NLP Programming Assignment 1: Hidden Markov Models

In this assignment, you will build a trigram hidden Markov model to identify gene names in biological text. Under this model the joint probability of a sentence $x_1, x_2, \cdots, x_n$ and a tag sequence $y_1, y_2, \cdots, y_n$ is defined as

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

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.

The files for the assignment are located in the archive `h1-p.zip`. 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`. The labeled files take the format of one word per line with 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 consists of identifying 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 (`http://biocreative.sourceforge.net/biocreative_2.html`).

To help out with the assignment we have provided several utility scripts. Our code is written in Python, but you are free to use any language for your own implementation. Our scripts can be called at the command-line to pre-process the data and to check results.

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 $Count(y \rightsquigarrow x)$, 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.

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.

Part 1 (20 points)

- 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

```
Nations I-ORG
```

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

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

  The expected result should match the result above. When you are ready to submit, run your model on `gene.test` and write the output to `gene_test.p1.out`. Run `python submit.py` to submit.

Part 2 (30 Points)

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

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

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

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

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

  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. When you are ready to submit, evaluate your model on `gene.test` and write the output to `gene_test.p2.out`. Run `python submit.py` to submit.

Part 3 (10 Points)

In lecture we discussed how HMM taggers can be improved by grouping words into informative word classes rather than just into a single class of rare words. For this part you should implement four rare word classes

Numeric The word is rare and contains at least one numeric characters.

All Capitals The word is rare and consists entirely of capitalized letters.

Last Capital The word is rare, not all capitals, and ends with a capital letter.

Rare The word is rare and does not fit in the other classes.

You can implement these by replacing words in the original training data and generating new counts by running `count_freqs.py`. Be sure to also replace words while testing.

The expected total development F1-Score is 42. When you are ready to submit, evaluate your model on `gene.test` and write the output to `gene_test.p3.out`. Run `python submit.py` to submit.