CIS5300: Assignment 2

Submission Due: October 4th, 11:59pm

Report page limit: 4 pages

In this assignment, you will implement several approaches for part-of-speech (POS) tagging. Unlike the classification of the previous assignment, POS tagging requires generation of an entire sequence of predicted values, the parts of speech corresponding to each word. To build your POS tagger, you will use data from the Penn Treebank that includes POS tagged sentences. After you have built your POS taggers, run them on the sentences in the test set, and submit your best POS tag predictions to the leaderboard.

Starter Code:
`https://cis.upenn.edu/~myatskar/teaching/cis530_fa22/homework/hmm/starter-code.zip`

Data Download:
`https://cis.upenn.edu/~myatskar/teaching/cis530_fa22/homework/hmm/data.zip`

Leaderboard:
Upload your model predictions to Gradescope

Report Template:
`https://www.overleaf.com/read/fpjxmxxppjrm`

Training, Development and Test Data Download the data listed at the top of the page. This dataset contains almost a million words of text from the Wall Street Journal. The sentences in the dataset are written out word-by-word in a flat, column format in the `*_x.csv` files, where individual documents are separated by `-DOCSTART-` tokens. Each of these words has a corresponding POS tag,[1] located in the `*_y.csv` files. The test POS tags file, `test_y.csv`, is missing — your job is to output a reasonable sequence of tags to recreate this file using the sequences of words in the `test_x.csv` file, and submit your predictions to the leaderboard.

Evaluation Script The starter code contains an evaluation script to help you evaluate on the development set. The script calculates the mean F-score across all classes:

```
python evaluate.py
     --predicted prediction.csv
     --development data/dev_y.csv
```

Some parts of the data may be easier for you to correctly label than others. To identify points where your model is struggling, the evaluation script comes with a confusion matrix built in. You can print out the confusion matrix for the development set by adding the `confusion` flag to the script:

```
python evaluate.py
     --predicted prediction.csv
     --development data/dev_y.csv
     --confusion
```

Tasks Please read all the tasks. Your design of earlier tasks should probably be influenced by what you will be asked to do later. The separation into tasks is intended to help you develop your system step-by-step. We strongly suggest you build the simplest possible version of each component first, and then improve on them, so that you can evaluate incremental improvements to your system.

[1] For a handy tag guide, see `https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html`.

1. *MLE Parameter Estimation.* In this task, you will build a **trigram** tagger with smoothing and a reasonable model for handling unknown words. Initially, you will use a greedy decoder to find a good scoring sequence. We recommend using the greedy decoder you are asked to build below.

   You should do something sensible for unknown words, using a technique like unknown word classes, suffix trees, or a discriminative (e.g., MaxEnt, MLP) model used with Bayes' rule as part of your emission model. The discriminative model is slightly more ambitious, but can work really well!

   You must implement at least two forms of smoothing. We discussed in class add-k smoothing and linear interpolation, but you can choose methods that weren't discussed, such as Good-Turing estimation and Kneser-Ney smoothing.

   Please make sure to clearly discuss what is your approach to handling unknown words, how accurately can you identify tags for words in the development set that are not observed during training, and which smoothing method was most successful.

2. *Implement a State Lattice.* Lattices are directed, acyclic graphs, whose nodes are states in a Markov model and whose arcs are transitions in that model, with weights attached. In this concrete case, states are pairs of tags in a position in the sentence. The arc weights are the scores you estimated above. How does one generate a reasonable sequence of tags given a lattice? One simple approach is to greedily select the best transition at any given point in the path.

   Try experimenting with greedy decoding. What happens if your greedy algorithm follows the k best paths so far instead of just one? (This slightly-less-greedy approach is called *beam search.*)

   Please make sure to discuss the following points:

   - How often does the greedy algorithm ($k = 1$) find the exact solution in your development set? Give examples of when it does not produce a fully correct solution. *Remember that error analysis of this kind is an important part of your report!* Hint: to identify non-exact solutions you can compare the score of the gold sequences to you predicted one. If the score of the gold sequence is higher, what does it mean? You can also do this comparison using your Viterbi implementation (see below).
   - How much does performance improve using beam search with $k = 2$ and $k = 3$? Do you get fewer sub-optimal solutions?

3. *Implement Viterbi.* You may have noticed that the greedy algorithm returns many sub-optimal sequences. Your next task in this assignment is to upgrade the greedy implementation with the Viterbi algorithm. You should observe **zero** sub-optimal solution with a Viterbi decoder.

   Please make sure to discuss how well does Viterbi work in comparison with your greedy algorithm.

4. *Generative Model.* Now do further experiments with the full POS tagger.

   Please make sure to at least discuss:

   - Experiment with bigram and trigram models. Which is more successful? Give examples of errors that each model makes.
   - *Optional:* Try a 4-gram model and compare with bigrams and trigrams.

Expected Results As a target, accuracies of 94+ are good, and 96+ are basically state-of-the-art. Unknown word accuracies of 60+ are reasonable, 80+ are good. *If your results are much lower, you likely have bugs or need to build better models.* The results will be used, along with code reviews and the content of your report, to estimate the correctness of your implementation.

Debugging Tips on debugging sub-optimalities:

- Find the shortest sentence that exhibits a sub-optimality.
- Output the trellis to the screen, along with the state of your Viterbi decoder. Go line-by-line through the output of your algorithm and see where the sub-optimality happens.

- Try to stick as closely as you can to the Viterbi algorithm given in class. First debug to get it right, and only then add fancy optimizations.

Implementation Tips and Tricks

- When populating the state lattice, make sure to just consider feasible edges. Remember, in a tri-gram model most of the edges from one column to the next are not feasible. You don't need to waste time on them.
- Brants (2000; `https://arxiv.org/pdf/cs/0003055.pdf`) details a very effective method to handle unknown words in Section 2.3 (you will have to read the entire paper to get the notation).
- For beam search, you just have to keep track of multiple (k) prefixes and their score. Greedy inference is a sub-case of k=1, so doesn't require any additional code. You can do this computation on the trellis, but then you don't need to populate the entire trellis and just keep pointers to the top-k values in each column. This is not exactly the top-k partial solutions (try to think why), but it should work just the same to solve the approximate $\arg\max$.
- To compute the number of sub-optimalities, a good heuristic is to use your gold sequence. You can easily compute its probability under your model. If it's higher than your predicted solution, it's clear your predicted solution is sub-optimal under your model. This tip appears as a hint on Task #2.
- You can easily represent your trellis as a large NumPy matrix and index directly into each cell. What indexing scheme you can use? Well, map each POS tag (hidden state) into an integer. Then to get into a cell that is indexed by a pair of tags with indices i and j, you can do something like $M[i \times NUM\_TAGS + j, t]$, where t is the time step (e.g., column number) and NUM_TAGS is the total number of tags. This is super fast and very memory efficient. You can maintain two such matrices, one for the π values and one for the back pointers.
- If you get a zero count in the denominator in the your MLE estimate, you can add a small epsilon (a very small value) to make sure there is no division by zero. Remember to make sure your distribution is normalized.
- Before you start implementing Viterbi, we recommend you review class materials that explain the algorithm. Your textbooks are a good source, for example Chapter 8 in Jurafsky and Martin, Chapter 7 in Eisenstein, or Michael Collins' notes on HMMs are especially good, and explicitly detail the trigram HMM case, `http://www.cs.columbia.edu/~mcollins/courses/nlp2011/notes/hmms.pdf`

Leaderboard Instructions Use your model to predict on the test set and create `test_y.csv`. Upload to Gradescope leaderboard, and we will run the evaluation script on your results, compare to the test results, and display your F1 score of your predictions per-token on the leaderboard.

Grading Your grade will be split between the report and code checks (80%) and the final performance of your tagger on the leaderboard (20%). We will look at code if something appears wrong and deduct appropriating. The grading of your test performance on the leaderboard will be computed as $\min(20, \frac{\max(0, f-0.87)}{0.96-0.87} \times 20)$, where f is the test F1 score from the leaderboard.

Student State of the Art The highest performance on this assignment has been 96.5.

Rubric

- Parameter Estimation (10%)
- Unknown Words (5%)
- Smoothing (2.5% per method = 5%)
- Sentence/Tag Sequence Probability Estimation (15%)

- Decoding Methods
  - Greedy (5% for bi-gram + 5% for tri-gram = 15%)
  - Beam-K (7.5% for bigram + 7.5% for trigram = 15%)
  - Viterbi (5% for bi-gram + 5% for tri-gram = 10%)
- Analysis (Report) (10%)
- Leaderboard Score (20%)
- Extra Credit
  - 4-gram model (10%)

Development Environment and Third-party Tools All allowed third-party tools are specified in the `requirements.txt` file in the assignment starter code.[2] The goal of the assignment is to gain experience with specific methods, and therefore using third-party tools and frameworks beyond these specified is not allowed. You may only `import` packages that are specified in the `requirements.txt` file or that come with Python's Standard Library. The version of Python allowed for use is 3.8.x. Do not use older or newer version. We strongly recommend working within a fresh virtual environment for each assignment. For example, you can create a virtual environment using conda and install the required packages:

```
conda create -n cis5302 python=3.8
conda activate cis5302
pip install -r requirements.txt
```

Leaderboard

Submission, Grading, and Writeup Guidelines Your submission on Gradescope is a writeup in PDF format. **The writeup must follow the template provided. Please replace the TODOs with your content. Do not modify, add, or remove section, subsection, and paragraph headers. Do not modify the spacing and margins.** The writeup must include at the top of the first page: the names of the students. The writeup page limit is **4 pages**. We will ignore any page beyond the page limit in your PDF (do not add a cover page).

The following factors will be considered: your technical solution, your development and learning methodology, and the quality of your code. This assignment includes a leaderboard and we will also consider your performance on the leaderboard. Our main focus in grading is the quality of your empirical work and implementation. We value solid empirical work, well written reports, and well documented implementations. Of course, we do consider your performance as well. The assignment details how a portion of your grade is calculated based on your empirical performance.

In your write-up, be sure to describe your approach and choices you made. Back all your analysis and claims with empirical development results, and use the test results only for the final evaluation numbers. It is sometimes useful to mention code choices or even snippets in write-ups — feel free to do so if appropriate, but this is not necessary.

Some general guidelines to consider when writing your report and submission:

- Your code must be in a runnable form. We must be able to run your code from vanilla Python command line interpreter. You may assume the allowed libraries are installed. Make sure to document your code properly.
- Your submitted code must include a `README.md` file with execution instructions.
- Please use tables and plots to report results. If you embed plots, make sure they are high resolution so we can zoom in and see the details. However, they must be readable to the naked eye (i.e., without zooming in). Specify exactly what the numbers and axes mean (e.g., F1, precision, etc).

[2] `https://realpython.com/lessons/using-requirement-files/`

- It should be made clear what data is used for each result computed.
- Please support all your claims with empirical results.
- All the analysis must be performed on the development data. It is OK to use tuning data. Only the final results of your best models should be reported on the test data.
- All features and key decisions must be ablated and analyzed.
- All analysis must be accompanied with examples and error analysis.
- Major parameters must include sensitivity analysis. Plots are a great way to present sensitivity analysis for numerical hyper parameters, but tables sometimes work better. Think of the best way to present your data.
- If you are asked to experiment with multiple models and multiple tasks, you must experiment and report on all combinations. It should be clear what results come from which model and task.
- Clearly specify what are the conclusions from your experiments. This includes what can be learned about the tasks, models, data, and algorithms.
- Make figures clear in isolation using informative captions.

This assignment was adapted from Dan Klein and Yoav Artzi.

Updated on September 15, 2022