W4705 HW1 (wj2227)

For fast run of all scripts, please use *run.sh*

Question 4

Part1: how to run your code step by step

To run the code, please follow the following commands:

python count_freqs.py ner_train.dat > ner.counts
python question4.py ner_train.dat ner_dev.dat > tagging_baseline
python eval_ne_tagger.py ner_dev.key tagging_baseline

Part2: performance for your algorithm

The performance of the algorithm is show below:

```
I:\W4705-NLP\hw1_>python eval_ne_tagger.py ner_dev.key tagging_baseline
Found 14043 NEs. Expected 5931 NEs; Correct: 3117.

         precision      recall         F1-Score
Total:   0.221961       0.525544       0.312106
PER:     0.435451       0.231230       0.302061
ORG:     0.475936       0.399103       0.434146
LOC:     0.147750       0.870229       0.252612
MISC:    0.491689       0.610206       0.544574
```

Part3: observations and comments about your experimental results.

We could see that the total precision for tagging is 22%, which is low due to we are predicting the tags of each words based on the highest frequency tag for it, without considering the context. So this is served as a baseline and will be compared to the result of trigram model later.
It is also noticed that the counts for "_RARE_" with some tags are huge, which means that there're many words in the training data whose counts are less than 5. Note that B-ORG is missing for _RARE_.

19243 WORDTAG O _RARE_
1466 WORDTAG I-LOC _RARE_
9 WORDTAG B-LOC _RARE_
5462 WORDTAG I-PER _RARE_
10 WORDTAG B-MISC _RARE_
863 WORDTAG I-MISC _RARE_
3207 WORDTAG I-ORG _RARE_

Part4: any additional information that is requested in the problem.

The function that computes emission parameters is named *calc_emission_prob(freqs, ngrams)* in *question4.py*
The new counts after using common symbol "_RARE_" is saved in file *ner.counts_rare*
The baseline result is saved in file *tagging_baseline*

Question 5

Part1: how to run your code step by step

To run the code, please follow the following commands:

python question6.py ner_train.dat ner_dev.dat > tagging_sets
python eval_ne_tagger.py ner_dev.key tagging_sets

python question6_2.py ner_train.dat ner_dev.dat > tagging_sets_2
python eval_ne_tagger.py ner_dev.key tagging_sets_2

Part2: performance for your algorithm

The performance of the two algorithms are show below:

```
I:\W4705-NLP\hw1_>python eval_ne_tagger.py ner_dev.key tagging_rare
Found 4704 NEs. Expected 5931 NEs; Correct: 3646.

         precision      recall         F1-Score
Total:   0.775085       0.614736       0.685661
PER:     0.761838       0.595212       0.668296
ORG:     0.611855       0.478326       0.536913
LOC:     0.876458       0.696292       0.776056
MISC:    0.830065       0.689468       0.753262
```

Part3: observations and comments about your experimental results.

We could see that the total precision for tagging is 77.5% and F1-Score 0.68, which is dramatically higher than the baseline in Question 4, since the trigram model used here takes the context into consideration. Due to the high demand of calculation for Viterbi algorithm, this script takes about 30s to run on my computer.
question5.py should run after *question4.py,* since it takes the outputs of *question4.py* as inputs.

Part4: any additional information that is requested in the problem.

The function that computes parameters

$$q(y_i|y_{i-1}, y_{i-2}) = \frac{Count(y_{i-2}, y_{i-1}, y_i)}{Count(y_{i-2}, y_{i-1})}$$

is named *calc_3gram_q(counts)* in *question5.py*, and the function *viterbi_3gram(sen, e, q, tags)* in *question5.py* implement the Viterbi algorithm. The tagging result is output in file *tagging_rare*

Question 6

Part1: how to run your code step by step

To run the code, please follow the following commands:

python question5.py ner.counts_rare emission_prob ner_dev.dat wordlist_5_or_more > tagging_rare
python eval_ne_tagger.py ner_dev.key tagging_rare

Part2: performance for your algorithm

Two sets classification rules are used to classify rare and unseen words into multiple sets.

1. In first method, rare and unseen words are classified as following:

[Bikel et. al 1999] **(named-entity recognition)**

Word class	Example	Intuition
twoDigitNum	90	Two digit year
fourDigitNum	1990	Four digit year
containsDigitAndAlpha	A8956-67	Product code
containsDigitAndDash	09-96	Date
containsDigitAndSlash	11/9/89	Date
containsDigitAndComma	23,000.00	Monetary amount
containsDigitAndPeriod	1.00	Monetary amount, percentage
othernum	456789	Other number
allCaps	BBN	Organization
capPeriod	M.	Person name initial
firstWord	first word of sentence	no useful capitalization information
initCap	Sally	Capitalized word
lowercase	can	Uncapitalized word
other	,	Punctuation marks, all other words

And the performance shows below:

```
I:\W4705-NLP\hw1_>python eval_ne_tagger.py ner_dev.key tagging_sets
Found 4829 NEs. Expected 5931 NEs; Correct: 3706.

         precision      recall         F1-Score
Total:   0.767447       0.624852       0.688848
PER:     0.737360       0.610990       0.668253
ORG:     0.632710       0.505979       0.562292
LOC:     0.865986       0.694111       0.770581
MISC:    0.826371       0.687296       0.750445
```

2. In second method, rare and unseen words are classified into 5 categories as numbers (_Nun_), capital words (_allCaps_), words containing digits and letters (_containsDigitAndAlpha_), words with capital initial letter (_initCap_) and other (_other_).
   The performance shows below:

```
I:\W4705-NLP\hw1_>python eval_ne_tagger.py ner_dev.key tagging_sets_2
Found 4907 NEs. Expected 5931 NEs; Correct: 3857.

         precision      recall         F1-Score
Total:   0.786020       0.650312       0.711755
PER:     0.792658       0.634385       0.704745
ORG:     0.646018       0.545590       0.591572
LOC:     0.865023       0.726827       0.789926
MISC:    0.820915       0.681868       0.744958
```

Part3: observations and comments about your experimental results.

1. For the first method, the performance is quite close to the result of Question 5 (~ 0.008 decrease in precision and ~0.003 increase in F1-score). It was found that some of the sets doesn't has any rare or unseen words falling in (e.g. *containsDigitAndDash*), and sets only have counts for "O" tag (e.g. *containsDigitAndComma, containsDigitAndPeriod*). It indicates that rather than using only "_RARE_", for this set of data that is not quite big, complex rules of classification like this doesn't help too much. So
2. For the second method, the performance increased some (about 1% improvement in precision and ~0.025 in F1-Score, and is relatively best in the methods above. Better classification rules may exist but needs more trials.

Part4: any additional information that is requested in the problem.

The tagging result is output in file *tagging_sets* and *tagging_sets_2* respectively.