Hidden Markov Model 을 이용한 Part-of-Speech Tagging

학번 및 이름: 202150155 인지컴퓨팅 연구실 석사 과정 박은환

Email: judepark@jbnu.ac.kr

GitHub: https://github.com/JudePark96/hidden-markov-model-pos-tagging.git

1. 학습 실행 방법 소개

1.1. 파이썬 환경 구축 (선택)

```
pip3 install -r requirements.txt
```

> Python 3.6.8 환경에서 코드를 작성하였습니다.

`virtualvenv` 를 사용한다는 가정 하에 환경 구축을 진행합니다. 압축 폴더 내의 requirements.txt 를 이용하여 필요한 dependency 를 설치합니다.

1.2. 코드 받기 (선택)

```
git clone https://github.com/JudePark96/hidden-markov-model-pos-tagging.git
```

혹시 다운로드하신 코드에 문제가 존재한다면 위의 명령어를 통하여 코드를 받아올 수 있습니다.

1.3. CoNLL-U 전처리

```
cd dataset
python3 preprocessing_conllu.py
```

CoNLL-U 포맷의 데이터를 프로젝트에 알맞는 형태로 전처리 과정을 거칠 필요가 있습니다. 전처리의 경우 위의 명령어를 실행하면 되며 `../rsc/preprocessed_dataset/` 에 `train.pkl` 와 `dev.pkl` 가 저장됩니다. CoNLL-U의 `form` 과 `upos` 를 데이터로서 사용하였습니다.

1.4. Vocab 구축

`cd vocab` 을 통하여 vocab 폴더로 들어갑니다.

```
python3 vocab_util.py
```

위의 명령어를 통하여 Vocabulary 객체를 구현할 수 있으며 이는 `../rsc/vocab/vocab.pkl` 로 저장됩니다. 물론 압축 파일 내의 프로젝트 폴더에 미리 만들어두었기 때문에 위의 명령어는 실행하지 않아도 됩니다.

1.5. Probability Matrix 구축

Hidden Markov Model(HMM) 및 Viterbi Algorithm 을 사용하기 위해서는 States (POS 태그)와 Initial Hidden States, Transition Probability, Emission Probability 가 필요로 합니다. States 의 경우 Vocab 을 구축하며 만들어졌기 때문에 나머지를 구축해야합니다. 이는 `./dataset/hmm_prob.py` 에 기술해 놓았습니다.

```
cd dataset
python3 hmm_prob.py
```

위의 명령어를 통하여 Initial Hidden States, Transition Probability, Emission Probability 를 구축할 수 있으며 **../rsc/probs/{{train, dev}*initial, emission, transition*}prob.pkl** 로 저장됩니다.

1.6. 결과 출력

```
python3 main.py
```

프로젝트의 루트 폴더에서 위의 명령어를 실행하였을 때 학습 그리고 검증용 데이터 집합에 대한 결과가 나옵니다. 평가 지표의 경우 F1 Score 를 사용하였으며 **학습용 데이터 집합에 대하여 88%, 검증용 데이터 집합에 대하여 83%**를 거두었습니다. 또한 실행 창에서 각 태그 별 F1 Score 를 볼 수 있는 Report 를 출력해줍니다.

1.6. 추론

```
python3 inference.py
```

위의 명령어를 실행하고 입력으로 예를 들어 `I have a dog .` 와 같은 문장을 넣으면 되며 whitespace 단위로 나누어서 입력해야합니다.

1.6. Streamlit 을 이용한 추론

```
streamlit run web_inference.py
```

위의 명령어로 실행한 후 URL을 타고 들어가 위와 같이 실행하면 결과가 나옵니다.

2. Implementation Details

```
# ./dataset/preprocessing_conllu.py

def preprocess_conllu(path: str) -> List[List[Tuple[Any, Any]]]:
    dataset = read_conllu(path)
    output = []
    for data in tqdm(dataset):
        output.append([(j['form'], j['upos']) for j in data])

    return output
```

HMM w/ Viterbi Algorithm 을 위해서 우선 CoNLL-U 의 포맷을 사용할 수 있는 포맷으로 전처리해야합니다. 그렇기 때문에, 위의 함수를 정의하고 실행하여 전처리를 진행합니다. `List[List[Tuple[Any, Any]]]` 포맷으로 작성되며 이는 예를 들어 `[[(I, PRON), (have, VERB), ...], ...]` 식으로 저장됩니다.

```
# ./vocab/vocab_util.py

class VocabUtil(object):
    def __init__(self,
                 words: List[str],
                 tags: List[str]) -> None:
        super(VocabUtil, self).__init__()
        self.words = words
        self.pos_tags = tags

        self.vocab = []
        self.tags = []

    def build_vocab(self):
        words, tags = set(self.words), set(self.pos_tags)

        for word in words:
            self.vocab.append(word)

        for tag in tags:
            self.tags.append(tag)

    def convert_words_to_ids(self, sequence: List[str]) -> List[int]:
        if not isinstance(sequence, list):
            raise ValueError('Sequence has to be List[str].')

        return [0 if token not in self.vocab else self.vocab.index(token) for
token in sequence]

    def convert_ids_to_words(self, sequence: List[int]) -> List[str]:
        if not isinstance(sequence, list):
            raise ValueError('Sequence has to be List[str].')

        return [0 if token_id > len(self.vocab) else self.vocab[token_id] for
token_id in sequence]

    def convert_tags_to_ids(self, sequence: List[str]) -> List[int]:
```

```
        if not isinstance(sequence, list):
            raise ValueError('Sequence has to be List[str].')

        return [0 if tag not in self.tags else self.tags.index(tag) for tag in
sequence]

    def convert_ids_to_tags(self, sequence: List[int]) -> List[str]:
        if not isinstance(sequence, list):
            raise ValueError('Sequence has to be List[int].')

        return [0 if tag_id > len(self.tags) else self.tags[tag_id] for tag_id
in sequence]
```

문장의 단어 및 태그를 unique index 로 등록해놓기 위한 객체입니다. 일반적으로 `defaultdict()` 와 같은 Dictionary 를 이용하여 Vocabulary 를 구축하지만, 다른 환경에서 `defaultdict()` 로 저장된 binary 가 제대로 동작하지 않는 사례가 간혹 발생합니다. 그렇기 때문에 `List` 로 구현하였습니다.

```
# ./dataset/hmm_prob.py

class HMMProbUtil(object):
    def __init__(self,
                 vocab: VocabUtil) -> None:
        super(HMMProbUtil, self).__init__()
        self.vocab = vocab

    def get_transition_prob(self, data) -> dict:
        states = self.vocab.tags
        prob = np.zeros((len(states), len(states)))

        for sentence in data:
            for i in range(len(sentence)):
                if i == 0:
                    continue
                prob[states.index(sentence[i - 1][1])][states.index(sentence[i]
[1])] += 1
        prob /= np.sum(prob, keepdims=True, axis=1)

        states = self.vocab.tags
        state_dict = {}
        for i, state in enumerate(states):
            state_dict[i] = state

        return_prob = {}

        for i in range(prob.shape[0]):
            for j in range(prob.shape[1]):
                return_prob[state_dict[j]+'|'+state_dict[i]] = prob[i][j]

        return return_prob

    def get_emission_prob(self, data) -> dict:
        vocab_len, states_len = len(self.vocab.vocab), len(self.vocab.tags)
        prob = np.zeros((vocab_len, states_len))

        for sentence in tqdm(data):
            for word, tag in sentence:
```

```
                prob[self.vocab.vocab.index(word)][self.vocab.tags.index(tag)]
+= 1

        sum_prob = np.sum(prob, keepdims=True, axis=0)
        prob = np.divide(prob, sum_prob, out=np.zeros_like(prob), where=sum_prob
!= 0)

        states = self.vocab.tags
        state_dict = {}
        for i, state in enumerate(states):
            state_dict[i] = state

        return_prob = {}

        for i in range(prob.shape[0]):
            for j in range(prob.shape[1]):
                if prob[i][j] == 0.0:
                    return_prob[self.vocab.vocab[i]+'|'+state_dict[j]] = 5e-5
                else:
                    return_prob[self.vocab.vocab[i]+'|'+state_dict[j]] = prob[i]
[j]

        return return_prob

    def get_initial_prob(self, data):
        initial_list = [0] * len(self.vocab.tags)
        for sentence in tqdm(data):
            tag = sentence[0][1]
            initial_list[self.vocab.tags.index(tag)] += 1
        initial_list = np.asarray(initial_list)
        initial_list = initial_list / np.sum(initial_list, keepdims=True)
        initial_probs = {}
        for i, state in enumerate(self.vocab.tags):
            initial_probs[state] = initial_list[i]
        return initial_probs
```

HMM을 위하여 Transition Probability와 Emission Probability 그리고 Initial Hidden States(Initial Probability)를 구축하는 객체입니다.

Transition Probability

$$P(q_i) = q_1, \ldots, q_{i-1} = P(q_i|q_{i-1})$$

위의 수식은 Markov Assumption 이며 이는 특정 state 의 확률은 오직 바로 직전 state 에만 의존성을 가지는 것을 의미합니다. 위의 수식을 바탕으로 Transition Probability 를 구합니다.

$$P(q_i|q_{i-1}) = \frac{C(q_{t-1}, q_i)}{C(q_{i-1})}$$

위의 수식에서 q_i 는 POS 태그를 의미하며 C는 Count를 의미합니다. Transition Probability 는 데이터 집합 내에서 이전 Tag State가 특정 State 와 일치하는 경우의 수를 카운트하고, 특정 state 에서 다음 state 로 넘어가는 경우의 수를 카운트하여 구할 수 있습니다.

Emission Probability

Emission Probability 도 위에서 언급한 Transition Probability 와 유사한 방식으로 구할 수 있습니다.

$$P(w_i|q_i) = \frac{C(q_i, w_i)}{C(q_i)}$$

위의 수식에서 q_i는 동일하게 POS 태그를 의미하고 w_i 는 단어를 의미합니다.

Initial Hidden States

$$P(q_0) = \frac{C(q_o)}{\sum q_0}$$

States 에 대하여 초기 확률 분포를 의미하며 위의 수식을 통하여 얻습니다.

```
def viterbi_forward(states, transition_probs, emission_probs, initial_probs,
observation):
    n = len(states)
    viterbi_list = []
    store = {}

    for state in states:
        viterbi_list.append(initial_probs[state] *
emission_probs[f'{observation[0]}|{state}'])

    for i, ob in enumerate(observation):
        if i == 0:
            continue

        temp_list = [None] * n

        for j, state in enumerate(states):
            x = -1
            for k, prob in enumerate(viterbi_list):
                val = prob * transition_probs[f'{state}|{states[k]}'] *
emission_probs[f'{ob}|{state}']
                if x < val:
                    x = val
                    store[f'{str(i)}-{state}'] = [states[k], val]
            temp_list[j] = x
        viterbi_list = [x for x in temp_list]

    return store, viterbi_list

def viterbi_backward(states, store, viterbi_list):
    num_states = len(states)
    n = len(store) // num_states
    best_seq = []
    best_seq_last = []
    x = states[np.argmax(np.asarray(viterbi_list))]
    best_seq.append(x)

    for i in range(n, 0, -1):
        val = store[str(i) + '-' + x][1]
        x = store[str(i) + '-' + x][0]
        best_seq = [x] + best_seq
```

```
        best_seq_last = [val] + best_seq_last

    return best_seq, best_seq_last
```

\alpha_t(j)는 모델이 주어졌을 때 j번째 상태와 observations 가 나타날 확률을 의미하고 \beta_t(j)는 후방 확률을 의미합니다. a는 전이 확률, b는 방출 확률을 의미합니다.

Forward Algorithm

$$\alpha_t(j) = \sum_{i}^{n} \alpha_{t-1}(i) \times a_{ij} \times b_j(o_t)$$

각각의 전이 확률과 방출 확률 그리고 observation 확률을 더해나가는 수식입니다.

Backward Algorithm

$$v_t(j) = max_j^n [v_{t_1}(i) \times a_{ij} \times b_j(o_t)]$$

전방확률 \alpha와 디코딩 과정에서 구하는 비터비 확률 v 계산은 유사함을 알 수 있습니다. Forward Algorithm은 각 상태에서의 \alpha를 구하기 위해 가능한 모든 경우의 수를 고려해 그 확률들을 더해줬다면(sum), 디코딩은 그 확률들 가운데 최대값(max)에 관심이 있습니다.

위의 수식들을 viterbi_forward, viterbi_backward 로 구현하였습니다.