Traitement automatique du langage
TP 3 — Identify gene names with HMMs

Haozhou Wang
Exercises prepared by Yves Scherrer
10.10.2018

Submit by 23.10.2019 midnight.

In this assignment, you will build a trigram hidden Markov model to identify gene names in biological text.[1]

Data

Input files for the assignment are located in the *genes.zip* archive, to be downloaded from Moodle. We provide a labeled training dataset (*gene.train*), labeled and unlabeled versions of the development dataset (*gene.key* and *gene.dev*, respectively), and an unlabeled test dataset (*gene.test*). Labeled files contain one word per line, with the associated tag on the same line, separated by a whitespace. Sentences are separated by a single blank line:

```
Pharmacologic NOGENE
aspects NOGENE
of NOGENE
neonatal NOGENE
hyperbilirubinemia NOGENE
. NOGENE

Beta NOGENE
blocking NOGENE
agents NOGENE
. NOGENE
```

[1]This practical assignment is a slightly adapted version of an assignment given by Michael Collins on Coursera. The course is not accessible anymore, but the related notes are still available at `http://www.cs.columbia.edu/~mcollins/hmms-spring2013.pdf`.

Unlabeled files contain one word per line, without any associated tag, and will be used to evaluate the performance of the model.

The task consists in identifying gene names within biological text. There is one entity of interest in the dataset, and hence only two labels: *GENE* and *NOGENE*. The dataset has been adapted from the BioCreAtIvE II shared task.[2]

Utility scripts

To help out with the assignment, we provide two utility scripts (inside the *genes.zip* archive): one for collecting counts, and one for evaluating the model.

Frequency counts

The *countfreqs.py* script aggregates counts over the data. It takes a training file as input, and produces trigram, bigram and emission counts as output. You should run the script on the training data and redirect the output into a file:

```
$ python3 countfreqs.py gene.train > gene.counts
```

Each line in the output corresponds to the count for one event. There are two types of events:

- Lines where the second token is *WORDTAG* indicate emission counts $Count(y \rightsquigarrow x)$. For example, the following line indicates that *consensus* was tagged 13 times as *GENE* in the the training data:

  ```
  13 WORDTAG GENE consensus
  ```

- Lines where the second token is *[123]-GRAM* indicate unigram counts $Count(y)$, bigram counts $Count(y_{n-1}, y_n)$, and trigram counts $Count(y_{n-2}, y_{n-1}, y_n)$. For example, the following line indicates that there were 16624 instances of the *NOGENE* tag being immediately preceded by the *GENE* tag.

  ```
  16624 2-GRAM GENE NOGENE
  ```

  And the following line indicates that there were 9622 instances of the *NOGENE* tag being immediately preceded by the bigram *GENE GENE*.

  ```
  9622 3-GRAM GENE GENE NOGENE
  ```

[2] http://biocreative.sourceforge.net/biocreative_2.html

Evaluation

The *evaltagger.py* script can be used to evaluate the output of a tagger. It takes the gold standard file and the predictions as input, and returns the performance of the tagger.

```
$ python3 evaltagger.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 A be the set of instances that our tagger marked as *GENE*, and B be the set of instances that are marked as *GENE* in the gold standard file. Precision is defined as $\frac{|A \cap B|}{|A|}$, whereas recall is defined as $\frac{|A \cap B|}{|B|}$. The F1-score represents the harmonic mean of these two values.

1 A unigram tagger

We start by implementing a baseline tagger that only uses emission counts to label words.

- Using the counts produced by *countfreqs.py*, write a function that computes emission parameters:
$$e(x \mid y) = \frac{Count(y \rightsquigarrow x)}{Count(y)}$$
- 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 training dataset with a common symbol *_RARE_*, then re-run *countfreqs.py* to produce counts that take the *_RARE_* class into account.

  Note: You can also replace infrequent words in the counts file. However, when doing so, make sure to convert only words that have a frequency lower than 5 across both classes.
- Implement a simple gene tagger that always produces the tag $y^* = \arg\max_y e(x \mid y)$ for each word x. Make sure the tagger uses the *_RARE_* word probabilities for rare and unseen words.

  The tagger should take the counts file and the development dataset (*gene.dev*) as input, and produce as output predictions in the same format as the training dataset.
- Write the output to a file named *gene.dev.p1.out*, and evaluate it by running:

```
$ python3 evaltagger.py gene.key gene.dev.p1.out
```

The expected result should match the result shown above under Evaluation.

- Run the tagger on *gene.test* and write the output to a file named *gene.test.p1.out*. Please hand in this file as part of the assignment.

2 A trigram HMM tagger

In a trigram HMM tagger, the joint probability of a sentence $x_1\ x_2 \dots x_n$ and a tag sequence $y_1\ y_2 \dots y_n$ is defined as

$$p(x_1 \dots x_n, y_1 \dots y_n) =$$

$$q(y_1 \mid *, *) \cdot q(y_2 \mid *, y_1) \cdot \prod_{i=3}^{n} q(y_i \mid y_{i-2}, y_{i-1}) \cdot \prod_{i=1}^{n} e(x_i \mid y_i) \cdot q(\text{STOP} \mid y_{n-1}, y_n)$$

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 is to implement this probabilistic model, and a decoder for finding the most likely tag sequence of new sentences.

- Using the counts produced by *countfreqs.py*, write a function that computes the transition parameters

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

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

- Use the same emission probabilities as in Part 1.

- Using the maximum likelihood estimates for transitions and emissions (as described above), implement the Viterbi algorithm to compute

  $$\arg\max_{y_1 \cdots y_n} p(x_1 \dots x_n, y_1 \dots y_n)$$

  Make sure to replace infrequent words ($\mathit{Count}(x) < 5$) in the training dataset and in the decoding algorithm with a common symbol *_RARE_*. The trigram tagger should have the same basic functionality as the baseline tagger.

- Run the trigram tagger on the development dataset and write the output to a file named *gene.dev.p2.out*. Evaluate it as above. The model should have a total F1-score of 0.40.

- Run the trigram tagger on *gene.test* and write the output to a file named *gene.test.p2.out*. Please hand in this file as part of the assignment.