Advanced Methods in NLP Lecturer: Jonathan Berant

Home Assignment 3: Tagging

Due Date: *May 8, 2018*

In this home assignment we will implement POS taggers. Please copy the data and supporting code from `~omrikosh/advanced_nlp/assignment3/code`. **Note that the Penn Treebank dataset is licensed! It is illegal to make it public in any way!**

Your code should run properly on Python 2.7 on linux (it doesn't have to run on Python 3 and above). It must run on `nova.cs.tau.ac.il` using `/usr/bin/python`.

To submit your solution, create a directory at `~omrikosh/advanced_nlp/assignment3/submissions/<id1>_<id2>` (where id1 refers to the ID of the first student) and put all relevant files in this directory. The submission directory should include the code necessary for running the tests provided out-of-the-box, as well as a written solution, and a text file including an e-mail of one of the students. Fill the submitters info also in `submitters_details.py`. Don't copy the dataset to your directory! Instead, add a soft link to the data from your submission directory:

```
cd <your submission directory>
ln -s ~/advanced_nlp/assignment3/code/data/Penn_Treebank/./
```

1 Data preprocessing

(a) As we saw in class, a common solution to the rare words problem is to pre-process the data and replace rare words with a category, or signature (e.g., numbers, dates, capitalization, prefixes, suffixes, etc. ...). Come up with good word categories/sigatures and implement `replace_word` in `data.py`. The following paper, where this was done for named entity recognition, can be helpful: `http://people.csail.mit.edu/mcollins/6864/slides/bikel.pdf`. You can test the efficacy of your implementation by evaluating your "most frequent tag" baseline (next problem).

2 Most frequent tag baseline

(a) The most frequent tag baseline tags each word with its most frequent tag, as seen in the training set. Implement the most frequent tag baseline in `most_frequent.py`.

(b) Implement the evaluation procedure in `most_frequent.py` that measures the accuracy of the most frequent tag baseline on some dataset. Evaluate your tagger against the development set. What is your accuracy on the development set?

3 HMM tagger

(a) **MLE estimators:** Use the training data to estimate the transition probabilities q and emission probabilities e. Fill the implementation of the training algorithm in the function `hmm_train.py` in `hmm.py`.

(b) **Viterbi:** Implement the Viterbi algorithm (as described in slide 48 in Tagging presentation) in `hmm_viterbi` function in `hmm.py`. The algorithm receives a sentence to tag as input, the counts computed by the training procedure. The algorithm returns the highest probability sequence of tags according to q and e. Recall that the estimates for q should be based on a weighted linear interpolation of $p(t_i|t_{i-1}, t_{i-2})$, $p(t_i|t_{i-1})$ and $p(t_i)$. Tune the hyper-parameters on the development set, and document the optimal λ_i values in your written solution.

Note: a straight-forward implementation of the Viterbi algorithm can be slow, so you should add some tag pruning (eliminating some tags for specific words). Training and evaluation (on dev set) should take up to few minutes. Document your pruning policy in your written solution.

(c) Implement the evaluation procedure `hmm_eval` in `hmm.py` that measures the accuracy of the HMM tagger with Viterbi algorithm on some dataset. What is your accuracy on the development set?

4 Maximum Entropy Markov Model (MEMM) tagger

In this part you will implement the MEMM tagger (a locally-normalized log-linear model). The learning part is already given in the skeleton code using `scikit-learn`, but you can use any learning package you are comfortable with that implements multi-class logistic regression.

(a) **Feature engineering:** Implement features for your model. You should implement the features from Ratnaparkhi (1996) mentioned in class (`nlp_loglinear.pdf` file slide 51), but you can add more feature templates if you want.

(b) **Greedy inference:** Implement a greedy inference algorithm, where you tag a sentence from left to right with your trained model. If you choose to use `scikit-learn`, you can use the function `logreg.predict(·)`. Fill your implementation in the function `memm_greedy` in `memm.py`.

(c) **Viterbi:** Implement the Viterbi algorithm for MEMMs. The tag distribution should be inferred from the trained model. If you chose to use the `sklearn` solver, you can use the function `logreg.predict_proba`. Fill your implementation in the function `memm_viterbi` in `memm.py`.

Note: prediction in this model is likely to be much slower than in the HMM model. You should consider optimizing your implementation by caching predictions, avoiding unnecessary feature extraction, etc. Training and evaluation (on dev set) should

take up to 6 hours. Document any optimization you have performed in the written solution.

(d) Implement the evaluation procedure in `memm.py`, that measures the accuracy of the MEMM tagger with Viterbi/greedy inference on some dataset. What is your accuracy on the development set?

(e) Sample errors from your best model and analyze them. What are common failure cases for your model. Where does it struggle? Summarize the results of your analysis in the written solution.