COMS 4705 Natural Language Processing (19 Spring)
Problem Set 1: Programming Part

Yue Luo - yl4003

18 Feb, 2019

Question 4

To run the codes in Question 4, you should:
1: First run **"python 4_1.py"**, this will generate the **"ner_train_rare.dat"**, which is the new training file, with low frequency words replaced by **"_RARE_"** labels.
2. Manually run **"python count_freqs.py ner_train_rare.dat > ner_rare.counts"**. This will provide the new counts for the new training data with replacement.
3. Then run **"python 4_2.py"**. This will generate the predict result **"4_2.txt"** on the validation set **"ner_dev.dat"** with our naive estimator.
4. Finally run **"python eval_ne_tagger.py ner_dev.key 4_2.txt"**, this will compare our predicted results with the actual keys.

The time for running is around 1s, and the result is shown as below:

```
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
```

Figure 1: Result of 5_2

This naive tagger does not perform quite well on the development file. It only gets overall f-1 score of 0.31. Its precision is 0.22 which is quite low, but the recall is much better, which is 0.53. Its recall is even higher for tags MISC and LOC, but for PER and ORG the recall is low.

This estimator only considers each word independently, and it is choosing the tag maximizing $e(word|tag)$. This totally depends on the (word, tag)'s and tag's appearances in the training set. Since it will not consider the context information, a word is always given the same tag. It may have higher recall rate since this tag is the most likely one, but the precision could be low since this word may have another meanings in different context. Since this estimator is too simple, overall f1-score could not be high.

Question 5

To run the codes in Question 5, you should:
1: Directly run **”python 5_1.py”**. This will count and calculate the trigrams, bigrams and the related log q(v|w,u). It will read from **”trigrams.txt”**
2: Directly run **”python 5_2.py”**. This will predict the result using HMM model and Viterbi algorithm. Result will be saved in **”5_2.txt”** .
3. Run **”python eval_ne_tagger.py ner_dev.key 5_2.txt”**, this will compare our predicted results with the actual keys.

The **”5_2.py”** is not fully optimized, and it takes 30s to run and produces the final result. The result is shown as below:

```
Found 4681 NEs. Expected 5931 NEs; Correct: 3579.

         precision       recall          F1-Score
Total:   0.764580        0.603440        0.674519
PER:     0.734564        0.563112        0.637512
ORG:     0.607656        0.474589        0.532942
LOC:     0.872093        0.695202        0.773665
MISC:    0.828758        0.688382        0.752076
```

Figure 2: Result of 5_2

Using a HMM model, we are now considering the context information, and we are now having better ability to understand the meaning of the word given the current context and the words close to it.

So we can see a dramatic increase in the precision for all the tags, and the overall precision has increased by 50%. Overall recall increase by a little, and we see decrease in some of the tags. We can say that the model tends to be more ‘thoughful’ and ’conservative’ when it is going to make a decision. It will always look for the surroundings to get the meaning of it, rathering than just giving the tag that maximizing the possibility of the word given this tag. Therefore precision is highly increased, and recall is not increased by too much.

Overall, the capacity of this tagger has been approved, and the total f-1 score now comes to 0.67.

Question 6

To run the codes in Question 6, you should:
1: Directly run **”python 6.py”**. All the files have been set in the program. This will generate the new counts table **”ner_diffrare.counts”**, and a final predicted file **6.txt** will be generated using the new labels for low frequency words.
2. Run **”python eval_ne_tagger.py ner_dev.key 6.txt”**, this will compare our predicted results with the actual keys.

In this program, we define 5 classes for the low frequency words:
[”_NUM_”]: If all the chars in the word are digits.
[”_ALLUPCASE_”]: If all the chars in the words are UPPERCASEs.
[”_UPCASE_”]: Not all, but contain at least one UPPERCASEs.
[”_SIGN_”]: All the chars are not digits and not alphabets (only signs.)
[”_OTHER_”]: Other cases.

The **”6.py”** takes 30s to run and produces the final result. The result is shown as below:

Found 5829 NEs. Expected 5931 NEs; Correct: 4308.

	precision	recall	F1-Score
Total:	0.739063	0.726353	0.732653
PER:	0.806037	0.784548	0.795148
ORG:	0.528743	0.659940	0.587101
LOC:	0.844860	0.739368	0.788601
MISC:	0.819608	0.680782	0.743772

Figure 3: Result of 6_2

By further dividing the low freqency words based on their patterns, our overall f-1 score increases and it is now 0.73. When we are dealing with the **”_RARE_”** words, we assign the result based on the general case that all those low frequency has.

But after we seperate them into different classes, the charateristics of words in the same class become much more similar. Then different classes are tents to have their own meanings in the sentences. For example, ALLUPCASE may have a greater probability to be a part of name, UPCASE may be a location, and NUM may be a location as well, etc. Even though we do not know what are the exacly relationships between them and the tags, these classes provide more information and the tagger can use this information to better recognize thew words.