Report 3

Last update: May 12, 2019

1 Baseline

Given a sentence $X = \{x_1, x_2, \cdots, x_L\}$, the baseline tagger assigns a tag to each word in the sentence and obtains the tag sequence $Y = \{y_1, y_2, \cdots, y_L\}$. To build the baseline tagger, we use **_RARE_** to replace the infrequent words (count<5) in the train file. Then the emission parameters are computed based on the new train file. The emission parameter $e(x|y)$ as described in equation (1) indicates the probability of observing the word x from the tag y. Based on the emission parameters, a word x in the sentence is tagged with $y^* = \arg\max_y e(x|y)$.

$$e(x|y) = \frac{Count(y \rightarrow x)}{Count(y)} \tag{1}$$

We evaluate the baseline tagger on the development set and obtains exactly the same result (see Table 1) as described in **A3-256-sp19.pdf**. Given a word which is rare or unseen, the tagger regards the word as the "word" **_RARE_**. In other words, the infrequent words are divided into one class. Here, we design two methods to group words into informative word classes.

The first method is to use stem words. The method comes from the claim that some words can be assigned to word class on the basis of their form or 'shape' [1]. In our method, the word class is denoted by the stem of the lower case of word and consist of the infrequent words that share the same stem word. For example, the class "pharmacolog" contains "Pharmacologic", "pharmacology", "Pharmacological" and so on. Firstly, we use the Porter Stemmer [2] to stem the the lower case of each infrequent word in the train data and obtain the word classes. The frequency of a class equals to the sum of counts of all words in it. The infrequent (count<5) classes are removed. Given a rare or unseen word, the tagger divides it into the word class that have the same stem word of its lower case. If the word doesn't belong to any obtained word classes, it will be divided into the word class **_RARE_**.

The second method is to use the first n (n=1,2,...,7) letters of the words. The intuition is similar to the first method. Since that words that have similar 'shape' can be assigned to one word class, the words that have the same first n letters can be divided into one word class. The word class is hence denoted by the first n letters of the infrequent word in train data and consist of the infrequent words that have the same first n letters. The

[1]https://www.ucl.ac.uk/internet-grammar/wordclas/criteria.htm

[2]https://tartarus.org/martin/PorterStemmer/

steps for categorizing the rare or unseen words are similar as the first method. Firstly, we obtain the set of the first n letters of the infrequent words in the train file. Then, we calculate the occurrances of every element in the set by adding the word counts and remove the infrequent (count<5) word classes. Given a rare or unseen word, the tagger divides it into the word class that have the same first n letters. The words that don't belong to any classes are categorized into the class **_RARE_**.

Evaluating on the train set, the first improved tagger obtains precision 0.185 and recall 0.71 and F1-score 0.294, the second one achieves precision 0.2, recall 0.7 and F1-score 0.311. The results of evaluating on the development set are presented in Table 1. The row n_classes shows how many word classes each tagger produces. Compared with the baseline tagger, the first improved tagger achieves 1.9% improvement on F1-score. The second improved tagger obtains different results when setting different n. It achieves the highest F1-score of 0.288 and 3.2% improvement compared with the baseline tagger, when setting n=3.

Table 1: Performance of simple gene taggers

n	_	_	1	2	3	4	5	6	7
n_classes	1	1388	76	1059	1730	1549	1353	1131	864
precision	0.159	0.173	0.163	0.179	0.186	0.179	0.176	0.172	0.167
recall	0.660	0.668	0.645	0.634	0.648	0.656	0.662	0.670	0.667
F1-score	**0.256**	**0.275**	0.260	0.280	**0.288**	0.281	0.278	0.274	0.267

2 Trigram HMM

One drawback of the taggers that we build in the last section is that they only consider the current state, i.e. the emission probability when producing a tag. However, the tags that are produced before should be taken into account when making the current prediction. For example, take the trigram model for an example, the probabilities "O" and "I-GENE" assigned after "* O" are 0.95 and 0.045 respectively. This means "O" will be most likely to appear after "* O". To take this transition information into account, we can adopt the Markov Chain Model along with the assumption that the occurrence of the current state relies the previous n states. Besides the transition information, the emission information is also important. Considering the importance of both the emission and transition probabilities, Hidden Markov Model (HMM) is hence applied to this task. Under this model, the probabilty of tagging word x_i with y_i is defined as equation (2). The emission and transition probabilities are calculated according to equation (1) and (3) respectively.

$$p(x_i|y_i) = e(x_i|y_i) * q(y_i|y_{i-2}, y_{i-1}) \tag{2}$$

$$q(y_i|y_{i-2}, y_{i-1}) = \frac{Count(y_{i-2}, y_{i-1}, y_i)}{Count(y_{i-2}, y_{i-1})} \tag{3}$$

2.1 Viterbi algorithm

Based on the probabilistic model HMM, Viterbi algorithm is applied to finding the most likely tag sequence for a given sentence. The Viterbi algorithm is a dynamic programming algorithm for finding the most likely state sequence that results in the observed sequence. Brute force method can also be applied to finding the best tag sequence, however this method will go through all possible solutions to pick the best one, thus it will be very time and memory consuming. Dynamic programming algorithm generally breaks the problem into small pieces so that it narrows down the search space and thus can be much faster than brute force method.

The specific implementation of the Viterbi algorithm is described as following. We first compute the emission parameters and obtain the emission matrix A, which stores the probability of observing a word x from the state (tag) y. The vocabulary contains the **_RARE_** words. Then, we compute the transition parameters and obtain the transition matrix B that stores the transition probability of transiting from the combination of previous two states y_{i-2} and y_{i-1} to the current state y_i. The boundary cases of $q(y_1|*,*)$ and $q(y_2|*,y_1)$ are considered. Given an observation space $X = \{x_1, x_2, \cdots, x_L\}$ and the state space $S = \{"O", "I-GENE", "STOP"\}$, the most likely state sequence $Y = \{y_1, y_2, \cdots, y_L\}$ can be produced by Viterbi algorithm as described in Algorithm 1.

Algorithm 1 Pseudo code of Viterbi algorithm

```
Input: $X = \{x_1, x_2, \cdots, x_L\}$; $S = \{s_1, s_2, s_3\}$; emission matrix A; transition matrix B;
Output: $Y = \{y_1, y_2, \cdots, y_L\}$;
  for each state k=1,2,3 do
      $delta[1, s_k] = A[s_k, x_1] * B[*\_*, s_k]$; $trace[1, s_k] = *$;
  end for
  for each observation i=2,3,..., L do
      for each state k=1,2,3 do
          for each last state m=1,2,3 do
              $T_m = delta[i-1, s_m] * B[trace[i-1, s_m]\_s_m, s_k]$;
          end for
          $delta[i, s_k] = A[s_k, x_i] * \max_m(T_m)$; $trace[i, s_k] = s_{\arg\max_m(T_m)}$;
      end for
  end for
  $a = s_{\arg\max_k(delta[L, s_k])}$; $y_L = a$;
  for each observation i=L-1,...,2,1 do
      $a = trace[i, a]$;
      $y_i = a$;
  end for
  return $Y$
```

2.2 Evaluation

The baseline trigram HMM tagger achieves F1-score of 0.405 on the development set. Compared with the baseline tagger in the last section, it achieves the improvement of 14.6%. We use the same two grouping methods in the last section to improve the baseline HMM tagger.

Evaluating on the train set, the first improved HMM tagger based on stemming achieves precision 0.575, recall 0.359 and F1-score 0.442. The second improved HMM tagger obtains precision 0.565, recall 0.393 and F1-score 0.463. The results of evaluating on the development set are presented in Table 2. We can find the first improved tagger achieves F1-score of 0.412 and the second improved tagger obtains the highest F1-score of 0.432 when setting n=2. Compared with the baseline tagger, the first and second improved tagger achieves an improvement of 0.7% and 2.7% respectively.

Table 2: Performance of trigram HMM taggers

n	_	_	1	2	3	4	5	6	7
n_classes	1	1388	76	1059	1730	1549	1353	1131	864
precision	0.572	0.557	0.537	0.538	0.539	0.546	0.557	0.560	0.568
recall	0.313	0.327	0.307	0.361	0.344	0.333	0.321	0.321	0.319
F1-score	**0.405**	**0.412**	0.390	**0.432**	0.420	0.414	0.407	0.408	0.409

3 Non-trivial Extensions

We add two extensions to the baseline trigram HMM tagger. The first one is implementing the Good-Turing method to smooth the emission parameters. The second is moving to the 4-gram HMM tagger.

The intuition behind implementing the Good-Turing smoothing is that using the maximum likelihood estimates for emissions is not suitable when the occurrance of a word-tag pair is below a minimal threshold. In our implementation, the Good-Turing method is applied to smooth the emission parameters. This extension obtains precision 0.587, recall 0.312 and F1-score 0.407 on the development set. It achieves 0.2% improvement on F1-score over the baseline trigram HMM tagger.

As for the 4-gram HMM tagger, the difference compared with trigram HMM tagger is that it considers the previous 3 words rather than the previous 2 words when calculating the transition probabilities. This extension obtains precision 0.569, recall 0.321 and F1-score 0.410 on the development set. It achieves 0.5% improvement on F1-score over the baseline trigram HMM tagger.