기초인공지능

Assignment#01

제출 마감: 24.12.08(일) 23:59까지 제출

1. Requirements

- Python version >=3.6

2. 과제 설명

입력으로 복수의 단어로 이루어진 여러 문장들이 주어진다. 각 단어마다 matching된 tag를 읽고, 이를 학습하여 주어진 단어의 tag를 예측하는 Tagger를 구현해야 한다.

구현해야 할 Tagger는 총 2개이며 아래의 설명과 hw1.py내의 주석 설명을 잘 읽고 지시사항을 따라 구현한다. 주어진 hw1.py내의 함수들을 구현하고 에러 이벤트 발생 line (raise NotImplementedError("..."))을 지운 후 zip 포맷으로 압축하여 제출한다.

샘플 데이터로 brown-training.txt와 brown-test.txt가 주어진다. brown-training.txt는 Tagger 학습에 사용되는 데이터이며, brown-test.txt는 학습한 Tagger의 성능 평가에 사용된다. 따라서 brown-test.txt를 사용 시 word만을 input으로 받으며, tag는 Tagger의 예측 결과와 비교하는 데 사용된다. 해당 파일을 읽는 함수는 모두 run.py 내에 정의되어 있다. 단, 직접 run.py 내의 함수를 call해서는 안 된다.

input으로 사용되는 데이터는 train, test가 있다. 각 train, test data는 여러 sentence를 element로 가지는 list이다. 구현할 함수는 train data의 word와 tag를 읽고 Tagger를 학습시킨 후, test data의 word를 바탕으로 tag를 예측하여 (word, tag) pair의 list를 return 하는 함수를 구현해야 한다.

train 데이터 내 k번째 문장에 N_k개의 단어가 있고, 총 m개의 문장이 있다고 가정할 때, train 데이터의 형식은 아래와 같다.

$$[(word1, tag1), (word2, tag2), (word3, tag3), \cdots, (wordN_1, tagN_1)]$$
$$[(word1, tag1), (word2, tag2), (word3, tag3), \cdots, (wordN_2, tagN_2)]$$
$$\cdots\cdots\cdots$$
$$[(word1, tag1), (word2, tag2), (word3, tag3), \cdots, (wordN_m, tagN_m)]$$

위의 예시와 같이 word마다 하나의 tag가 matching 되어 있으며 (word, tag)의 pair(tuple)로 주어진다. 각 문장은 이들 tuple을 element로 가지는 list이다. 모든 문장의 **첫 pair는 ("START", "START")**로 시작한다. 또한, 모든 문장의 **마지막 pair는 ("END", "END")**로 끝난다.

test 데이터 내 k번째 문장에 N_k개의 단어가 있고, 총 m개의 문장이 있다고 가정할 때, test 데이터의 형식은 아래와 같다.

$$[word1, word2, word3, \cdots, wordN_1]$$
$$[word1, word2, word3, \cdots, wordN_2]$$
$$\cdots\cdots$$
$$[word1, word2, word3, \cdots, wordN_m]$$

위의 예시와 같이 word만을 test data로 제공하며 각 함수에서는 주어진 word마다 matching 되는 tag를 예측하여 train data와 같은 형태로(tuple (word, tag)를 원소로 가지는 list) return해야 한다. 모든 문장의 **첫 word는 "START"**, **마지막 word는 "END"**이다. train data에서 나타나지 않은 unseen word가 존재할 수 있다.

Tagger에 관한 설명은 아래와 같다.

Baseline (Baseline Tagger)

Baseline Tagger는 모든 단어가 서로 독립적임을 가정한다. 즉, HMM과 다르게 현재 tag가 이전 tag에 영향을 받지 않는다.

$$P(T_k|w_{1:k}) = P(T_k|w_k)$$

이전 상태로부터 영향을 받지 않으므로, 각 단어마다 tag의 빈도를 바탕으로 확률을 구한 후, 가장 확률이 높은 tag를 선택하도록 구현한다. 만약 test data에서 unseen word가 존재할 경우, train data 내에서 가장 높은 빈도를 보인 tag와 match한다. 이 때, 하나의 word 또는 train data에 대하여 가장 높은 빈도의 tag는 여러 개일 수 있다.

Viterbi (HMM Tagger)

Viterbi algorithm에 따라 현재 tag가 이전 tag에 종속적인 확률을 갖는 HMM을 구현한다.

$$maxP(T_{k+1}|w_{1:k+1}) = P(w_{k+1}|T_{k+1}) \max_{T_k} P(T_{k+1}|T_k) P(T_k|w_{1:k})$$

HMM Tagger는 아래의 세 가지 확률을 고려해야 한다.

- 각 문장의 첫 tag 확률 (initial probability)
- 주어진 tag A 뒤에 나타나는 tag B의 확률 (transition probability)

- 각 tag마다 matching되는 word 확률 (emission probability)

알고리즘은 다음 순서를 따라 구현한다.

① Count tag, tag pairs, (word, tag) pairs
② Take the log of each probability
③ Find the maximum probability path

"START" 이후 첫 tag의 확률 분포(prior)는 "START" tag만을 이전 state로 가지게 되므로, 해당 tag의 확률 분포는 구현 시 각자만의 방법으로 정의하여 구현한다.
② 과정에서 모든 확률에 대해 log를 취하여 더하는 것으로 확률의 곱을 대신한다. 확률 간 곱 연산이 반복되면 긴 단어의 문장에서 계산된 확률이 매우 작아질 수 있다. 이를 방지하기 위해 모든 확률에 log를 취해 scaling 하여 계산한다.

Viterbi_ec (HMM Tagger)

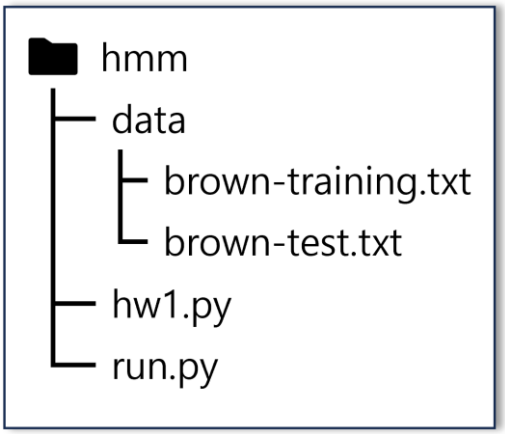
Viterbi algorithm을 실행하면 Baseline Tagger보다 낮은 정확도를 출력한다. 이는 주어진 test data에 unseen words가 포함되어 있기 때문이다. unseen words는 training data에는 등장하지 않은 단어로 zero possibility를 유발한다. 따라서, 각 확률을 계산하기 전에 unseen word에 대한 Laplace smooting을 적용해야 한다.

$$P(w_k|T_k) = \frac{N_{w_k}}{N}\ (original), P(w_k|T_k) = \frac{N_{w_k} + \alpha}{N + \alpha d}\ (smooting)$$

tag T_t에 대해 N은 전체 word의 개수, N_{w_t}는 w_t의 등장 빈도이다. smoothing을 적용할 경우, α는 smoothing parameter, d는 feature dimension이다. Laplace smoothing 함수는 주어진 hw1.py에 구현되어 있다.

Hint 1) Viterbi_ec 정의에 따르면 unseen words에 대해서는 위의 식으로 smoothing을 수행하지만 주어진 test data에서 unseen words는 한 번만 나타난 word와 유사한 tag를 가지고 있다. 이를 고려하여 smoothing하면 unseen words의 성능이 향상된다.
Hint 2) Laplace smoothing 시 smoothing parameter는 작을 때 성능이 오를 수 있다.

3. 실행 방법



1. hmm 폴더로 이동
2. python run.py

폴더 경로 수정 시 에러가 발생할 수 있으니 가급적 주어진 폴더에서 실행하는 것을 추천한다.

4. 채점 기준

```
======== Baseline =========
time spent: 1.7912 sec
accuracy: 0.7975
multi-tag accuracy: 0.7717
unseen word accuracy: 0.6955
======== Viterbi =========
time spent: 6.8568 sec
accuracy: 0.9912
multi-tag accuracy: 0.6783
unseen word accuracy: 0.0635
======== Viterbi_ec =========
time spent: 6.6783 sec
accuracy: 0.8646
multi-tag accuracy: 0.1135
unseen word accuracy: 0.8480
```

총 3개의 함수를 구현해야 하며 run.py 실행 시 위와 같이 출력된다. 위의 사진은 예시 사진으로 출력 값은 random이다. 실제 함수 구현 시 위와 다른 수가 출력된다. 모든 함수의 **time spent는 90초 이내**여야 한다. 90초가 넘는 경우 해당 함수는 구현하지 못한 것으로 간주한다.

Baseline (15점)

- 조건에 맞게 구현한 경우 accuracy, multi-tag accuracy, unseen word accuracy가 특정 값으로 출력된다. 특정 값의 ± 0.1%까지는 올바르게 구현한 것으로 채점한다. 예를 들어, 위의 예시에서 accuracy가 0.7981이 출력된 경우에도 올바르게 구현한 것이다. **각 출**

력 당 3점씩 도합 15점이다.

Viterbi (30점)

- Viterbi 함수는 unseen words를 고려하지 않았기 때문에 baseline보다 성능이 낮아야 한다. 따라서 Baseline을 올바르게 구현한 경우 accuracy와 unseen word accuracy는 **Baseline보다 낮아야 한다**. 최소 accuracy는 90% 이상이다. **accuracy와 unseen word accuracy 각각 15점씩 도합 30점**이다. 지정 범위를 벗어날 경우 구현하지 못한 것으로 간주한다.

Viterbi_ec (45점)

- Viterbi_ec 함수는 unseen words를 고려하여 계산한다. 따라서 성능이 Viterbi보다 향상되어야 한다. **각 출력 당 15점씩 도합 45점**이다. 각각의 값이 Viterbi 함수보다 작은 경우 해당 값은 구현하지 못한 것으로 간주한다. 예를 들어, 위의 예시에서 accuracy와 unseen word accuracy는 Viterbi보다 작으므로 15점으로 채점한다.

과제를 제출한 경우 **기본 점수는 10점**이며 **외부 라이브러리는 numpy만 허용한다.** 만약 제출 양식을 지키지 않거나 numpy 이외의 라이브러리를 사용한 경우 기본 점수 10점에서 감점한다.

5. 제출 방식

hw1.py에서 지정한 함수만 구현하여 zip파일로 압축한 다음 사이버캠퍼스에 제출한다.

제출 시 형식은 **학번_이름_hw1.py**을 압축하여 **학번_이름_hw1.zip으로 제출한다.**