Programming Assignment 4 : Sequence Tagging

CSE 256: Statistical NLP: Spring 202

Baseline (4 points)

Our baseline is exact match with PDF for this part, and for rare word, we have come up with the following classes:

- SYM_INIT_CAP
  - The word is start with first letter capital and the rest are lower cases
  - Make use of istitle()
  - Example: Trigram, Hmm
- SYM_SYMBOL
  - The word contains ONLY special chars (no letter or digits)
  - Example: .,]-
- SYM_DIGIT
  - The word contains only digits (can be interpret as number)
  - Example: 8964, 0535
- SYM_LOWERCASE
  - The word is only normal word in lower case
  - Example: somemakeupword
- RARE
  - For all other words not in the above categories.

The intuition behind this is that the sentences usually are combination of word and symbols. However, symbols may usually not be consider impacting sentences meaning. At the same time, digits most of the time stand for data related stuff and may have special meaning in general. Word with capital letter at the beginning, except for the first word in the sentence, usually stands for special Terms, which may be helpful for our Gene cases. Lowercase word usually hard say, but I set this group to help minimize the RARE tag where everything mixed up.

I have tried using only SYM_SYMBOL and RARE and all classes mentioned above, and here is the result:

```
baseline:
Found 2669 GENEs. Expected 642 GENEs; Correct: 424.

     precision   recall     F1-Score
GENE:    0.158861   0.660436    0.256116
---------------------------

group: symbol only
Found 2675 GENEs. Expected 642 GENEs; Correct: 424.

     precision   recall     F1-Score
```

```
GENE:    0.158505   0.660436    0.255653
-----------------------------

group: all classes
Found 2649 GENEs. Expected 642 GENEs; Correct: 424.

      precision   recall     F1-Score
GENE:    0.160060   0.660436    0.257672
```

As we can see, `SYM_SYMBOL` lowers a bit of the precision and F1-Score, possibly because in our case, composition words are separated. E.g. `some-word` are divided into `some, -, word`, which made - into its own category, making some confusion. However, when applied all classes together, we can see a slightly improvement over all values.

Trigram HMM (8.5 points)

The purpose of Viterbi Algorithm is to make use of the previous state we have calculated about the global tag sequences to reduce the workload. Greedy algorithm, by choosing the maximum possibilities each time selecting the tags, fails into the trap of local optimized solution, since each time when choosing a tag, instead of just the current node/tag to the previous word/tag, the former choice also affected current possibilities. E.g. in trigram, the former two words will affect current maximum possibilities sequence, and only evaluate one former word maximum is not enough for the global maximum. Dynamic Programming, by "remember" previous calculation, saved unnecessary, repeated process from brute force but also avoid local maximum dilemma.

In my implementation, I made two dict: `pi` and `bp`. `pi` stores the max possibilities of `(location of word, u, w)`, where `u, w` are tags for its previous two words' tag sequence. `bp` stores max possibilities corresponded tags of the current location. Initially we start with `pi[(0, "*", "*")] = 1`. In each step, we calculate our maximum possibilities on each combination of Tri-tag sequence by looping thru them and get from the previous maximum result.At each stage, I calculate `pi[(k, w, u)] * tri_emi[(w, u, v)] * get_t_emi_prob(sentence[k], v)`, v is the current word's potential tag, and replace `pi[(k + 1, u, v)]` will max value and `bp[(k + 1, u, v)]` save the corresponded maximum possibly tags for later sequence retrieval.

When looping toward the last word, we also need to calculate the special `STOP` word, where it doesn't have a emission value, so the equation would be `pi[(len_of_sentence, u, v)] * tri_emi[(u, v, STOP_WORD)]`. When looping thru different combination of tag `u, v`, we will have the max possibilities, which indicate that current `u, v` is for our second last tag and first last tag for the sentence. Starting here, we could loop thru `bp` from `(len(sentence)~2, first_last, second_last)`. The 2 since we have padded the sentence with `"*", "*"` for Trigram, and each time we use the last two tags for the max path.

Our result is shown below:

```
Found 373 GENEs. Expected 642 GENEs; Correct: 202.

     precision   recall     F1-Score
GENE:   0.541555   0.314642   0.398030
```

Compared with baseline model, it is a huge improvement from ~15% precision to ~55%. However, recall rate has cut half. F1-Score is slightly less than 0.4 but close enough. Potential error may from precision lost during calculation (log with addition is not in use for current implementation). It is believe that the with more context it is more likely for tagger to match accurately, but also limits its capabilities to find less relevant text from training set.

Same as part 2, we perform only SYM_SYMBOL and RARE and all features classes. The result is shown below:

```
baseline:
Found 373 GENEs. Expected 642 GENEs; Correct: 202.

   precision  recall     F1-Score
GENE:   0.541555   0.314642   0.398030

----------------------------
group: symbol only
Found 372 GENEs. Expected 642 GENEs; Correct: 202.

   precision  recall     F1-Score
GENE:   0.543011   0.314642   0.398422

----------------------------
group: all classes
Found 419 GENEs. Expected 642 GENEs; Correct: 220.

   precision  recall     F1-Score
GENE:   0.525060   0.342679   0.414703
```

Compared with baseline tagger, SYM_SYMBOL class has slightly more precision improvement, and all classes has less precision but higher recall rate and F1-Score.

With more classes, model has some advantages to find more relationships than before, however, class with less clear boundary will decrease the precision. There is a tradeoff between more fine-grind tagging and precision. However, all of these classes need more consideration in the context of the text it tries to conquer. Symbols doesn't seen to be a good identifier for the Gene text in our case for example.

Bonus: Non-trivial Extensions (1 point)

In this section, we tried to use 4-grams and compare with 3-grams result:

Trigram
Found 373 GENEs. Expected 642 GENEs; Correct: 202.

	precision	recall	F1-Score
GENE:	0.541555	0.314642	0.398030

4-grams
Found 380 GENEs. Expected 642 GENEs; Correct: 199.

	precision	recall	F1-Score
GENE:	0.523684	0.309969	0.389432

In this comparison, both model uses RARE tag for less frequent word as the first part of Part 2. We can see that 4-grams has slightly less performance than Trigram model, by finding more GENEs but less correct rate.

I believe this issue here is mainly because 4-grams need larger data set for more accurate prediction and current train set does not include enough to support such a complex model. More relationship are marked 0 possibilities since they are not appear in the train set thus negatively impact the outcome. Trigram at this time may have better performance under this dataset.