Programming Assignment 3 : Sequence Tagging

CSE 256: Statistical NLP: Fall 2019

University of California, San Diego

Released: October 31, 2019

Due: November 13, 2019

Given a natural language sentence $\boldsymbol{x} = \langle x_1, x_2, \ldots, x_n \rangle$, the task of sequence tagging is to find the most likely sequence of tags $\hat{\boldsymbol{y}} = \langle y_1, y_2, \ldots, y_n \rangle$.
For this homework you will build a Hidden Markov Model (HMM) to tag gene names in biological text — an example problem of named entity recognition. Under this model the joint probability of a sentence $x_1, x_2, \ldots, x_n$ and a tag sequence $y_1, y_2, \ldots, y_n$ is defined as:

$$P(x_1 \ldots x_n, y_1 \ldots y_n) = q(y_1|*,*) \times q(y_2|*, y_1) \times q(\texttt{STOP}|y_{n-1}, y_n) \times \prod_{i=3}^{n} q(y_i|y_{i-2}, y_{i-1}) \times \prod_{i=1}^{n} e(x_i|y_i)$$

where $*$ is a padding symbol that indicates the beginning of a sentence and `STOP` is a special HMM state indicating the end of a sentence. Your task will be to implement this probabilistic model and a decoder for finding the most likely tag sequence for new sentences.

1 Preliminaries

1.1 Data

We provide a labeled training data set `gene.train`, a labeled and unlabeled version of the development set, `gene.key` and `gene.dev`, and an unlabeled test set `gene.test`. Notice that there is no labeled test set for this assignment, reporting performance on the dev set will suffice. The labeled files take the format of one word per line with the word and tag separated by space and a single blank line separates sentences, e.g.

```
Comparison O
with O
alkaline I-GENE
phosphatases I-GENE
and O
5 I-GENE
- I-GENE
nucleotidase I-GENE

Pharmacologic O
aspects O
of O
neonatal O
hyperbilirubinemia O
.O
...
```

The unlabeled files contain only the words of each sentence and will be used to evaluate the performance of your model. The task is to identify gene names within biological text. In this dataset there is one type of entity: gene (GENE). The dataset is adapted from the BioCreAtIvE II shared task.

Several utility scripts are provided to help with the assignment. The scripts can be called at the command-line to pre-process the data and to check results.

1.2 Collecting Counts

The script `count_freqs.py` handles aggregating counts over the data. It takes a training file as input and produces trigram, bigram and emission counts. To see its behavior, run the script on the training data and pipe the output into a file.

```
python count_freqs.py gene.train > gene.counts
```

Each line in the output contains the count for one event. There are two types of counts:

- Lines where the second token is `WORDTAG` contain emission counts $Count(y \rightsquigarrow s)$, for example

  ```
  13 WORDTAG I-GENE consensus
  ```

  indicates that `consensus` was tagged 13 times as `I-GENE` in the the training data.

- Lines where the second token is `n-GRAM` (where `n` is `1, 2` or `3`) contain unigram counts $Count(y)$, bigram counts $Count(y_{n-1}, y_n)$ or trigram counts $Count(y_{n-2}, y_{n-1}, y_n)$, for example

  ```
  16624 2-GRAM I-GENE O
  ```

  indicates that there were 16624 instances of an `O` tag following an `I-GENE` tag and

  ```
  9622 3-GRAM I-GENE I-GENE O
  ```

  indicates that in 9622 cases the bigram `I-GENE I-GENE` was followed by an `O` tag.

1.3 Evaluation

The script `eval_gene_tagger.py` provides a way to check the output of a tagger. It takes the correct result and a user result as input and gives a detailed description of accuracy.

```
python eval_gene_tagger.py gene.key gene_dev.p1.out

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

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

Results for gene identification are given in terms of precision, recall, and F1-Score. Let $\mathcal{A}$ be the set of instances that our tagger marked as `GENE` and $\mathcal{B}$ be the set of instances that are correctly GENE entities. Precision is defined as $|\mathcal{A} \cap \mathcal{B}|/|\mathcal{A}|$ whereas recall is defined as $|\mathcal{A} \cap \mathcal{B}|/|\mathcal{B}|$ F1-score represents the geometric mean of these two values.

2 Baseline (30%)

- Using the counts produced by `count_freqs.py`, write a function that computes emission parameters

  $$e(x|y) = \frac{Count(y \rightsquigarrow x)}{Count(y)}$$

- We need to predict emission probabilities for words in the test data that do not occur in the training data. One simple approach is to map infrequent words in the training data to a common class and to treat unseen words as members of this class. Replace infrequent

words ($Count(x) < 5$) in the original training data file with a common symbol `_RARE_`. Then re-run `count_freqs.py` to produce new counts.

- As a baseline, implement a simple gene tagger that always produces the tag $y^* = \arg\max_y e(x|y)$ for each word x. Make sure your tagger uses the `_RARE_` word probabilities for rare and unseen words. Your tagger should read in the counts file and the file `gene.dev` (which is `gene.key` without the tags) and produce output in the same format as the training file. For instance

  Write your output to a file called `gene_dev.p1.out` and evaluate by running

  ```
  python eval_gene_tagger.py gene.key gene_dev.p1.out
  ```

  The expected result should match the result above.

- Your tagger can be improved by grouping words into informative word classes rather than just into a single class of rare and unseen words. Propose more informative word classes for dealing with such words.

  In your report, describe your design decisions for rare and unseen words. Compare the model performance of a few different design choices on the train and development sets. Pick your best model and report its performance on the dev set.

3 Trigram HMM (60%)

- Using the counts produced by `count_freqs.py`, write a function that computes parameters

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

  for a given trigram y_{i-2} y_{i-1} y_i. Make sure your function works for the boundary cases: $q(y_1|*,*)$, $q(y_2|*, y_1)$, $q(\texttt{STOP}|y_{n-1}, y_n)$.

- Using the maximum likelihood estimates for transitions and emissions, implement the Viterbi algorithm to compute

  $$\arg\max_{y_1 \ldots y_n} \; p(x_1 \ldots x_n, y_1 \ldots y_n)$$

  Be sure to replace infrequent words ($Count(x) < 5$) in the original training data file and in the decoding algorithm with a common symbol `_RARE_` . Your tagger should have the same basic functionality as the baseline tagger.

  Run the Viterbi tagger on the development set. The model should have a total F1-Score of 40.

  Your report should include a brief summary (e.g., a short paragraph) of your Viterbi algorithm implementation and any specific challenges or issues that came up. If you could not get your implementation working, describe where you got stuck.

- Here again, your tagger can be improved by grouping words into informative word classes rather than just into a single class of rare and unseen words. Propose more informative word classes for dealing with such words. Using the same strategy in Section 2 is fine.

  In your report, describe your design decisions for rare and unseen words. Compare the model performance of a few different design choices on the train and development sets. Pick your best model and report its performance on the dev set.

- The Collins notes on HMMs should be very helpful for this part of the assignment.

4 Non-trivial Extensions (10%)

There are several extensions you can consider making: smoothing the parameters, moving to 4-grams or higher, and so on. Add two non-trivial extensions to your tagger.

5 Report Guideline

5.1 Baseline

a. Design two choices of informative word classes to replace rare/unseen words. What is the intuition behind your choice of word classes? Given a word, how would you categorize it into classes for each design method?

b. Evaluate and compare the new baseline model(s) on train and dev sets

5.2 Trigram HMM

a. What is the purpose of the Viterbi algorithm – dynamic programming vs. brute force vs. greedy?

b. What are the specifics of your implementation: what is the base case; how did you implement the recursive formulation; how did you obtain the joint probability of word sequence and tag sequence; how do you go from this joint probability to final tag sequence, i.e., what path in your dynamic programming table gives you the final tag sequence?

If you got stuck and your implementation is not working, describe the key challenges and the issues you faced.

c. Evaluate the HMM tagger on the dev set, verify the F-1 score; how does it compare to the baseline tagger?

d. Evaluate the new HMM model on dev set with informative word classes you designed.

e. Provide insightful comparison based on these results;

5.3 Extensions

a. Describe the extensions you made to the Trigram HMM tagger, evaluate the extended model and provide analysis.

b. Specific emphasis is placed on improved performance w.r.t the Trigram HMM model

6 Submission Instructions

Submit your work on Gradescope.

- **Code:** You will submit your code together with a neatly written README file to instruct how to run your code with different settings. We assume that you always follow good practice of coding (commenting, structuring), and these factors are not central to your grade.
- **Report:** Submit your report, it should be **four pages long, or less, in pdf** (reasonable font sizes).

7 Acknowledgments

Adapted with some changes from a similar assignment by Michael Collins.