CSCI 544 — Applied Natural Language Processing

Coding Exercise 3

Overview

In this assignment you will write a Hidden Markov Model part-of-speech tagger for Italian, Japanese, and a surprise language. The training data are provided tokenized and tagged; the test data will be provided tokenized, and your tagger will add the tags. The assignment will be graded based on the performance of your tagger, that is how well it performs on unseen test data compared to the performance of a reference tagger.

Data

A set of training and development data will be made available as a compressed ZIP archive. The uncompressed archive will have the following files:

- Two files (one Italian, one Japanese) with tagged training data in the word/TAG format, with words separated by spaces and each sentence on a new line.
- Two files (one Italian, one Japanese) with untagged development data, with words separated by spaces and each sentence on a new line.
- Two files (one Italian, one Japanese) with tagged development data in the word/TAG format, with words separated by spaces and each sentence on a new line, to serve as an answer key.
- A readme/license file (which you won't need for the exercise).

The grading script will train your model on all of the tagged training and development data (separately for Italian and Japanese), and test the model on unseen data in a similar format. The grading script will do the same for the surprise language, for which all of the training, development and test data are unseen.

Programs

You will write two programs: `hmmlearn.py` will learn a hidden Markov model from the training data, and `hmmdecode.py` will use the model to tag new data. If using Python 3,

you will name your programs `hmmlearn3.py` and `hmmdecode3.py`. The learning program will be invoked in the following way:

```
> python hmmlearn.py /path/to/input
```

The argument is a single file containing the training data; the program will learn a hidden Markov model, and write the model parameters to a file called `hmmmodel.txt`. The format of the model is up to you, but it should follow the following guidelines:

1. The model file should contain sufficient information for `hmmdecode.py` to successfully tag new data.
2. The model file should be human-readable, so that model parameters can be easily understood by visual inspection of the file.

The tagging program will be invoked in the following way:

```
> python hmmdecode.py /path/to/input
```

The argument is a single file containing the test data; the program will read the parameters of a hidden Markov model from the file `hmmmodel.txt`, tag each word in the test data, and write the results to a text file called `hmmoutput.txt` in the same format as the training data.

The accuracy of your tagger is determined by a scoring script which compares the output of your tagger to a reference tagged text. Note that the tagged output file `hmmoutput.txt` must match line for line and word for word with the input to `hmmdecode.py`. A discrepancy in the number of lines or in the number of words on corresponding lines could cause the scoring script to fail.

Grading

After the due date, we will train your model (separately for each language) on a combination of the training and development data, run your tagger on new (unseen) test data, and compute the accuracy of your output compared to a reference annotation. Your grade will be the accuracy of your tagger, scaled to the performance of a reference HMM tagger developed by us. Since part-of-speech tagging can achieve high accuracy by using a baseline tagger that just gives the most common tag for each word, only the performance above the baseline will be scaled:

- If your accuracy <= baseline accuracy, your grade is your accuracy.
- If baseline accuracy < your accuracy < reference accuracy, your grade is baseline + (1 – baseline) × (yours – baseline) / (reference – baseline).
- If reference accuracy <= your accuracy, your grade is 100.

For example, if the baseline is 90%, the reference in 95%, and your accuracy is 93%, then your grade will be 0.9 + 0.1 × 0.03 / 0.05 = 96%.

Each language (Italian, Japanese, and the surprise language) is worth one-third of the overall grade for this assignment.

Notes

- **Tags.** Each language has a different tagset; the surprise language will have some tags that do not exist in the Italian and Japanese data. You must therefore build your tag sets from the training data, and not rely on a precompiled list of tags.
- **Slash character.** The slash character '/' is the separator between words and tags, but it also appears within words in the text, so be very careful when separating words from tags. Slashes never appear in the tags, so the separator is always the last slash in the word/tag sequence.
- **Smoothing and unseen words and transitions.** You should implement some method to handle unknown vocabulary and unseen transitions in the test data, otherwise your programs won't work.
    - **Unseen words:** The test data may contain words that have never been encountered in the training data: these will have an emission probability of zero for all tags.
    - **Unseen transitions:** The test data may contain two adjacent unambiguous words (that is, words that can only have one part-of-speech tag), but the transition between these tags was never seen in the training data, so it has a probability of zero; in this case the Viterbi algorithm will have no way to proceed.
- The reference solution will use add-one smoothing on the transition probabilities and no smoothing on the emission probabilities; for unknown tokens in the test data it will ignore the emission probabilities and use the transition probabilities alone. You may use more sophisticated methods which you implement yourselves.
- **End state.** You may choose to implement the algorithm with transitions ending at the last word of a sentence (as in the written homework assignment or in Jurafsky and Martin, figure 8.5), or by adding a special end state after the last word (see for example an older draft of Jurafsky and Martin, figure 9.11). The reference solution will use an end state.
- **Runtime efficiency.** Vocareum imposes a limit on running times, and if a program takes too long, Vocareum will kill the process. Your program therefore needs to run efficiently. **In our experience, using log probabilities contributes to runtime-efficient code.** Run times for the reference solution are approximately 1 second for running `hmmlearn.py` on the training data and 3 seconds for running `hmmdecode.py` on the development data, running on a MacBook Pro from 2016.