CSC5031Z – Natural Language Processing

Assignment 1

Scott Hallauer (HLLSCO001) and Steve Wang (WNGSHU003)

Implementation

Data Processing

This application uses two datasets for the Afrikaans language: (1) a training dataset containing 2,613 sentences, and (2) a testing dataset containing 329 sentences. The pre-processing of these datasets is handled by **dataset.py**. Datasets are read into a list of dictionaries, each representing a sentence containing words and tags. Subsets of these datasets can easily be obtained (using the `subset` function) for splitting data into training and development sets (for linear interpolation optimisation). The `prepare` function replaces all words that only occur once in the dataset with the <UNK> word token.

After pre-processing, **statistics.py** is used to calculate the transition (q) and emission (e) distributions for the training data. These are calculated using the unigram and bigram tag counts, as well as the word-tag pair counts returned by the `get_counts` function. Two types of smoothing have been implemented: (1) Laplace smoothing, and (2) linear interpolation smoothing. Laplace smoothing can be used for both distributions, whereas linear interpolation can only be used for the transition distribution. For the second-order HMM, the trigram counts are also determined to calculate the trigram transition distribution.

Hidden Markov Model Tagger

The first-order HMM POS tagger is implemented in **tagger_1.py**. For each sentence in the test set, a lattice is created. The lattice is the list of words with each corresponding to a set of POS tags. Each POS tag has a pi value and a back pointer (see Table 2). The `populate_lattice` function populates the lattice for each word and calculates the pi values and back pointers. The pi value of the first word in a sentence is calculated using the formula: pi = q("START", pos_tag) × e(word, pos_tag). Other words are calculated using the formula: pi = pi(prev_tag) × q(prev_tag, pos_tag) × e (word, pos_tag). The back pointer would be the previous_tag that produces the maximum pi value.

Once the lattice is populated, use `get_pos_tag` function to find the most probable sequence of part of speech tags for a given sentence. This is done by finding the max pi value for the POS tag of the last word and following the back pointers. This sequence of tags is the reverse so that each tag corresponds to the words in the sentence in order.

The second-order HMM tagger is implemented in **tagger_2.py.** A similar algorithm applies with a pi value calculation that uses trigram distribution: pi = pi(prev_tag) * trigram_dist(pre_prev_tag, prev_tag, pos_tag) * e (word, pos_tag)

Discussion

As shown in table 1 of the appendix, both applying the linear interpolation smoothing or using second-order HMM improved the part of speech tagging accuracy. Second-order HMM reduces the performance significantly because it uses a 3 dimensional q matrix. In general, the tags that it produces has a shorter sequence of incorrect tags (i.e: It recovers quicker from an upset in the pos tag sequence) This is evident by comparing the **tagger_1.out** with **tagger_2.out.** The use of linear interpolation requires more time in the training phase to find the optimal $\boldsymbol{\lambda_1}$ and $\boldsymbol{\lambda_2}$ values. Performance in finding the most likely sequence of POS tags would be the same as First-order HHM with laplace smoothing.

Furthermore, the use of <UNK> tag for all words that occurred only once has reduced the data size significantly and hence improved the performance.

Appendix

Table 1: Part-of-speech tagging accuracy on test set for first- and second-order HMMs using either Laplace or linear interpolation smoothing.

Smoothing	**First-Order HMM**	**Second-Order HMM**
Laplace	0.7234929954552283	0.8176663811544984
Linear Interpolation	0.8310579661048351	–

Table 2: Example of a lattice used by HMM to track pi values and back pointers.

Word 1	{POS_1:[pi_value , back_pointer], POS_2:[pi_value , back_pointer], ….}
Word 2	{POS_1:[pi_value , back_pointer], POS_2:[pi_value , back_pointer], ….}
Word 3	{POS_1:[pi_value , back_pointer], POS_2:[pi_value , back_pointer], ….}
...	...

Table 3: Results of lambda parameter optimisation for linear interpolation smoothing, with the highest average accuracy highlighted in green. Average accuracy is taken from ten training iterations, each using a different tenth of the training data as the development set.

λ_1	λ_2	**Average Accuracy**
0.00	1.00	0.8137457109593038
0.05	0.95	0.815108544210106
0.10	0.90	0.8185459826116185
0.15	**0.85**	**0.8204146700663142**
0.20	0.80	0.8174945238645451
0.25	0.75	0.8131427157477876
0.30	0.70	0.8074653366108387
0.35	0.65	0.7988384027783819
0.40	0.60	0.7888595798295455
0.45	0.55	0.7794094111924816
0.50	0.50	0.7660785356315876
0.55	0.45	0.7548223696184055
0.60	0.40	0.7396016069279808

0.65	0.35	0.7194746413236286
0.70	0.30	0.6975746866449024
0.75	0.25	0.6694676542490774
0.80	0.20	0.6467814465113182
0.85	0.15	0.6175829597356206
0.90	0.10	0.5935514792870287
0.95	0.05	0.5700539512677724