **BiLSTM层（Bidirectional LSTM Layer）**：
    - 词向量序列输入到双向LSTM中，双向LSTM可以捕捉到每个单词的上下文信息。
    - 双向LSTM包含两个LSTM：一个正向（从前到后）和一个反向（从后到前）。
    - 输出是形状为（句子长度，隐藏层维度）的张量，隐藏层维度是正向和反向LSTM的输出拼接在一起的。
4. **全连接层（Linear Layer）**：
    - LSTM的输出经过一个线性层，将每个时间步的隐藏状态映射到标签空间。
    - 输出是形状为（句子长度，标签集大小）的张量，对应每个单词在每个标签上的得分。
5. **CRF层（CRF Layer）**：
    - 将线性层输出的得分输入到CRF层中，CRF层用来考虑标签序列的依赖关系，输出最可能的标签序列。
    - CRF层通过动态规划算法（如Viterbi算法）找到最优标签序列。

读取数据函数

```
def read_data(file_path):
    sentences = []
    sentence = []
    with open(file_path, "r", encoding="utf-8") as f:
        for line in f:
            line = line.strip()
            if not line:
                if sentence:
                    sentences.append(sentence)
                    sentence = []
            else:
                word, tag = line.split()
                sentence.append((word, tag))
        if sentence:
            sentences.append(sentence)
    return sentences
```

NER数据集类

```
class NERDataset(Dataset):
    def __init__(self, sentences, word2idx, tag2idx):
        self.sentences = sentences
        self.word2idx = word2idx
        self.tag2idx = tag2idx

    def __len__(self):
        return len(self.sentences)

    def __getitem__(self, idx):
        sentence = self.sentences[idx]
        words = [self.word2idx[word] if word in self.word2idx else self.word2idx["<UNK>"]
for word, tag in sentence]
        tags = [self.tag2idx[tag] for word, tag in sentence]
        return torch.tensor(words, dtype=torch.long), torch.tensor(tags, dtype=torch.long)
```

- `sentences`：句子的列表。
- `word2idx`：词汇表的索引映射。
- `tag2idx`：标签的索引映射。
- `__len__(self)`：返回数据集的大小。
- `__getitem__(self, idx)`：返回给定索引张量形式的句子及其标签。

填充和整理批次函数

```
def pad_collate_fn(batch):
    (xx, yy) = zip(*batch)
    x_lens = [len(x) for x in xx]
    xx_pad = torch.nn.utils.rnn.pad_sequence(xx, batch_first=True, padding_value=0)
    yy_pad = torch.nn.utils.rnn.pad_sequence(yy, batch_first=True, padding_value=-1)
    return xx_pad, yy_pad, x_lens
```

- `pad_collate_fn(batch)`：用于将不等长的序列填充到相同长度，并整理为批次。解压批次中的输入和标签。返回填充后的输入、标签和原始长度。

BiLSTM_CRF 类的定义

该类继承自 `torch.nn.Module`，并实现了一个基于BiLSTM和CRF（条件随机场）的NER模型。

1. 初始化方法（`__init__`）

```
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.word_embeds = nn.Embedding(vocab_size, embedding_dim)
    self.lstm = nn.LSTM(embedding_dim, hidden_dim // 2, num_layers=1, bidirectional=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>"], :] = -10000
    self.transitions.data[:, tag_to_ix["<STOP>"]] = -10000
```

- **参数：**
    - `vocab_size`：词汇表大小。
    - `tag_to_ix`：标签到索引的映射字典。
    - `embedding_dim`：词嵌入维度。
    - `hidden_dim`：LSTM隐藏层维度。
- **成员变量：**
    - `word_embeds`：词嵌入层。
    - `lstm`：双向LSTM层。
    - `hidden2tag`：线性层，用于将LSTM输出映射到标签空间。
    - `transitions`：CRF的转移矩阵。初始化转移矩阵的所有元素为随机数，这些随机数来自标准正态分布。同时初始化标签和标签的转移概率，防止发生非法转移。

2. 负对数似然损失（`neg_log_likelihood`）

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

计算负对数似然损失，用于训练模型。

- 接收两个参数：
    - `sentence`：输入句子，通常表示为单词索引的序列。
    - `tags`：句子中每个单词对应的真实标签。
- 调用 `_get_lstm_features` 方法，获取输入句子的 LSTM 特征表示 `feats`。
- `feats` 是一个张量，包含每个时间步的特征，大小为 `(句子长度，标签集大小)`。
- 调用 `_forward_alg` 方法，使用前向算法计算所有可能标签序列的总得分。
- `forward_score` 是一个标量，表示所有可能路径的归一化因子。

- `gold_score` 是一个标量，表示模型为真实路径分配的得分。调用 `_score_sentence` 方法，计算真实标签路径的得分。 返回前向分数与真实路径分数之间的差值，即负对数似然损失。

3. 获取LSTM特征（`_get_lstm_features`）

```
def _get_lstm_features(self, sentence):
    self.hidden = self.init_hidden()
    embeds = self.word_embeds(sentence).view(len(sentence), 1, -1)
    lstm_out, self.hidden = self.lstm(embeds, self.hidden)
    lstm_out = lstm_out.view(len(sentence), self.hidden_dim)
    lstm_feats = self.hidden2tag(lstm_out)
    return lstm_feats
```

获取LSTM输出的特征，用于后续的CRF计算。

`_get_lstm_features` 方法首先通过嵌入层将句子转换为嵌入向量，然后通过 LSTM 层获取每个时间步的隐藏状态，再通过线性层将隐藏状态转换为每个时间步的标签得分。

- `self.hidden = self.init_hidden()` ：`init_hidden` 方法返回两个形状为 `(2, 1, self.hidden_dim // 2)` 的张量，分别表示双向 LSTM 的前向和后向隐藏状态。
- `self.word_embeds(sentence)` 将输入的句子转换为嵌入表示。`sentence` 是一个包含单词索引的张量，通过嵌入层 `word_embeds` 转换为对应的嵌入向量。`.view(len(sentence), 1, -1)` 将嵌入向量的形状调整为 `(seq_len, batch_size, embedding_dim)`，其中 `seq_len` 是句子长度，`batch_size` 为 1，`embedding_dim` 是嵌入维度。这是为了符合 LSTM 层的输入要求。
- `self.lstm(embeds, self.hidden)` 将嵌入向量输入到 LSTM 层，`self.hidden` 是初始隐藏状态。LSTM 返回 `lstm_out` 和更新后的隐藏状态 `self.hidden`。`lstm_out` 是 LSTM 的输出，其形状为 `(seq_len, batch_size, hidden_dim)`，即每个时间步的隐藏状态。`self.hidden` 是更新后的隐藏状态，用于下一次调用 LSTM 时初始化。
- `.view(len(sentence), self.hidden_dim)` 将 `lstm_out` 形状调整为 `(seq_len, hidden_dim)`，去掉批量大小的维度，使得每个时间步对应一个隐藏状态向量。
- `self.hidden2tag(lstm_out)` 将调整后的 LSTM 输出通过一个线性层 `hidden2tag`，该层将隐藏状态向量转换为每个时间步上所有标签的得分。`lstm_feats` 的形状为 `(seq_len, tagset_size)`，即每个时间步对应一个标签得分向量。
- 返回 `lstm_feats`，即通过 LSTM 和线性层处理后的特征表示，用于后续的 CRF 层处理。

4. `_forward_alg`

```
def _forward_alg(self, feats):
    init_alphas = torch.full((1, self.tagset_size), -10000.)
    init_alphas[0][self.tag_to_ix["<START>"]] = 0.

    forward_var = init_alphas
    for feat in feats:
        alphas_t = []
        for next_tag in range(self.tagset_size):
            emit_score = feat[next_tag].view(1, -1).expand(1, self.tagset_size)
            trans_score = self.transitions[next_tag].view(1, -1)
            next_tag_var = forward_var + trans_score + emit_score
```

```
            alphas_t.append(torch.logsumexp(next_tag_var, dim=1).view(1))
        forward_var = torch.cat(alphas_t).view(1, -1)
    terminal_var = forward_var + self.transitions[self.tag_to_ix["<STOP>"]]
    alpha = torch.logsumexp(terminal_var, dim=1)
    return alpha
```

方法实现了前向算法，用于处理序列标注问题。具体来说，它计算从起始状态到每个可能的标签序列的累积概率。

- `init_alphas` 初始化为一个大小为 `(1, self.tagset_size)` 的张量，初始值为 `-10000`。这样做是为了在 log-space 下避免数值上溢和下溢问题。
- `init_alphas[0][self.tag_to_ix["<START>"]] = 0.` 将起始标签 `<START>` 的初始分数设为 `0`，因为在起始位置，只有从 `<START>` 标签开始的路径是有效的。
- `forward_var` 变量保存到当前时间步 `t` 的所有可能路径的累积分数。初始化为 `init_alphas`。 遍历每个时间步的特征 `feat`。`feats` 是从 LSTM 获取的特征表示。
- `alphas_t` 存储每个标签 `next_tag` 的前向变量值。
- `emit_score` 是当前时间步 `feat` 中 `next_tag` 标签的发射分数。它扩展到 `(1, self.tagset_size)` 的大小。
- `trans_score` 是从所有标签到 `next_tag` 的转移分数。
- `next_tag_var` 是从所有前一个标签转移到 `next_tag` 的路径分数。`forward_var` 是前一个时间步的前向变量，加上 `trans_score` 和 `emit_score`。
- `torch.logsumexp(next_tag_var, dim=1)` 计算 `next_tag_var` 中所有路径分数的 log-sum-exp（在 log-space 下的总和）。
- `alphas_t.append(torch.logsumexp(next_tag_var, dim=1).view(1))` 将计算的前向变量值添加到 `alphas_t` 列表中。
- `forward_var = torch.cat(alphas_t).view(1, -1)` 更新 `forward_var`，以便在下一个时间步使用。
- `terminal_var` 是所有路径的累积分数，加上从最后一个标签到 `<STOP>` 标签的转移分数。
- `alpha = torch.logsumexp(terminal_var, dim=1)` 计算所有路径到达 `<STOP>` 标签的总和（在 log-space 下）。
- 返回最终计算的 `alpha`，即序列的分区函数值。这个值表示所有可能路径的总分数。

通过逐步计算每个时间步的前向变量，累积路径分数，最终得到所有可能标签序列的总分数。在训练过程中，这个分数用于计算模型的损失函数，帮助模型学会在给定特征表示的情况下，生成正确的标签序列。

5. 计算路径得分（`_score_sentence`）

```
def _score_sentence(self, feats, tags):
    score = torch.zeros(1)
    tags = torch.cat([torch.tensor([self.tag_to_ix["<START>"]], dtype=torch.long), tags])
    for i, feat in enumerate(feats):
        score = score + self.transitions[tags[i + 1], tags[i]] + feat[tags[i + 1]]
    score = score + self.transitions[self.tag_to_ix["<STOP>"], tags[-1]]
    return score
```

`_score_sentence` 方法用于计算给定标签序列的得分。得分由标签之间的转移分数和每个标签在对应时间步上的发射分数组成。具体步骤如下：

- `score = torch.zeros(1)` 初始化得分为0，开始累加各部分的得分。
- `torch.cat([torch.tensor([self.tag_to_ix["<START>"]], dtype=torch.long), tags])` 在标签序列的开头添加一个起始标签 `<START>`。这样可以考虑从起始标签到第一个实际标签的转移。
- `for i, feat in enumerate(feats):` 遍历每个时间步的特征向量 `feats`。
- `self.transitions[tags[i + 1], tags[i]]` 获取从标签 `tags[i]` 到标签 `tags[i + 1]` 的转移分数。
- `feat[tags[i + 1]]` 获取在时间步 `i` 上标签 `tags[i + 1]` 的发射分数。
- `score = score + self.transitions[tags[i + 1], tags[i]] + feat[tags[i + 1]]` 将当前时间步的转移分数和发射分数加到总得分中。
- `self.transitions[self.tag_to_ix["<STOP>"], tags[-1]]` 获取从最后一个实际标签到结束标签 `<STOP>` 的转移分数。
- `score = score + self.transitions[self.tag_to_ix["<STOP>"], tags[-1]]` 将结束标签的转移分数加到总得分中。
- 返回最终计算的总得分。
- `_score_sentence` 方法计算给定标签序列的总得分，该得分包括标签之间的转移分数和每个标签在对应时间步上的发射分数。
- 具体来说，方法首先初始化得分为0，然后在标签序列的开头添加一个起始标签。接着，遍历每个时间步，累加每个标签的转移分数和发射分数。最后，累加从最后一个实际标签到结束标签的转移分数，返回总得分。

6. viterbi解码

```
def _viterbi_decode(self, feats):
    backpointers = []

    # 初始化前向变量
    init_vvars = torch.full((1, self.tagset_size), -10000.)
    init_vvars[0][self.tag_to_ix["<START>"]] = 0

    forward_var = init_vvars
    for feat in feats:
        bptrs_t = []
        viterbivars_t = []
        for next_tag in range(self.tagset_size):
            # 计算从当前所有标签转移到next_tag的得分
            next_tag_var = forward_var + self.transitions[next_tag]
            # 找到到达next_tag的最佳前一个标签
            best_tag_id = torch.argmax(next_tag_var)
            # 将best_tag_id存入bptrs_t
            bptrs_t.append(best_tag_id)
            # 将到达next_tag的最佳得分存入viterbivars_t
            viterbivars_t.append(next_tag_var[0][best_tag_id].view(1))
```

```
            # 更新forward_var，包含当前时间步的发射分数feat
            forward_var = (torch.cat(viterbivars_t) + feat).view(1, -1)
            # 将当前时间步的回溯指针bptrs_t加入backpointers
            backpointers.append(bptrs_t)

        # 计算从最后一个标签到结束标签<STOP>的得分
        terminal_var = forward_var + self.transitions[self.tag_to_ix["<STOP>"]]
        best_tag_id = torch.argmax(terminal_var)
        # 最佳路径的总得分
        path_score = terminal_var[0][best_tag_id]

        best_path = [best_tag_id]
        # 反向遍历backpointers，找到最佳路径上的标签
        for bptrs_t in reversed(backpointers):
            best_tag_id = bptrs_t[best_tag_id]
            best_path.append(best_tag_id)
        start = best_path.pop()
        assert start == self.tag_to_ix["<START>"]
        best_path.reverse()
        return path_score, best_path
```

初始化：

- `backpointers`：用于存储每一步的回溯指针。
- `init_vvars`：前向变量的初始值，大小为`(1, tagset_size)`，值为`-10000`，表示极小的概率。
- `init_vvars[0][self.tag_to_ix["<START>"]] = 0`：设置起始状态的得分为`0`。

动态规划计算最优路径和回溯指针：

- `forward_var = init_vvars`：初始化前向变量。
- 对于`feats`中的每个`feat`：
  - 初始化`bptrs_t`和`viterbivars_t`为当前时间步的回溯指针和最佳得分列表。
  - 对于每个`next_tag`：
    - 计算从当前所有标签转移到`next_tag`的得分：`next_tag_var = forward_var + self.transitions[next_tag]`。
    - 找到最佳前一个标签：`best_tag_id = torch.argmax(next_tag_var)`。
    - 将`best_tag_id`存入`bptrs_t`。
    - 将到达`next_tag`的最佳得分存入`viterbivars_t`。
  - 更新`forward_var`，包含当前时间步的发射分数`feat`：`forward_var = (torch.cat(viterbivars_t) + feat).view(1, -1)`。
  - 将当前时间步的回溯指针`bptrs_t`加入`backpointers`。

计算终止状态的得分并回溯找到最佳路径：

- 计算从最后一个标签到结束标签`<STOP>`的得分：`terminal_var = forward_var + self.transitions[self.tag_to_ix["<STOP>"]]`。
- 找到到达结束状态得分最高的前一个标签：`best_tag_id = torch.argmax(terminal_var)`。
- 获取最佳路径的总得分：`path_score = terminal_var[0][best_tag_id]`。
- 初始化最佳路径，包含结束标签的最佳前一个标签。
- 反向遍历`backpointers`，找到最佳路径上的标签。

- 移除起始标签<START>。
- 反转最佳路径，使其按正确顺序排列。
- 返回最佳路径的得分path_score和最佳路径best_path。

7. train_and_predict 函数

作用是读取训练和验证数据，训练 BiLSTM-CRF 模型，并对验证数据进行预测。

创建词汇表和标签字典

```
    word2idx = {"<PAD>": 0, "<UNK>": 1}
    tag2idx = {"<START>": 0, "<STOP>": 1}
    for sentence in train_sents + validation_sents:
        for word, tag in sentence:
            if (word, tag) not in word2idx:
                word2idx[word] = len(word2idx)
            if tag not in tag2idx:
                tag2idx[tag] = len(tag2idx)
```

- 初始化词汇表 word2idx 和标签字典 tag2idx。
- 遍历所有训练和验证句子，将每个单词和标签添加到相应的字典中。

创建数据集和数据加载器

```
    train_dataset = NERDataset(train_sents, word2idx, tag2idx)
    validation_dataset = NERDataset(validation_sents, word2idx, tag2idx)
    train_loader = DataLoader(train_dataset, batch_size=batch_size,
collate_fn=pad_collate_fn, shuffle=True)
    validation_loader = DataLoader(validation_dataset, batch_size=batch_size,
collate_fn=pad_collate_fn)
```

- 创建训练和验证数据集。
- 使用 DataLoader 创建数据加载器，以便在训练过程中按批次加载数据。collate_fn 用于对数据进行填充。

定义模型和优化器

```
    model = BiLSTM_CRF(len(word2idx), tag2idx, embedding_dim=embedding_dim,
hidden_dim=hidden_dim)
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)
```

- 实例化 BiLSTM-CRF 模型。
- 使用 Adam 优化器进行模型参数的更新。

9. 训练与预测

```
    for epoch in range(num_epochs):
        model.train()
        for sentences, tags, lengths in train_loader:
            model.zero_grad()
            sentences = sentences.squeeze(0)
            tags = tags.squeeze(0)
            loss = model.neg_log_likelihood(sentences, tags)
            loss.backward()
            optimizer.step()
        print(f"Epoch {epoch + 1}/{num_epochs} completed")

    model.eval()
    predictions = []
    with torch.no_grad():
        for sentences, tags, lengths in validation_loader:
            sentences = sentences.squeeze(0)
            score, predicted_tags = model(sentences)
            predictions.append(predicted_tags)
```

结果验证

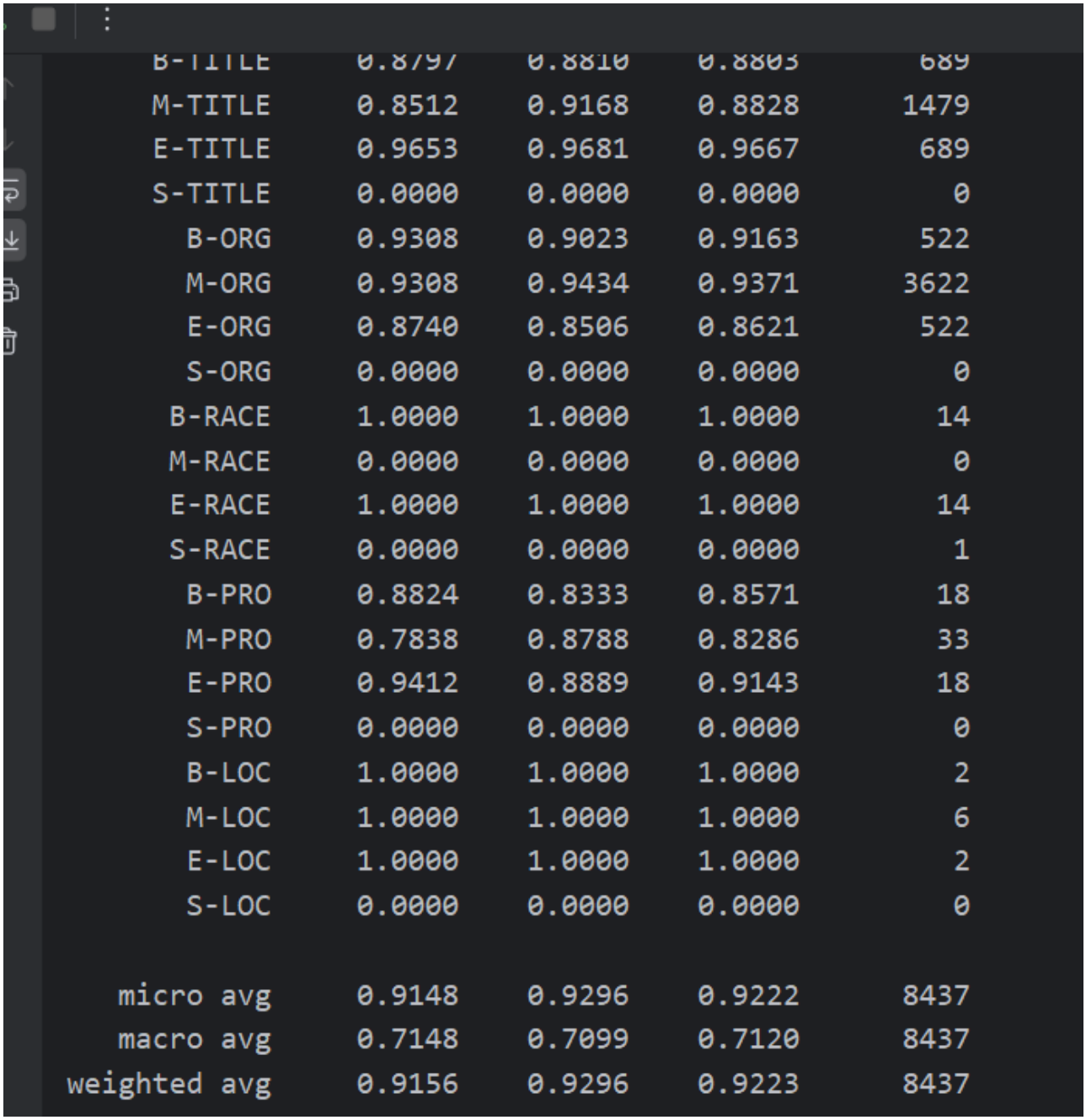
1. HMM模型 Chinese:

```
              precision    recall  f1-score   support

       M-EDU     0.9605    0.8249    0.8875       177
       E-EDU     0.9529    0.7642    0.8482       106
       S-EDU     0.0000    0.0000    0.0000         0
     B-TITLE     0.8779    0.8345    0.8557       689
     M-TITLE     0.9335    0.7877    0.8544      1479
     E-TITLE     0.9588    0.9129    0.9353       689
     S-TITLE     0.0000    0.0000    0.0000         0
       B-ORG     0.8602    0.8487    0.8544       522
       M-ORG     0.8454    0.9555    0.8971      3622
       E-ORG     0.8012    0.7797    0.7903       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     1.0000    1.0000    1.0000         1
       B-PRO     1.0000    0.1111    0.2000        18
       M-PRO     1.0000    0.0606    0.1143        33
       E-PRO     1.0000    0.1111    0.2000        18
       S-PRO     0.0000    0.0000    0.0000         0
       B-LOC     0.0000    0.0000    0.0000         2
       M-LOC     0.0000    0.0000    0.0000         6
       E-LOC     0.0000    0.0000    0.0000         2
       S-LOC     0.0000    0.0000    0.0000         0

   micro avg     0.8759    0.8732    0.8745      8437
   macro avg     0.6311    0.5068    0.5315      8437
weighted avg     0.8799    0.8732    0.8702      8437
```

English:

	precision	recall	f1-score	support
B-PER	0.9591	0.6873	0.8008	1842
I-PER	0.8796	0.5310	0.6622	1307
B-ORG	0.8099	0.7181	0.7613	1341
I-ORG	0.8673	0.5047	0.6380	751
B-LOC	0.8996	0.8242	0.8602	1837
I-LOC	0.8788	0.6770	0.7648	257
B-MISC	0.9301	0.7646	0.8393	922
I-MISC	0.9351	0.4162	0.5760	346
micro avg	0.8945	0.6787	0.7718	8603
macro avg	0.8949	0.6404	0.7378	8603
weighted avg	0.8966	0.6787	0.7661	8603

2. CRF模型 Chinese:

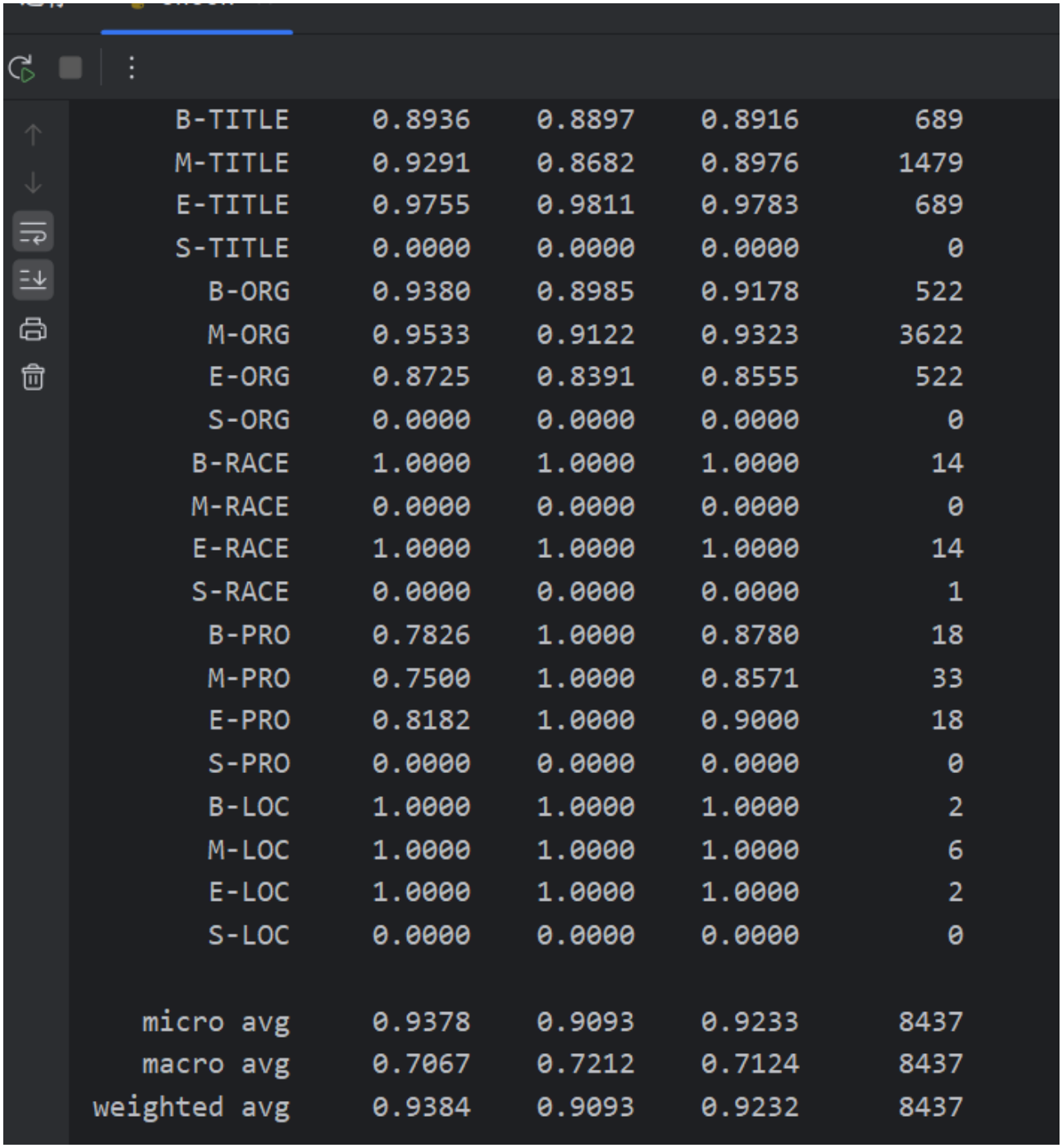


B-TITLE	0.8797	0.8810	0.8803	689
M-TITLE	0.8512	0.9168	0.8828	1479
E-TITLE	0.9653	0.9681	0.9667	689
S-TITLE	0.0000	0.0000	0.0000	0
B-ORG	0.9308	0.9023	0.9163	522
M-ORG	0.9308	0.9434	0.9371	3622
E-ORG	0.8740	0.8506	0.8621	522
S-ORG	0.0000	0.0000	0.0000	0
B-RACE	1.0000	1.0000	1.0000	14
M-RACE	0.0000	0.0000	0.0000	0
E-RACE	1.0000	1.0000	1.0000	14
S-RACE	0.0000	0.0000	0.0000	1
B-PRO	0.8824	0.8333	0.8571	18
M-PRO	0.7838	0.8788	0.8286	33
E-PRO	0.9412	0.8889	0.9143	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	0.9148	0.9296	0.9222	8437
macro avg	0.7148	0.7099	0.7120	8437
weighted avg	0.9156	0.9296	0.9223	8437

English:

```
E:\360MoveData\Users\DELL\Desktop\HMM\.venv\Scripts\python.exe E:\360MoveDat
              precision    recall  f1-score   support

       B-PER     0.9115    0.6265    0.7426      1842
       I-PER     0.8724    0.7376    0.7993      1307
       B-ORG     0.8652    0.6607    0.7493      1341
       I-ORG     0.7821    0.6644    0.7185       751
       B-LOC     0.9319    0.7899    0.8550      1837
       I-LOC     0.6077    0.7354    0.6655       257
      B-MISC     0.9365    0.7842    0.8536       922
      I-MISC     0.9069    0.6474    0.7555       346

   micro avg     0.8801    0.7079    0.7846      8603
   macro avg     0.8518    0.7058    0.7674      8603
weighted avg     0.8848    0.7079    0.7843      8603
```

3. BiLSTM_CRF模型 Chinese:



B-TITLE	0.8936	0.8897	0.8916	689
M-TITLE	0.9291	0.8682	0.8976	1479
E-TITLE	0.9755	0.9811	0.9783	689
S-TITLE	0.0000	0.0000	0.0000	0
B-ORG	0.9380	0.8985	0.9178	522
M-ORG	0.9533	0.9122	0.9323	3622
E-ORG	0.8725	0.8391	0.8555	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	0.7826	1.0000	0.8780	18
M-PRO	0.7500	1.0000	0.8571	33
E-PRO	0.8182	1.0000	0.9000	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	0.9378	0.9093	0.9233	8437
macro avg	0.7067	0.7212	0.7124	8437
weighted avg	0.9384	0.9093	0.9232	8437

English:

```
E:\360MoveData\Users\DELL\Desktop\HMM\.venv\Scripts\python.exe E:\360M
              precision    recall  f1-score   support

       B-PER     0.8388    0.6694    0.7446      1842
       I-PER     0.8427    0.7582    0.7982      1307
       B-ORG     0.6496    0.6801    0.6645      1341
       I-ORG     0.5317    0.6924    0.6015       751
       B-LOC     0.8657    0.7403    0.7981      1837
       I-LOC     0.7130    0.6187    0.6625       257
      B-MISC     0.8158    0.7061    0.7570       922
      I-MISC     0.7188    0.5983    0.6530       346

   micro avg     0.7629    0.7013    0.7308      8603
   macro avg     0.7470    0.6829    0.7099      8603
weighted avg     0.7778    0.7013    0.7344      8603


进程已结束，退出代码为 0
```