Machine Learning Report

50.007

Shang Zewen 1003623
Wang Zilin 1003764
Zhang Shaozuo 1003756

Part 2- The state generation with emission parameters

2.1. Estimate the emission parameters

Emission parameters are estimated with the following equation:

$e(x|y) = \frac{count(y \to x)}{count(y)+k}$ If word token x appears in the training set

$\frac{k}{count(y)+k}$ If word token x is the special token #UNK#

We initialize two python dictionaries which are `state_value_pair` and `state_count` which correspond to `count(y->x)` and `count(y)`.

2.1.1. Fill in the nested dictionary `state_value_pair`

```
Iterate train file:
    If the state is not in the state_value_pair dictionary:
        updating the state to the dictionary
    if the word is not in the state_value_pair[state] dictionary:
        state_value_pair[state].get(word, 1)
        updating the word with the value of 1.
    If the word is already in the corresponding dictionary:
        state_value_pair[state][word]+=1
        increasing the count by 1.
```

After iteration, the nested dictionary `state_value_pair` will contain all the information to calculate the emission parameters as shown below.

```
{'state 1' : {'word 1' : 3, 'word 2' : 5, ... , 'word n ': 2},
...
 'state m': {'word 1' : 7, 'word 2' : 11, ... , 'word n ': 5}}
```

There are two layers. The first layer `state_value_pair.keys()` contains all the states which appeared as a state-word pair in the `train` file. In each of the second layers, each key-value pair `'word 1' : 7` be the occurence of `'word 1'` tagged as `'state m'`.

2.1.2. Fill in `state_count`

Initialize `state_count` dictionary as the following, where `state_count.keys()` is the collections of all unique state
`{'state 1' : 0, 'state 2' : 0,..., 'state m' : 0 }`
Iterate `train` file(Same iteration as 2.1.1., they are running concurrently) and fill in `state_count`

Last but not least, perform division on `state_value_pair` by each of `count(y)+k` to get the emission parameters. For '#UNK#', perform k/(`count(y)+k`) instead. In our case k=0.5. then check whether the token x is in `train` or not, if it is in, which means that the count for appearance will be larger than 0 then we will divide them by using the formula $\frac{Count(u,v)}{count(u) + k}$ where k is 0.5. On the other hand, if the count of appearance is 0 which means the token does not appear in the `train` then we will use the formula $\frac{k}{Count(u)+k}$ where k is 0.5 then all change its state to '#UNK#'.

2.2. The state generation

A function called `get_max_prob` has been written to get the state list that has the highest probability. It simply return y=$y *= argmax(e(x|y))$ over all y(s) to return the most probable state that would emit the word token x. Note that if word token x doesn't exist in `train`, we must set it to be '#UNK#' first.

2.3. Evaluations

```
python .\EvalScript\evalResult.py .\EN\dev.out .\EN\dev.p2.out
python .\EvalScript\evalResult.py .\CN\dev.out .\CN\dev.p2.out
python .\EvalScript\evalResult.py .\SG\dev.out .\SG\dev.p2.out
```

	EN dev.p2.out	CN dev.p2.out	SG dev.p2.out
#Entity in gold data	13179	700	4301
#Entity in prediction	18650	4248	12237
#Correct Entity	9542	345	2386
Entity precision	0.5116	0.0812	0.1950
Entity recall	0.7240	0.4929	0.5548
Entity F	0.5996	0.1395	0.2885
#Correct Sentiment	8456	167	1531
Sentiment precision	0.4534	0.0393	0.1251

Sentiment recall	0.6416	0.2386	0.3560
Sentiment F	0.5313	0.0675	0.1851

Part 3

In part 3, we implement the following:

- Learning the transition parameters and emission parameters with parameters with `train`
- Implementing the Viterbi algorithm with two parts
  - Forward recursion: Generate the scoretable of each sequence in `dev.in`
  - Backtracking: Apply argmax function to return the state list with the highest probability

3.1. Estimate the transition parameters

Read `train` to `pandas.dataframe` named data with `columns=['token', 'tag']`, take note it is necessary to set `skip_blank_lines=False` because `\n` indicates both the end of the previous sequence and the start of a new sequence.

Create a new `pandas.dataframe` named `tranTable` filled with zeros,
set the index name and columns name to be the collections of all unique states with 'START' inserted in the beginning and END inserted in the end.

Iterate all the states in `data` along the rows, there are four scenarios:

1. Middle of a sequence: `tranTable[u][v]+=1`
2. End of a sequence: `tranTable['STOP'][u]+=1`
3. Start of a sequence, `tranTable[u][v]+=1`
4. End of a document, do nothing;

(1)	tag	(2)	tag	(3)	tag	(4)	tag
	v		v				
	u				u		

After the iteration, perform division on the table by corresponding $count(y_i)$

3.2. Estimate the emission parameters

Create a new `pandas.dataframe` named `emiTable` filled with zeros,
set the index name to be the collections of all words in the sequence including '#UNK#',
set columns name to be the collections of all unique states.
Iterate all the words w and states u in data along the rows, do

5. `emiTable[u][w]+=1`

After the iteration, perform division on the table by corresponding $count(y_i) + k$

Last but not least, fill in the emission parameters of '#UNK#' by doing
`emiTable[y_i]['#UNK'] = k/[count(y_i)+k]`

3.3. The Viterbi algorithm implementation

3.3.1. Forward recursion: Generate the score table for each sequence

Instantiate a new pandas.dataframe name scoreTable filled with zeros,
set the index name to be collections of all unique states,
set the columns name to be collections of all words in the sequence including '#START#' in the beginning and '#STOP#' in the end.

#START #	word 1	...	word i-1	work i	...	word n	#STOP
1			v_1	u_1			
1			...	...			
1			v_j	u_j			
1			...	...			
1			v_m	u_m			

Iterate all the words, assume i is the index of the word,eg. '#START#' has i=0

- Base case(i=0):
  `scoreTable.iloc[:,i] = scoreTable.iloc[:,i].add(1)`
  It essentially means $\pi(0, START) = 1$
- Forward recursive case:
  First check whether the word exists in emission table, and assign it to be "#UNK#" if not. After that we perform the following:
  `temp = scoreTable.iloc[:,i-1].mul(tranTable[u])`
  This line is equivalent to $\pi(i,u) = max\{\pi(i-1,v) \times count(v,u)\} for\ every\ v$
  `scoreTable.iloc[uindex,i]=temp.max()*emiTable[u][word]`

  This line fills in the column where current word locates
- Last step(i=n+1):
  `scoreTable.iloc[:,i] = scoreTable.iloc[:,i-1].mul(tranTable['STOP'])`
  This line is equivalent to $\pi(n+1, STOP) = max\{\pi(n,v) \times count(v, STOP)\} over\ v$
  For '#STOP#' column in scoreTable, we don't need to consider the emission parameters since 'STOP' emits nothing.

3.3.2. Backtracking: return the state list for each sequence

Iterate all words in the sequence backwards()

- Base case(i=n-1):
  `last_state = scoreTable['#STOP#'].idxmax()`
- Recursive case:
  `temp =  scoreTable.iloc[:,i+1].mul(tranTable[next_state])`
  `current_state = temp.idxmax()`

SIn order to determine where a state `u` is the most probable state, given the score from START to $\pi(i+1,u)$ and the next most probable state is `next_state`, we have to calculate whether $\pi(i+1,u) \times count(u, nextstate)$ give us the maximum value among all possible `u`. Note that we don't have to consider the emission parameters since the probability of `next_state` emits the word is independent of `u`.

3.4. Evaluations

```
python .\EvalScript\evalResult.py .\EN\dev.out .\EN\part3\dev.p3.out
python .\EvalScript\evalResult.py .\CN\dev.out .\CN\part3\dev.p3.out
python .\EvalScript\evalResult.py .\SG\dev.out .\SG\part3\dev.p3.out
python .\EvalScript\evalResult.py .\test\test.out .\test\part3\dev.p3.out
```

	EN dev.p3.out	CN dev.p3.out	SG dev.p3.out	test dev.p3.out
#Entity in gold data	13179	700	4301	12761
#Entity in prediction	12633	866	4675	12414
#Correct Entity	10139	202	2064	9545
Entity precision	0.8026	0.2280	0.4415	0.7689
Entity recall	0.7693	0.2886	0.4799	0.7480
Entity F	0.7856	0.2547	0.4599	0.7583
#Correct Sentiment	9666	117	1674	9057
Sentiment precision	0.7651	0.1321	0.3581	0.7296
Sentiment recall	0.7334	0.1671	0.3892	0.7097
Sentiment F	0.7490	0.1475	0.3730	0.7195

Part 4 - Top 3 best output sequences

4.1. Buffer

Initialize a path dictionary `path_dict` first which starts with 'START' as the first key and ends with a 'STOP' as the last key, the remaining key will be our path.

```
buffer\Path:    'START'--> 'state 1' --> ...--> 'state m'--> 'STOP'
buffer 1  :             (p,v,from_k_th)
buffer 2  :             (p,v,from_k_th)
buffer 3  :             (p,v,from_k_th)}
```

For each key in `path_dict.keys()` we have k buffers, k is the number of top results we want to return and each Buffer is another python dictionary which contains 3 key-value pairs: (probability, previous state, from_k_th), they are initialized to be `(-sys.maxsize-1,'NA',-1)` The buffer class is implemented in part_4_buffer.py, there are some helper functions to get each key-value pairs within the buffer.

In `path_dict` we initialized every key in the dictionary with a dictionary that contains all the states except 'START' and 'STOP' then the value of each state will be k buffers. Which can help us to retrieve the kth top result we want.

```
State\Path:   ... --> 'state K' --> ...
'state 1': {buffer 1, buffer 2, buffer 3}
...
'state m': {buffer 1, buffer 2, buffer 3}
```

Then we apply the standard Viterbi algorithm to train our model which contains 3 buffers to show the top 3 best results . After the model has been trained, a list called `path_reverse` has been created and another variable will be created which is top, top will be which top result we want it will help us to select which top result we want. Then the algorithm will go through every state in the `path_dict` to retrieve the 3rd best state then append to the `path_reverse`. Since we go through different layers backward which means in the end we need to reverse the `path_reverse` list again to get the correct path for a single entry. After that, another function called `evaluate_doc_part4` has been implemented to gather all the tags for all entries together. The result will be written to `dev.p4.out`.

4.2. Evaluations

python .\EvalScript\evalResult.py .\EN\dev.out .\EN\dev.p4.out

	EN dev.p4.out
#Entity in gold data	13179

#Entity in prediction	13596
#Correct Entity	10486
Entity precision	0.7711
Entity recall	0.7957
Entity F	0.7832
#Correct Sentiment	9929
Sentiment precision	0.7302
Sentiment recall	0.7534
Sentiment F	0.7416

Part 5 - Design challenge- Perceptron

5.1. Perceptron implementation

For the better design of the better design for developing an improved sentiment analysis system for tweets, a perceptron algorithm is applied. Our algorithm aimed at improving the reliability of the transition matrix and emission matrix. The transition and emission matrices are initialized to be all zeros at the beginning. During training, the two tables are updated based on the prediction. The predicted tags given by the Viterbi algorithm using the two matrices are compared with the original tags. If the prediction is correct, nothing happens. However, the values in the matrices will change if there are wrong predictions. For the transition matrix, the wrong prediction of tags will lead to the increase of the values for the correct transitions and the decrease of the values for the wrongly predicted transitions. Similarly, the wrong prediction of tags will lead to the increase of the values for the correct emissions and the decrease of the values for the wrong predicted tag and token pairs. The program runs many iterations until convergence. We had a tradeoff between the execution time and the performance, and set the iteration time to 80. Another improvement is that we have a function to remove URLs from data sets which is called `remove_marks`, when we encounter an URL as an entry, we will replace it with a word ‘URL’ instead of the original link.

5.2. Evaluations

python .\EvalScript\evalResult.py .\EN\dev.out .\EN\dev.p5.out
python .\EvalScript\evalResult.py .\test\test.out .\EN\test.p5.out
`test.out` has not been released yet so we cannot evaluate our `test.p5.out`

	EN dev.p5.out	test test.p5.out
#Entity in gold data	13179	
#Entity in prediction	12986	
#Correct Entity	10575	
Entity precision	0.8143	
Entity recall	0.8024	
Entity F	0.8083	
#Correct Sentiment	10145	
Sentiment precision	0.7812	
Sentiment recall	0.7698	
Sentiment F	0.7755	