ML project

Student

Yu Peijia 1004538

Wu Ji 1004660

Alyssa Ng 1003886

Part 1: Build the emission matrix and find the y that maximizes e(x|y)

The emission matrix will be a `len(distinct_x) * len(distinct_y)` matrix, where the row represents different `x`s appear in `train` or `dev.in`, and the column represents different `y` labels. The matrix can be visualized in the graph below (`n` stands for the total number of distinct words in a file)

	O	B-positive	B-negative	I-negative	I-positive	B-neutral	I-neutral
x1							
x2							
.....							
xn							

Build the emission matrix for `train`

The function `def training_emission_parameters(x,y,x_distinct,y_distinct)` is to build the emission matrix for `train`.

1. `count_uo` represents the occurrences that a state `u` generates an observed variable `o`.
2. `count_u` represents the occurrences of a state `u`.
3. The emission matrix will be `count_uo[i][j] / count_u[j]` for each i-th observed variable in `x_distinct` and j-th state in `y_distinct`.

Build the emission matrix for `dev.in`

The function `def k_emission_parameters(count_uo, count_u, k, train_x_distinct, test_x_distinct, y_distinct)` to build the emission matrix for `dev.in` is similar. We traverse the `x`s in the list `x_distinct` of `dev.in`. When `x` also

exists in the list `x_distinct` of `train`, we give the emission parameter `Count(y -> x) / (Count(y) + k)`, or we give `x` the emission parameter `k / (Count(y) + k)`.

A simple sentiment analysis

The function `def predict(test_x, test_x_distinct,y_distinct,k_emission)` is to implement the simple sentiment analysis to find the `y` that maximizes `e(x|y)` for each x in `dev.in`.We use the emission matrix we get in the above function, and find the column that gives maximized value for each row.

The function `def write_output_from_emission(test_x, y_predict, output_path, x_sequence)` is defined to write the output of the simple sentiment analysis to a new file `dev.p1.out`, which will have the same structure as the file `train`.

Result

ES/ dev.p1.out

Python

```
Entity in gold data: 255
Entity in prediction: 1733

Correct Entity : 205
Entity  precision: 0.1183
Entity  recall: 0.8039
Entity  F: 0.2062

Correct Sentiment : 113
Sentiment  precision: 0.0652
Sentiment  recall: 0.4431
Sentiment  F: 0.1137
```

RU/ dev.p1.out

Python

```
Entity in gold data: 461
Entity in prediction: 2089

Correct Entity : 335
Entity  precision: 0.1604
Entity  recall: 0.7267
Entity  F: 0.2627

Correct Sentiment : 136
Sentiment  precision: 0.0651
Sentiment  recall: 0.2950
Sentiment  F: 0.1067
```

Part 2: Build the transition matrix and implement the Viterbi algorithm

Build the transition matrix

The transition matrix will be a `(len(distinct_y) + 1) * (len(distinct_y) + 1)` matrix, where the row represents the state `v` and the column represents the state `u`. Each value in the cell shows the transition parameter from state `v` to state `u`. The matrix can be visualized in the graph below.

	O	B-positive	B-negative	I-negative	I-positive	B-neutral	I-neutral	STOP
START								
O								
B-positive								
B-negative								
I-negative								
I-positive								
B-neutral								
I-neutral								

The function `def training_transition_parameter(y, y_distinct)` is defined to build the transition matrix. `count_vu` represents the number of occurrences from state `v` to state `u`. `count_v` represents the number of occurrences of state `v`. The list `row` equals to `[START] + y_distinct`, the list `column` equals to `y_distinct + [STOP]`. The transition matrix will be `count_vu[i][j] / count_v[i]` for the i-th state in the list `row` transisting to the j-th state in the list `column`.

Viterbi algorithm implementation

The algorithm is applied to each sentence, not a whole file.

1. The function `def build_emission_probability_matrix(k_emission, x_distinct, ob_sequence, y_length)` is used to select necessary rows to build the emission matrix only for the word in that sentence.
2. `def viterbi(trainsition_probability,emission_probability,obs_seq, y_distinct)` is used to find the possible label for each word in a sentence by Viterbi algorithm.
    a. `pis` is a `len(y_distinct) * (len(obs_seq) + 1)` matrix that contains `π(j, u)` for `j = 1, ....., n` and `u ∈ y_distinxt`, and the last column represents `π(n + 1, STOP)`. Calculate the value of each cell in `pis`:
        i. `j = 1`, we are at the 0-th column, `π(1, u) = pis[u][0]`
        ii. `j = 2, ....., n`, we are at the (j - 1)-th column of the matrix, create a `pi_matrix` to calculate all the possible `π` from state `v` to state `u`, and choose the maximized `π` for each state `u` given `v ∈ y_distinxt`, assign it to `pis[u][j - 1] = π(j, u)`.
        iii. `j = n + 1`, we are at the j-th column, `pis[u][j] = π(n + 1, STOP)`
    b. After getting `pis`, traverse back to predict the state for each word:
        i. `j = n`, we find the state `u` that maximizes `pis[u][j]`, append it to the list `y_pre`
        ii. `j = n - 1, .....,1`, we traverse from `col = n -2, ....., 0` we find `u` that maximizes `pis[u][col] * emission_probability[col + 1][y_n_plus_one] * trainsition_probability[u + 1][y_n_plus_one]` (Here u needs to plus one because the first row of the transition matrix stands for `START`). Here `y_n_plus_one` is the state that is already determined and put into `y_pre` during `j + 1`
        iii. Reverse `y_pre` to get the output, since we start appending from `y_n`
3. The function `write_output_from_viterbi(output_path, transition, k_emission, x_distinct, x_sequence, y_distinct)` is used to write the output of the Viterbi algorithm to a new file `dev.p2.out`. It first builds the new list for each sentence, then creates the emission matrix accordingly, and passes the new list and the

emission matrix plus the transition matrix to `def viterbi()`, finally writes the output with the same structure as `train`.

Result

ES/ dev.p2.out

Python

```
Entity in gold data: 255
Entity in prediction: 551

Correct Entity : 131
Entity  precision: 0.2377
Entity  recall: 0.5137
Entity  F: 0.3251

Correct Sentiment : 104
Sentiment  precision: 0.1887
Sentiment  recall: 0.4078
Sentiment  F: 0.2581
```

RU/ dev.p2.out

Python

```
Entity in gold data: 461
Entity in prediction: 533

Correct Entity : 219
Entity  precision: 0.4109
Entity  recall: 0.4751
Entity  F: 0.4406

Correct Sentiment : 144
Sentiment  precision: 0.2702
Sentiment  recall: 0.3124
Sentiment  F: 0.2897
```

Part 3: Find the 5-th best output sequences

From Part2, we can get the best output path for each sentence, and we can know that the sub-path from location `j + 1` to `STOP` of the best path is the best among all the sub-paths from location `j + 1` to `STOP`. If the statement is not true, when arriving at `j + 1` by changing

the states from `START` to `j`, we can always find a better path from `j + 1` to `STOP`, so the best path should be the path from `START` to `j + 1` plus the newly found best path.

`def fifth_viterbi(trainsition_probability,emission_probability,obs_seq, y_distinct)` finds the 5-th best output paths for each sentence.

1. At first, it calls the function `viterbi()` in part 2 to generate the best path
2. For `i = 0, ....., n - 1`, `j = 1, ....., n` acoordingly where `j` means the location. The path from `j + 1` to `STOP` remains the same as the best path, replace the path from `START` to `j + 1` by random combination of states, calculate the score for each path.
3. Loop until there are more than five paths recorded
4. Sort the paths according to the score, and report the 5-th best paths
5. The path will not be reported if there are less than 5 paths that give non-zero score

A helper class `Cartesian()` is created to find the combination for states from `START` to `j`. To reduce the time complexity, the state that gives zero emission probability of `x` at location j is ignored. The class finds the combination of states by looping `i = j - 1, ....., 1` ,during each loop it fixes states from the first location to the i-th location, and modifies the rest.

> ❗ Although some impossible states are removed, the combination process may still be time-comsuming for long sentence, which is something that can be improved in the future work

Result

ES/ dev.p3.out

Python

```
Entity in gold data: 255
Entity in prediction: 572

Correct Entity : 106
Entity  precision: 0.1853
Entity  recall: 0.4157
Entity  F: 0.2563

Correct Sentiment : 67
Sentiment  precision: 0.1171
Sentiment  recall: 0.2627
Sentiment  F: 0.1620
```

RU/ dev.p3.out

```
Entity in gold data: 461
Entity in prediction: 653

Correct Entity : 163
Entity  precision: 0.2496
Entity  recall: 0.3536
Entity  F: 0.2926

Correct Sentiment : 82
Sentiment  precision: 0.1256
Sentiment  recall: 0.1779
Sentiment  F: 0.1472
```

Part 4: Improved sentiment analysis system

Re-build the emission matrix

1. From part 1 we can see that a k-emission matrix regards all the words that appear in `dev.in` but not in `train` as `#UNK#` and counts the occurrence of all the unknown words one time as a whole. To improve the emission matrix, we assign the same possibility of occurrence to each unknown word.
2. In the matrix `count_uo` that counts the occurrence of a state `u` generating the observed variable `o`, if the variable appears in `train`, we add one to the value at the specific position with the row number of `o` and column number of `u`. If the variable does not appear in `train` but appears in `dev.in`, we assume that it has the equal possibility to be generated by each state `u`, and add `1 / len(y_distinct)` to each column with the row number of `o`.
3. The emission matrix will be `count_uo[i][j] / count_u[j]` for each i-th observed variable in `x_distinct` and j-th state in `y_distinct` of `dev.in`.

Then we will run the viterbi algorithm we get in part 2 using the same transition matrix and the new emission matrix.

Result

ES/ dev.p4.out

Python

```
Entity in gold data: 255
Entity in prediction: 306

Correct Entity : 116
Entity  precision: 0.3791
Entity  recall: 0.4549
Entity  F: 0.4135

Correct Sentiment : 96
Sentiment  precision: 0.3137
Sentiment  recall: 0.3765
Sentiment  F: 0.3422
```

RU/ dev.p4.out

Python

```
Entity in gold data: 461
Entity in prediction: 523

Correct Entity : 220
Entity  precision: 0.4207
Entity  recall: 0.4772
Entity  F: 0.4472

Correct Sentiment : 149
Sentiment  precision: 0.2849
Sentiment  recall: 0.3232
Sentiment  F: 0.3028
```

From the results above, we can see that the number of entities in prediction gets closer to the number of entities in gold data, while the number of correct entities and correct sentiment do not change much compared to the decrease of the number of entities in prediction. It means that there are fewer wrong predictions after modification, which results in the increase of `F-score` for both `correct entity` and `correct sentiment` for `ES` and `RU` increases. We make the hypothesis that it is because we increase the influence of words that only appear in `dev.in` on `count(u)` for each state, the emission possibility is more customed towards the file `dev.in`, so we are less likely to predict them wrongly.