기초 인공지능(CSE4185) Assignment #5

2분반 20200183 박정원

1. Baseline Tagger 및 HMM Tagger의 알고리즘과 구현 방법

Baseline Tagger

```
def baseline(train, test):
    '''
    Implementation for the baseline tagger.
    input:  training data (list of sentences, with tags on the words)
            test data (list of sentences, no tags on the words, use utils.strip_tags to remove tags from data)
    output: list of sentences, each sentence is a list of (word,tag) pairs.
            E.g., [[(word1, tag1), (word2, tag2)], [(word3, tag3), (word4, tag4)]]
    '''
    N = len(train)
    freq = defaultdict(list) # freq = {word : [], ...}
    tags = [] # 태그만 모아둔 리스트, 나중에 unseen word에 대한 처리를 위해
    words = defaultdict(str) # words = {word : tag}, freq 딕셔너리를 바탕으로 최대 빈도 수 태그를 해당 단어와 매치해서 저장.
    for i in range(N):
        for w, t in train[i]: # train[i] is tuple / (word, tag)
            freq[w].append(t) # 해당 단어에 리스트에 매치된 태그를 전부 append
            tags.append(t) # 태그만 모아둔 리스트에 태그 append

    tags_freq = Counter(tags) # train data에 등장한 tag 빈도 수 세기
    Max_tag = max(tags_freq, key=tags_freq.get) # train data에 나타난 가장 많은 빈도 수를 가진 태그, unseen word에 매치할 태그

    for word, tags in freq.items(): # 각 단어에 대한 태그의 빈도 수 처리
        tags_cnt = Counter(tags) # 각 단어의 리스트에 매치된 태그 전부가 들어가 있으므로, Counter를 통해 세기
        words[word] = max(tags_cnt, key=tags_cnt.get) # 최대 빈도 수 가진 태그를 바탕으로 words 딕셔너리에 word : tag 로 추가

    output = []
    M = len(test)
    for i in range(M):
        lst = []
        for w in test[i]:
            tup = None
            if w not in words: # if the corresponding word is not in the words dictionary, it means it is an unseen word
                tup = (w, Max_tag) # match it with the Max_tag
            else:
                tup = (w, words[w]) # match it with the words[w]
            lst.append(tup) # add it to the lst
        output.append(lst) # append the list to output list

    return output
```

Baseline Tagger 는 PDF 상에 나와있는 구현 지시사항에 따라 그대로 구현하였다.
먼저 각 단어 별로 매치된 태그들을 딕셔너리 형태로 관리하는 딕셔너리 freq 와 Test 데이터의 unseen words에 대해서 처리하기 위해 태그 중에서 빈도 수가 가장 높은 태그를 구하기 위해 태그 별 빈도 수를 구하는 tags 자료구조를 Counter 딕셔너리로 관리했다.

words 딕셔너리는 freq 딕셔너리를 바탕으로 각 단어 별로 빈도 수가 가장 높게 매치된 태그를 저장하는 자료구조이다. 매치된 태그가 여러 개인 경우 빈도 수가 가장 높은 태그를 매치해야 하기 때문이다.

이제 테스트 데이터의 모든 단어를 돌면서, words 딕셔너리에 있는 단어이면, 저장되어 있는 태그 값을 반환해주고, unseen words이면 미리 구해둔 태그들 중 가장 빈도 수가

높은 Max_tag를 반환하도록 한다. 이를 (단어, 태그) 형태로 lst에 넣어주고, 이 리스트를 output에 추가하면 된다.

HMM Tagger

HMM Tagger를 구현하기 위해서는 일단 CPT를 구해야 한다. 수업 시간에 다루었던 문제들은 이미 CPT 가 주어진 문제만 다루었으나, 이번 과제에서는 이러한 조건부 확률을 미리 구해 두어야 한다.

과제 지시사항을 확인해보면,

1. Count tag, tag pairs, (word, tag) pairs

라고 되어있다.

Train 데이터 안에 태그들의 빈도 수는 transition probability 나 emission probability를 구할 때 분모 역할을 하기 때문에 필요하다.

또한, tag pairs 는 trainsition probability를 구하기 위해서 관리해야 하는 것으로, Train 데이터 안에서 각 문장에 대해서 인덱스 기준으로 for j in range(1, len(문장)) 으로 (문장[j-1], 문장[j]) 형식으로 세어주면 된다. 수업 시간에 배웠던 것으로 (T_k-1, T_k)의 개념이라고 보면 된다.

(word, tag) pairs는 emission probability를 구하기 위해서 관리해야 하는 것으로, Train 데이터에 나와있는 (word, tag) 별로 빈도 수를 세어주면 된다.

빈도 수를 구할 때는 처음에 리스트에 모두 append 한 후에 Counter(리스트) 로 형변환을 하면 쉽게 딕셔너리 형태로 바꿔줄 수 있다.

```python
def viterbi(train, test):

    '''
    Implementation for the viterbi tagger.
    input:  training data (list of sentences, with tags on the words)
            test data (list of sentences, no tags on the words)
    output: list of sentences with tags on the words
            E.g., [[(word1, tag1), (word2, tag2)], [(word3, tag3), (word4, tag4)]]
    '''
    # 문장 안에서 나오는 단어의 위치가 수업 시간에 배운 time 과 비슷한 개념이라고 생각하자. == 앞에 나온 단어가 뒤에 나온 단어에 영향을 미친다.
    # 구해야 할 것은 inital, emission, transition probability 이다.

    '''
    1. Count tag => emission 구할 때 denominator 역할, tag pairs == (T_k-1, T_k), (word, tag) pairs
    '''
    N = len(train)
    words = [] # 등장한 단어들을 저장
    tags = [] # 등장한 태그들을 저장
    wtot = [] # (word, tag) pair
    ttot = [] # (T_k-1, T_k) pair

    for i in range(N):
        for j, (w, t) in enumerate(train[i]):
            words.append(w) # 단어 추가
            tags.append(t) # 태그 추가
            wtot.append((w,t)) # append word to tag pair
            if j > 0:
                ttot.append((train[i][j-1][1], train[i][j][1])) # append tag to tag pair

    words = Counter(words) # 각 단어 별 빈도 수
    tags = Counter(tags) # 각 태그 별 빈도 수
```

다음은 PDF 구현 지시사항에서 아래 세 가지 확률을 구할 것이다.

- 각 문장의 첫 tag의 확률(Initial probability)
- 주어진 Tag A 뒤에 나타나는 Tag B의 확률
- 각 tag마다 matching되는 word의 확률

먼저, **initial probability** 다.

```python
'''
inital probability 구현
'''
initial = []
for i in range(N):
    initial.append(train[i][0][1]) # 각 문장의 첫 번째 단어가 매칭된 tag append, 전부 "START"가 될 것이다.
initial = Counter(initial) # Counter 딕셔너리로 변환
initialf = []
for t in tags.keys():
    if t in initial:
        initialf.append(initial[t]) # initial에 있다면, 해당 frequency를 append
    else:
        initialf.append(0) # initial에 없다면, 0을 추가
inpb = {} # 태그 별 initial probability를 저장
initialp = smoothed_prob(initialf, 0.001) # Laplace Smooth

for i, t in enumerate(tags.keys()):
    inpb[t] = initialp[i] # 해당 태그의 initial probability를 저장
```

각 문장의 첫 tag는 모두 “START”이다. 하지만, “START”의 확률 값을 1로 하고, 나머지는 전부 0으로 하면, 결국에 Viterbi algorithm을 돌릴 때 나머지 태그들이 상대적으로 반영되지 못하는 문제가 일어나게 된다. 그렇기 때문에 Train 데이터에 나타난 첫번째 태그들의 빈도를 모두 구한 후(이 경우 “START”만 저장되어 있을 것이다.), 없는 태그의 경우 0으로 추가한 다음, 조교님이 구현해둔 smoothed_prob 함수에 각 태그의 빈도가 들어간 리스트를 인자로 넘겨준다. 그렇게 되면, “START” 태그의 확률은 1보다 낮아질 것이고, 확률이 0인 다른 태그의 확률은 0보다 살짝 높아질 것이다. 이렇게 Initial probability를 구했다.

다음은 **trainsition probability** 다.

```
'''
transition probability 구현
'''
ttot = Counter(ttot) # (T_k-1, T_k) 별 빈도 수
trpb = {}
for t in tags.keys():
    denom = 0
    for tt in tags.keys():
        if (t, tt) in ttot:
            denom += (ttot[(t,tt)]) # 분모에 t가 오는 경우 frequency 더 하기
    for tt in tags.keys():
        if (t, tt) in ttot:
            trpb[(t,tt)] = (ttot[(t,tt)] + 0.001) / (denom + 0.001 * len(tags)) # Laplace smoothing
        else:
            trpb[(t,tt)] = 0.001 / (len(ttot)+ len(tags)* 0.001)
```

$$P(T_{k+1}|T_k)$$

Transition probability의 식이다.
ttot 딕셔너리에는 (T_k-1,T_k) 별로 빈도 수가 저장되어 있다.

이 때, T_k-1(코드에서는 t)이 분모에 오게 되는데, 따라서, denom에 t에 대해서 모든 tag pairs의 빈도 수를 더해준다.

이제 모든 tag pairs를 돌면서 t를 T_k-1로 가진 pair에 대해서 transition probability를 계산한다. 이 때 역시 Laplace smoothing을 하는데, 여기에서는 각각의 pair에 대해 하는 것이 편해서, 미리 구현되어 있는 smoothing 함수를 쓰진 않았다. 하지만, 구현되어 있는 함수를 참고하니 분모에는 빈도 수의 합과 리스트 길이 * 알파 값이 들어가는 것을 확인했다. 이를 바탕으로 분자에는 T_k-1 과 T_k의 joint probability를, 분모에는 T_k-1의 확률을 넣어주고, 위 Laplace smoothing을 해주면 된다. 이 때, dimension feature 값은 태그

들의 길이가 될 것이다.
만약 tag pair가 없는 태그들의 경우에는 0으로 하면 안되고, 이 역시 Laplace smoothing을 적용한 결과가 되도록 한다. Denom이 len(ttot), 즉 모든 tag pair들의 개수로 바뀐 것, 분자에는 그냥 0.001값만 있는 것 빼고 구조는 동일하다.

다음은 마지막으로, **emission probability** 이다.
이 부분에서 보고서 2번 해당 과제에서 적용한 개선 사항에 대한 설명(one time word scaling)을 같이 하겠다.
Emission probability는 고려할 것이 더 많다. 우선은 emission probability를 계산할 때에는 unseen words에 대한 처리, 즉 이번 과제에서 데이터셋은 unseen words는 한번만 등장한 단어들과 유사한 태그를 갖는다는 특성이 있기 때문에 이를 고려하여 one time word scaling을 해주어야 한다.

```
'''
emission probability 구현
'''

# emission probability of one time word

once = set() # 한번만 등장한 단어 저장
for w, f in words.items():
    if f == 1:
        once.add(w)

wtoto = {} # (w,t) 의 빈도 수 저장, 이 때 w는 한번만 등장한 단어이다.
empbo = {} # emission probability of one time word
wtot = Counter(wtot)
total = 0
for (w,t), f in wtot.items():
    if w in once:
        wtoto[(w,t)] = f # 한번만 등장한 단어의 경우 wtoto에 빈도 수 저장
        total += f

for t in tags: # 태그 별 one time word에 대한 확률 분포 구하기
    Sum = 0
    for (w,tt) in wtoto:
        if tt == t:
            Sum += wtoto[(w,tt)]
    empbo[t] = (Sum + 0.001) / (total + 0.001 * len(wtoto)) # Laplace smoothing

empb = {} # one time word 확률 분포 적용 emission probability
for (w,t) in wtot:
    empb[(w,t)] = (wtot[(w,t)] + 0.001 * empbo[t]) / (tags[t] + 0.001 * len(tags) * empbo[t]) # one time word scaling with laplace smoothing

unseen_em = {}
for t in tags:
    unseen_em[t] =(0.001 * empbo[t]) / (tags[t] + 0.001 * empbo[t] * len(tags)) # one time word scaling with laplace smoothing for unseen words
```

먼저, 한번만 등장한 단어들을 담을 once 리스트를 준비한다.
그리고 (한번만 등장한 단어, 태그) 의 빈도수를 저장할 wtoto 딕셔너리를 세팅한다.

그리고 transition probability를 구할 때, Denom에 t 태그 빈도 수를 모두 더해준 것과 같이, (한번만 등장한 단어, t)의 빈도수를 Sum이라는 변수에 모두 더해준다.

여기서 좀 더 고민할 부분은 데이터셋에서 한번만 등장한 단어는 많은데다가, unseen words에 대해서는 어떤 태그를 매칭시켜야할지, 매번 모든 한번만 등장한 단어들을 iterate하면서 확률 계산을 해야하는지이다. 처음에는 Viterbi algorithm을 돌릴 때, Unseen

words가 나오면 모든 한번만 등장한 단어를 돌면서 최대 확률 값을 계산하도록 했지만, 이는 time complexity가 너무 커져버려 시간 내에 돌아가지 못했다. 그래서 하나의 태그에 대해 한번 등장한 단어의 빈도 수를 모두 더해서 하나의 태그에 대해 한 확률만 관리하도록 더해준 것이다. 이 Sum이라는 값을 분자에 놓고 Laplace smoothing을 해준다.

이제 원래 Emission probability에 이 One time word 의 확률 분포를 scaling 해줄 차례이다.

모든 (word, tag) pair에 대해서 laplace smoothing을 적용한 확률 값을 딕셔너리에 저장할 것인데, 이 때, 단순히 스무딩 파라미터만을 넣는 것이 아니라, 여기에 앞서 구해둔 one time word에 대한 해당 태그의 확률 값으로 스케일링을 해주면 성공적으로 반영이 된다.

마지막으로, 테스트 데이터의 단어를 볼 때, 단어가 unseen인 경우에는 empb 딕셔너리에서 찾아올 수 없다. 따라서, unseen_em 이라고 따로 이러한 Unseen words에 대한 처리를 위한 딕셔너리를 만들었다. 모든 태그에 대해서 one time word에 대한 확률 분포를 laplace smoothing을 적용하여 할당한다.

2. Viterbi Algorithm에 관한 설명과 해당 과제에서 적용된 개선 사항들에 대한 설명

해당 과제에 적용된 개선 사항, 즉 one time scaling은 위에 emission probability 파트에서 자세히 설명했으므로, 앞에 이어서, Viterbi algorithm에 대한 설명을 하겠다.

위의 initial, transition, emission probability를 구하는 과정은 Viterbi algorithm을 돌리기 위한 사전 작업이다. 이제 비로소 Viterbi algorithm을 돌릴 수 있다.

```
'''
viterbi algorithm
'''
M = len(test)
ans = []
for i in range(M):
    forward = [] # save the probability for [index][tag]
    backward = [] # save the actual tag that is predicted
    output = []
    for _ in range(len(test[i])):
        forward.append({t:0 for t in tags.keys()})
        backward.append({t:0 for t in tags.keys()})
    # 인덱스 0 번째 태그에 대한 전처리
    for t in tags.keys():
        ip = inpb[t]
        if (test[i][0], t) in empb:
            ep = empb[(test[i][0],t)] # word to tag 가 있다면, empb에서 확률 반환
        else:
            ep = unseen_em[t] # 없으면, unknown_em에서 확률 반환
        forward[0][t] = log(ip) + log(ep) # 첫 번째 태그이므로, transition 이 없다.
    for j in range(1, len(test[i])):  # "START" 이후 첫번째 단어부터 predict
        for cur in tags.keys(): # 모든 태그를 돌아본다. 랜덤 변수 T가 가질 수 있는 도메인은 모든 태그이므로
            maxp = -100000000 # most likely tag probability
            pret = None # most likely tag

            if (test[i][j], cur) in empb: # 해당 word가 이미 있으면
                emiss_prob = empb[(test[i][j],cur)] # 미리 구해둔 (word, tag) 확률 반환
            else:
                emiss_prob = unseen_em[cur] # unseen word에 대해 one time word scaling을 적용한 emission probability 반환

            for prev in tags.keys(): # transition probability를 위해서, 이전 태그 iterate
                trans_prob = trpb[(prev, cur)]
                if log(emiss_prob) + log(trans_prob) + forward[j-1][prev] > maxp: # 모든 확률에 대해 log를 취한 후 더 한 것이 maxp보다 크면
                    maxp = log(emiss_prob) + log(trans_prob) + forward[j-1][prev]
                    pret = prev # 해당 태그 저장
            forward[j][cur] = maxp # 해당 확률 저장, 다음 j에서 활용될 것임. Dynamic programming 처럼.
            backward[j][cur] = pret # 나중에 backtrace을 하기 위해 태그 저장

    idx = len(test[i]) - 1 # 뒤에서 부터 backtrace
    max_key = max(forward[idx], key=forward[idx].get) # 문장의 마지막 단어에 대한 태그 가져오기 (최댓값)
    for j in range(idx,0,-1):
        output.append((test[i][j],max_key)) # output 리스트에 (word, 예측 태그) append
        max_key = backward[j][max_key] # backward에 저장해놓았으므로, j일 때 max_key를 만든 이 전 tag 반환
    output.append((test[i][0],max(forward[0], key=forward[0].get))) # 문장의 맨 처음 단어에 대한 태그 가져오기 (최댓값)
    output.reverse() # 역순이었으므로 다시 뒤집어 주기
    ans.append(output) # ans에 output 리스트 추가
return ans
```

먼저, 자료구조에 대한 설명이다.
Forward 리스트는 원소로 딕셔너리를 갖고, 길이는 문장의 길이와 같다. 이 딕셔너리는 모든 태그 t에 대해 0으로 초기화 되어 있다. Forward[i][t] 는 문장에서 i번째 단어에 t라는 태그를 매칭했을 때, 확률이라고 생각하면 되고, 모든 t에 대해서 확률 값을 구하여, viterbi algorithm을 돌릴 것이다.

Backward 리스트는 i번째 단어에 t 태그를 매칭했을 때, 이전 단어의 태그를 저장한다. 즉, 나중에 맨 마지막 단어부터 backtrace 하기 위한 자료구조이다. Forward와 기본적으로 동일한 구조를 가진다.

이제 본격적인 Viterbi 알고리즘에 대한 설명이다.
원래 알고리즘과는 살짝 다른 것이 이번 과제에서는 확률값이 너무 작아지지 않도록 곱을 확률 값에 로그를 취한 값을 더해주는 것으로 바꾸었다. 기본 구조에서 곱을 모두 해당 연산으로 바꾸어주면 된다.

먼저 0번째 태그에 대한 처리를 해주어야 한다. 무조건 해당 단어가 “START”라고 해서, “START” 태그에 대한 확률만 forward에 저장하면 안되고, 앞서 다른 태그들에 대해서도 확률 값을 저장한다. (다음 단어의 확률을 계산할 때 사용되기 때문에) 처음 단어의 경우 transition probabilbity가 없다. 따라서, Emission과 initial probability만을 더해서 확률 값을 저장한다.

다음으로 본격적으로 첫번째 단어부터 해당 문장의 끝 단어까지 For loop을 돌면서 하나씩 태그를 매칭해나갈 예정이다.

For cur in tags.keys() 로 모든 태그를 iterate 한다.
이 때, cur 은

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

이 수식에서 T_k+1을 나타낸다고 보면 된다.

만약 주어진 word와 cur 태그가 Train data에 있던 조합이면, 해당 emission probability를 반환하고, 아니면 unseen_em에서 unseen word에 대한 확률을 반환한다. (one time word scaling이 적용된)

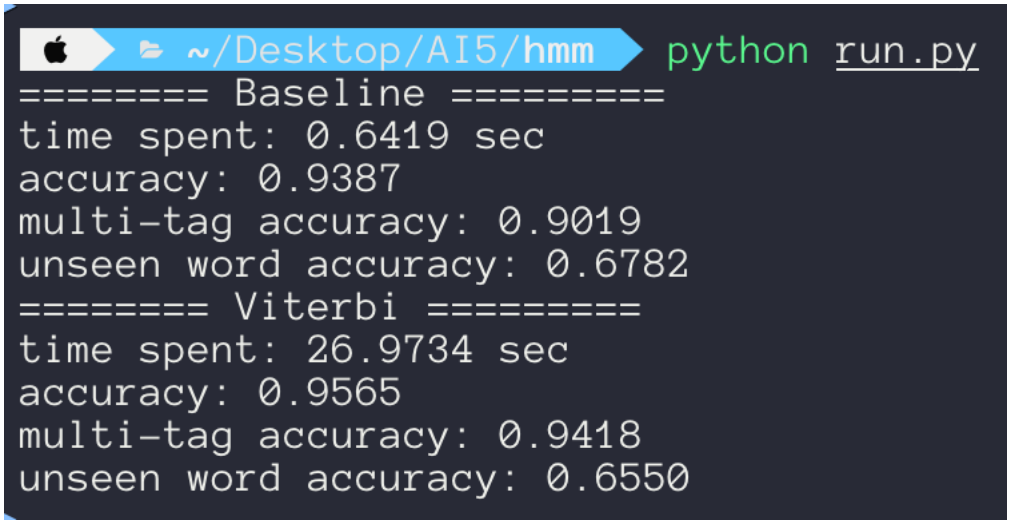
Emission probability를 구했으니 이제 Transition probability를 구해야 하는데, for prev in tags.keys()로 돌면서, 미리 구해둔 transition probability 테이블에서 확률을 가져온다.

이제 모든 값이 구해졌으니 모든 prev 에 대해서 확률을 계산한 후에 최댓값을 저장할 것이다. Log(emission proability) + log(transition probability) + forward[j-1][prev] 가 기존 max 확률보다 크면, 갱신하고, 해당 key를 저장해둔다.

이렇게 최대 확률을 forward[j][cur]에 저장하고, backward[j][cur]에는 해당 prev key를 저장한다. (Viterbi algorithm은 다이나믹 프로그래밍처럼 전의 값을 활용하여 현재 값을 구하는 식이고, backward는 DP에서 경로를 찾을 때 하는 방식처럼 저장된다.)

이런식으로 모든 단어에 대해 Viterbi algorithm을 돌리고 난 후에, 맨 마지막 단어부터 (word, key) 형식으로 output 리스트에 넣어준다. 이 때, key는 backward에 미리 저장해두었으므로, key = backward[j][key] 형식으로 backtrace해서 넣어준다. 그리고 For loop을 나온 후에는 첫번째 단어에 대해서도 (word, key)를 추가하고, 해당 리스트를 reverse 해주면 예측 값들이 저장된 리스트가 완성이 되고, 이를 반환할 리스트에 추가하면 된다.

3. 실행 캡처 화면


~/Desktop/AI5/hmm python run.py
======== Baseline =========
time spent: 0.6419 sec
accuracy: 0.9387
multi-tag accuracy: 0.9019
unseen word accuracy: 0.6782
======== Viterbi =========
time spent: 26.9734 sec
accuracy: 0.9565
multi-tag accuracy: 0.9418
unseen word accuracy: 0.6550