NLP - Problem Set 1 - Programming

Juliana Louback - jl4354@columbia.edu

February 19, 2015

Problem 4

i. Calculate emission parameters $e(x|y)$
Python code: question4.py (lines 23-42)
Usage: python question4.py ner.counts ner_dev.dat > [output_file]
(ner_counts is generated by count_freqs.py)
Pseudocode:
1. Iterate through each line in ner_counts file (containing counts for each word-label association):
1.1 Store each word-label-count combo in a dictionary count_xy containing a dictionary for each word encountered. Each word dictionary contains key-value pairs of the label given to the word and its respective counter. i.e. count_xy[Peter][I-PER] returns the number of times the word 'Peter' was labeled 'I-PER' in the training data.
1.2 The dictionary count_y contains 8 items, one for each label (_RARE_, O, I-MISC, I-PER, I-ORG, I-LOC, B-MISC, B-PER, B-ORG, B-LOC); at each line, add the count to its respective label in count_y to obtain the absolute tag frequency, $Count(y)$.

Note: The actual emission parameter e(x—y) is not calculated; this can be obtained for a given word-label by count_xy[word][label] divided by count_y[label]

ii. Re-label words in training data with frequency < 5 as _RARE_
Python code: label_rare.py
Usage: python label_rare.py [input_file]
Pseudocode:
1. Uses Python Counter to obtain word counts in [input_file]; removes all word-count pairs with count < 5, store remaining pairs in rare_words.
2. Iterates through each line in [input_file], checks if word is in rare_words dictionary, if so, replaces word with _RARE_.

Note: In step 1, running Counter also calculates the tag frequency, but as each of the tags is used at least 5 times this is not an issue; tags are removed leaving only the words considered rare. This isn't optimal, but adequate for the given training data.

iii. Implement a simple named entity tagger
Python code: question4.py (lines 44-67)
Usage: python question4.py ner_counts ner_dev.dat > [prediction_file]
Pseudocode:

```
1. Iterate through each line in ner_dev.dat file
   1.1 If word is in count_xy (see item i.), indicating the word was seen during training, calculate emission = max (count_xy[word][label] / count_y[label]), save label responsible for maximum emission, write ”word label log(emission)” to predictions.
   1.2 If word is not in count_xy, calculate emission = max (count_xy[”_RARE_”][label] / count_y[label]), save label responsible for maximum emission, write ”word label log(emission)” to predictions.
```

iv. Evaluate model
Command: python eval ne tagger.py ner dev.key prediction_file
Output:

```
Found 14042 NEs. Expected 5931 NEs; Correct: 3117.
         precision   recall     F1-Score
 Total:  0.221977    0.525544   0.312121
 PER:    0.435451    0.231230   0.302061
 ORG:    0.475936    0.399103   0.434146
 LOC:    0.147764    0.870229   0.252632
 MISC:   0.491689    0.610206   0.544574
```

According to the results above, the model has rather poor performance. It excessively tags words as NE (14,042 tagged NE in a dataset containing less than 6,000 NEs) and even so only identified about half of the NEs in the dataset correctly. It predicts NEs in the MISC category best, with 49% precision and 61% recall. The performance indices for the LOC category suggest the model is 'blindly' tagging words as a LOC type NE, as it has a high recall index (87%) indicating it correctly identified most of the NE's in the dataset, but a very low precision level (14%), indicating that very few of the LOC predictions were correct.

Problem 5

i. Function that computes parameters $q(y_i|y_{i-1}, y_{i-2})$
Python code: question5a.py
Usage: python question5_a.py n-gram_counts [input_file] > [output_file]
I have placed the n-gram counts from the ner.counts file generated by count_freqs.py in a file named n-gram_counts
Pseudocode:

```
1. Iterate through each line in the n-gram_counts file
1.1 If the line contains '2-GRAM', add an item to the bigram_counts dictionary using the bigram as key, count as value. This dictionary will contain $Count(y_{i-2}, y_{i-1})$.
1.2 If the line contains '3-GRAM', add an item to the triigram_counts dictionary using the trigram as key, count as value. This dictionary will contain $Count(y_{i-2}, y_{i-1}, y_i)$.

2. Iterate through each line in the [input_file]
2.1 Single out the space separated tags $y_{i-2}, y_{i-1}, y_i$, from the line, generate a trigram string and a bigram string;
2.2 If both the trigram $(y_{i-2}, y_{i-1}, y_i)$ and the bigram $y_{i-2}, y_{i-1})$ are found in trigram_counts and bigram_counts respectively, calculate the log probability given by log(trigram_counts[trigram]/bigram_counts[bigram]). Write the current line to the output file, with an additional column containing the log probability.
2.3 If both the trigram $(y_{i-2}, y_{i-1}, y_i)$ and the bigram $y_{i-2}, y_{i-1})$ are not found in trigram_counts and bigram_counts, set log probability to zero. Write the current line to the output file, with an additional column containing the log probability.
```

Note: This does not calculate the accumulated probability, only the probability of individual trigrams.

ii. Implement the Viterbi algorithm
Python code: question5b.py
Usage: python question5_b.py ner.counts ner_dev.dat> [output_file]
Pseudocode:

```
1. Iterate through each line in the file ner.counts
1.2 If the line contains 'WORDTAG'
    1.2.1 Store each word-label-count combo in a dictionary count_xy. (See 4, Step 1.1)
    1.2.2 Add label-count as key-value to the dictionary count_y (See 4, Step 1.2)
1.3 Else (for lines containing N-GRAM counts, code same as item 5i, Step 1)
    1.3.1 If the line contains '2-GRAM', add to the bigram_counts dictionary using the bigram as key, count as value for $Count(y_{i-2}, y_{i-1})$.
    1.3.2 If the line contains '3-GRAM', add to the trigram_counts dictionary using the trigram as key, count as value for $Count(y_{i-2}, y_{i-1}, y_i)$.
```

Once step 1 is complete, we have $Count(x \rightsquigarrow y), Count(y)$ *to calculate emission and* $Count(y_{i-2}, y_{i-1}, y_i), Count(y_{i-2}, y_{i-1})$ *to calculate the trigram parameter* $q(y_i|y_{i-1}, y_{i-2})$. *Step 2 is the Viterbi algorithm which makes use of these values.*

```
2. Iterate through each line in the file ner_dev.dat.
2.1 If the word was seen in training data (present in the count_xy dictionary), for each
of the possible labels for the word:
    2.1.1 Calculate emission = count_xy[word][label] / float(count_y[label]
    2.1.2 Calculate $q(y_i|y_{i-1}, y_{i-2})$ = trigram_counts[trigram])/float(bigram_counts[bigram]
    Note: $y_{i-2} = *, y_{i-1} = *$ for the first round
    2.1.3 Set probability = emission $\times q(y_i|y_{i-1}, y_{i-2})$.
    2.1.4 Update max(probability) and arg max if needed.
2.2 If the word was not seen in the training data, for each label for '_RARE_':
    2.2.1 Calculate emission = count_xy[_RARE_][label] / float(count_y[label].
    2.2.3 Calculate $q(y_i|y_{i-1}, y_{i-2})$ = trigram_counts[trigram])/float(bigram_counts[bigram].
    Note: $y_{i-2} = *, y_{i-1} = *$ for the first round
    2.2.4 Set probability = emission $\times q(y_i|y_{i-1}, y_{i-2})$.
    2.2.5 Update max(probability) if needed, arg max = label
2.2 Write arg max and log(max(probability)) to output file.
2.3 Update $y_{i-2}, y_{i-1}$.
```

iii. Evaluate model

Command: `python eval_ne_tagger.py ner_dev.key prediction_file`

Output:

```
Found 4188 NEs. Expected 5931 NEs; Correct: 3093.
         precision    recall   F1-Score
 Total:  0.738539   0.521497   0.611325
 PER:    0.738215   0.400435   0.519224
 ORG:    0.556322   0.361734   0.438406
 LOC:    0.833442   0.698473   0.760012
 MISC:   0.755102   0.642780   0.694428
```

The Viterbi algorithm presents significantly better performance than the first model in almost all counts. When it comes to recall, both models performed similarly: 3090/5931 NE's correctly identified using the Viterbi algorithm and 3117/5931 NEs using the emission-only model. However, the Viterbi algorithm is much more precise, without under fitting the data to boost recall. The total precision level is near 74% compared to a mere 22% obtained previously. The most dramatic change is seen in the LOC category, going from 14% precision to 83%, suggesting this tag is particularly dependent on the context. Curiously, the recall index is significantly less impressive in the Viterbi algorithm. This may simply be due to the fact that as the emission model tagged 3 times as many words as NE's, by the rule of averages it boosted the recall. Another interesting observation is that the performance in the ORG category is quite similar in both models: 55% precision and 36% recall for the Viterbi model compared to 47% precision and 39% recall for the emission model.

Problem 6

i. New symbol for uncommon words
Python code: label_rare2.py
Usage: python label_rare2.py ner_train.dat
Pseudocode (similar to 4i):

```
1. Uses Python Counter to obtain word counts in [input_file]; removes all word-count
pairs with count $< 5$, store remaining pairs in rare_words.
2. Iterates through each line in [input_file], checks if word is in rare_words dictionary.
If so, if word is:
    - All numeric: replaces word with _NUMBERS_.
    - Mix of numbers and dashes: replaces word with _NUMBER_CODE_.
    - A possible date: replaces word with _DATE_.
    - All upper case: replaces word with _ALL_UPPER_.
    - First letter is upper case: replaces word with _FIRST_UPPER_.
    - Default: replaces word with _RARE_.
```

ii. Viterbi algorithm + new symbols
Pseudocode: Just as in 5ii, with the following modification:

```
2.2 If the word was not seen in the training data, check what symbol category it fits in (
_NUMBERS_, _NUMBER_CODE_, _DATE_,_ALL_UPPER_,_FIRST_UPPER_,_RARE_).
2.3 For each
    2.2.1 Calculate emission = count_xy[word][symbol] / float(count_y[label].
    2.2.3 Calculate $q(y_i|y_{i-1}, y_{i-2})$ = trigram_counts[trigram])/float(bigram_counts[bigram].
    2.2.4 Set probability = emission $\times q(y_i|y_{i-1}, y_{i-2})$.
    2.2.5 Update max(probability) if needed, arg max = label
```

iii. Evaluate model
Output:

```
Found 6105 NEs. Expected 5931 NEs; Correct: 4003.
          precision   recall     F1-Score
 Total:   0.655692    0.674928   0.665171
 PER:     0.545918    0.873232   0.671829
 ORG:     0.633373    0.402840   0.492462
 LOC:     0.835294    0.696838   0.759810
 MISC:    0.741071    0.630836   0.681525
```

The precision levels are similar but mostly lower than those presented by the model in section 5 that used only the _RARE_ symbol. In spite of that, in this model the recall levels are all significantly higher, resulting in a higher or equal F1-Score. As the new set of symbols was chosen based on anecdotal evidence, it is suggested that this model can be further improved by a more adequate selection of symbols to represent categories of uncommon words.