COMP 9318 Data Warehousing and Data Mining T1, 2019

Project 1

Group: Dalao

Wanze LIU (z5137189), Zhou JIANG (z5146092)

School of Computer Science and Engineering

UNSW Sydney

Introduction

This report briefly illustrates the steps we made to accomplish this project. We have three tasks in this project, the first is to find the implicit sequence with the highest probability , the second one is to find top-k path, and the last one is to improve task one

Task 1

Viterbi Algorithm

for HMM, the most useful function is to find the most likely implicit sequence according to its observation, In general, the HMM problem can be described by the following five elements:

```
observations : we observed phenomenon sequence
states : all the possible implicit states
start_probability : the initial probilities of each implicit states
transition_probability : the probility of transfering from one implicit states to another
emission_probability : the probility of some implicit states emit some observed phenomenon
```

If you use the brute-force method to exhaust all possible state sequences and compare their probability values, the time complexity is O(n^m), obviously , this is unacceptable when we want to find a long sequnce with large dataset, however, we can decrease its time complexity by using Viterbi Algorithem,

we can consider this probelm as dynamic programming , the last_state is the probability of each implicit state corresponding to the previous observed phenomenon, and curr_pro is the probability of each implicit state corresponding to the current observed phenomenon. Solving cur_pro actually depends only on last_state, this is core thinking of Vitberi Algorithem.

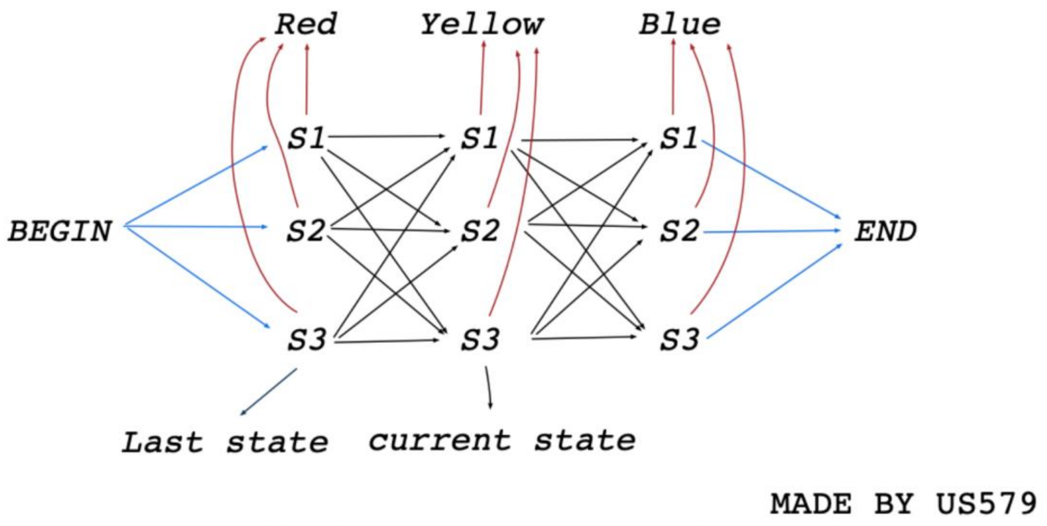
Here, I describe the second sample (toy) as a example to illustrates this algorithem, In this case below, the output of the implicit sequence is

```
[3, 0, 0, 1, 2, 4, -9.843403381747937]
```

with log probility `-9.843403381747937`

Parameter breakdown

```
0:S1       1:S2      2:S3     3:BEGIN      4:END
```



1.Initial Probilities

The blue line represent the initial probability (Pi) which can be deemed as equivalent to transition probabilities from the BEGIN state to all the implicit state
So, we caculate as

```
for s in states[:-2]:
    transition_probability[len(states)-2][states.index(s)])
```

2.Emission Probilities

The red line represent its emission probability from state after smoothing is

$$B[i,j] = \frac{n(i,j)+1}{n(i)+M+1}$$

If the symbol is an unknown symbol, its emission probability from state after smoothing is

$$B[i,j] = \frac{1}{n(i)+M+1}$$

```
for i in range(1,len(obs)):
    for cur in range(len(states[:-2])):
                #if there is no emission from states `cur` to observation
`sym.index(obs[i])`(this is the index in the symbol list),we us add one smoothing (in
case it is 0)
                if str(cur)+'-'+str(sym.index(obs[i])) not in emission_probability:
                    emission_rate = 1.0 / (dic_distance[cur] + n2 +1)
                else:
                #otherwise i will use the formula below
                    emission_rate = emission_probability[str(cur)+'-
'+str(sym.index(obs[i]))]
```

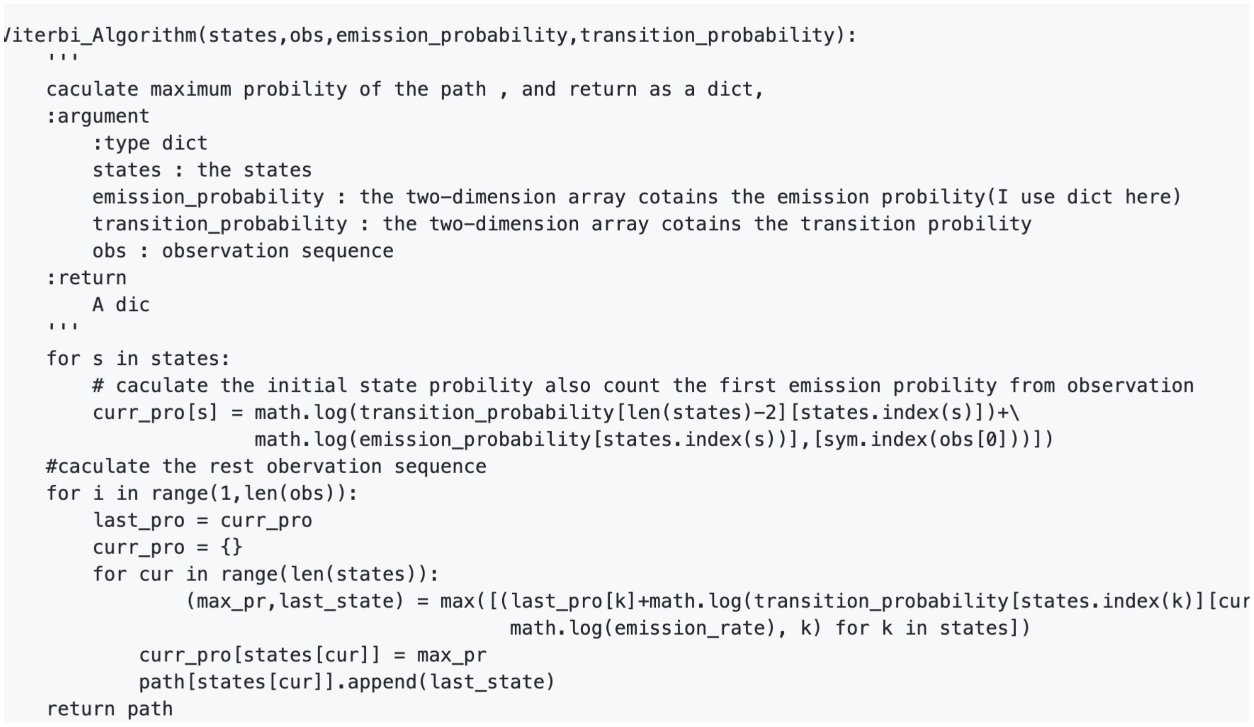
2.Transition Probilities

```
for i in range(n1):
    for j in range(n1):
        transition_probability[i][j] = (float(distance[i][j])+1) /
(sum(distance[i])+n1-1)
```

the number of state_i transfer to state_j divide by the total number of transfering state_j to any states , i also use add-1 smoothing here

3.Viterbi Algorithm

Considering this problem from the problem of dynamic programming, according to the definition of the first figure, last_state is the probability of each implicit state corresponding to the previous observation phenomenon, and curr_state is the probability of each implicit state corresponding to the current observation phenomenon. Solving curr_state actually depends only on last_state. And their dependencies can be expressed in the python code below



```
/iterbi_Algorithm(states,obs,emission_probability,transition_probability):
    '''
    caculate maximum probility of the path , and return as a dict,
    :argument
        :type dict
        states : the states
        emission_probability : the two-dimension array cotains the emission probility(I use dict here)
        transition_probability : the two-dimension array cotains the transition probility
        obs : observation sequence
    :return
        A dic
    '''
    for s in states:
        # caculate the initial state probility also count the first emission probility from observation
        curr_pro[s] = math.log(transition_probability[len(states)-2][states.index(s)])+\
                      math.log(emission_probability[states.index(s))],[sym.index(obs[0]))])
    #caculate the rest obervation sequence
    for i in range(1,len(obs)):
        last_pro = curr_pro
        curr_pro = {}
        for cur in range(len(states)):
                (max_pr,last_state) = max([(last_pro[k]+math.log(transition_probability[states.index(k)][cur
                                            math.log(emission_rate), k) for k in states])
            curr_pro[states[cur]] = max_pr
            path[states[cur]].append(last_state)
    return path
```

Task 2

To return the top-k state sequences, a heap queue is used for sorting, then find K paths having the top K probability, in such steps:

1. Initialize array for terminal probability (i.e. the probability when a path reaches the last layer), along with argument max array (i.e. the current max probability until this layer) and the rank array (i.e. path ranking), which are 3-dimension arrays.

2. Use forward propagation to implement HMM process, testing combinations between current layer and previous layer, in which add-one smoothing is in use.

3. Every time a layer of states is calculated, ranking is changed in heap queue. Also, a fixed-size (size K) ranking dictionary is maintained to store top-K paths. Once a path is no more a top-K paths, it will be replaced.

4. When all observations are calculated, top-K paths are generated by using backward retraction. This is generated based on heap queue which contains node information. Finally, a paths array and a probability array are returned.

Feature:

As we use heap queue rather than the built-in list object to manage the ranking, the querying and sorting time are reduced. Also, we calculate ranking each time when a layer of states has been calculated, avoid timely-excessive steps if sort ranking at the end of searching. Finally, path is generated only at the end of the function, instead of using n^m space to store temporary paths.

Task 3

To optimize the precision of probability of task 1, we consider that Add-one smoothing is not an ideal smoothing method as **1 is considered dominating compared to the little probabilities**, hence the probability is not calculated precisely. **Add-theta smoothing** could perform better than add-one, and easy to implement, which is something like this:

$$P_{Add-1}(w_i|w_{i-1}) = \frac{c(w_{i-1}w_i) + \delta}{c(w_{i-1}) + \delta V}$$

We therefore choose a less-dominated theta to optimize the result. After several attempts, theta=$1*10^{-4}$ has been decided and works well on this HMM model.