Timothy Kim - W01011895
HW #3 - CSCI 404 @11AM
February 28, 2015
POS Tagging with HMM

Part 1 - Datasets

For this assignments implementation of a Hidden Markov Model for Bigram Part-of-Speech Tagging we are working with 2 different groups of data sets. First is the **ic** or ice cream cone data set. The files in this set includes: *ictrain, ictest, ic2train,* and *ic2test*. Theses .txt files contain an ice cream cone sequence followed by a 1-character tag, either **C** or **H**. This data set is very small and used for testing purposes. The second and main data set is the **en** or English words data set. Theses files include: *entrain, entest, enraw, ...* which are standard .txt documents. All but *enraw* have a sequence of an english word followed by its 1 character POS tag. The tags are a simplified version of the Penn Treebank tag set which can be found in the *hw3.pdf* documentation.

Parts 2 & 3 - Viterbi decoder on ic/en data

Running a Viterbi tagger allows one to find the single best path through an HMM. In our ice cream example, the single most likely weather sequence given amount of ice cream eaten. Implementation of **vtag.py** (which all running code is located) mainly focuses on running on the **en** dataset but was built first testing on the smaller and easier **ic** dataset. See included **vtag.py** source code for full implementation details. When run by:

```
vtag.py ictrain.txt ictest.txt
```

It outputs to the screen a summary of its performance on the test data. The format is a line that, with the variables replaced by the actual values, reads:

```
Tagging accuracy (Viterbi decoding): o% (known: k% novel: n%)
```

The first step in the program is to take the training data and obtain all the necessary counts and tables to properly train our model. It does this by creating 5 unique dictionaries data structures.

```
tag_dict # key: word, value: tag
tag_count # key: tag, value: count
tagtag_count # key: tag2tag3, value: count
word_count # key: word, value: count
wordtag_count # key: 'word/tag', value: count
```

tag_dict stores all possible tags for each word. This allows us to only consider tags that are allowed for the particular word.
tag_count keeps track of the amount of times a 'tag' appears. Also useful for obtaining a list of all tags.

tagtag_count keeps track of amount of times a sequence of tag_{i-1}, tag_i appear in the training set.
word_count keeps track of amount of times a 'word' appears in data.
wordtag_count keeps track the amount of times the specific 'word' has the specific 'tag'

Population of these data structures is done with one pass through the first command line argument (e.g. ictrain.txt). Each file has a format of 1 'word/tag' per line. So the file is iterated line by line, updating all relevant variables as needed. Next, the testing data is read into memory. Taking the second command line argument (e.g. ictest.txt), the file is taken line by line and 2 new lists are created. One is a string of all the words in order and the other is its tags, again in order. The *test_tags* list is needed so we can compute the accuracy of our best possible with the actual sequence of tags.

Before the Viterbi tagging algorithm could properly function we first needed 2 additional functions to calculate probabilities:

ttMLE(tag3, tag2) calculates the probability of P(tag3|tag2) using Maximum Likelihood Estimate and add-one smoothing.
wtMLE(word, tag) is similar but instead calculates probability of P(word|tag)

These functions make use of the tables formed earlier to compute the probabilities on demand as needed. The functions make use of the equation:

$$ttMLE : P(t_i|t_{i-1}) = \frac{C(t_{i-1},t_i) + 1}{C(t_{i-1}) + V} \tag{1}$$

$$wtMLE : P(w_i|t_i) = \frac{C(w_i,t_i) + 1}{C(t_i) + V} \tag{2}$$

Where C is the count from the dictionaries, and V is the vocabulary size. For ice cream data, the vocab size is number of unique words in test (since all possible words and tags were observed in training file). The vocab size differs depending on working on the **ic** or **en** dataset so the functions are altered to work on the **en** dataset as outlined later on.

Now for implementing the Viterbi tagging algorithm, pseudo-code from the specifications pdf and class notes were used. Two lists are created to keep track of data as we loop through the test file.

viterbi = [] stores dictionaries that map each tag to the probability of best tag sequence.
backpointer = [] is similar but maps each tag to the previous tag in the best tag sequence.

The initialization step then begins by taking care of the first entry in viterbi manually. This is done so the beginning '###' tag is ignored and only other available tags are checked. Now the main recursive step is performed. Using nested *for* loops, we iterate over all the remaining words in the test string and all the available tags for the current word. If a test string word is novel (not observed in training), then we allow any tag to be possible except the '###' tag. With each iteration a 2 new dictionaries are created (this_viterbi, this_backpointer) which are appended to the viterbi/backpointer list mentioned earlier. For each tag we determine what the best previous tag is, as well as the probability of the best tag sequence ending with the current tag. Calling the probability functions stated above, we find the best previous tag by adding 3 probabilities. We add instead of multiply because when taking the probability we convert each to a natural log-probabilities to prevent underflow. For readability this can look something like:

```
best_previous = None
   best_prob = 0.0
   for prevtag in prev_viterbi.keys():
      prob = prev_viterbi[prevtag] + ttMLE(tag, prevtag)
             + wtMLE(test_string[i], tag)
      if prob >= best_prob:
         best_previous= prevtag
         best_prob = prob
```

Now in the termination step we append the appropriate values to the *viterbi* and *backpointer* data structures before moving on to the next iteration. Similarly to what we did to the first '###' tag we do the same to the last manually. We can now go through our created list of backpointers obtain the best tag sequence for our test string/file. Finally with the best tag sequence we can calculate the accuracy of our test model. Three percentages are calculated: overall, known-word, and novel-word accuracy. Overall considers all word tokens except '###', known considers only tokens of workds that also appear in train, and novel considers only tokens that did not appear in train.1 Incrementing through the tag sequence, we compare to the test tag sequence we stored earlier while parsing. Simple counting and arithmetic is used to divide the respective values with the adjusted totals.

The implementation of the Viterbi tagging algorithm for the **en** dataset is basically the same as the **ic** dataset. Only change is the vocabulary size, V, used in the probability functions. For **en** datasets, the vocabulary size is determined by the number of word types observed in the Union of the **entrain.txt** file and the **enraw.txt**. So in order for the Python program to complete without error, the appropriate training and test files must be availible as well as the **enraw.txt** file (as program is set up to test the english dataset not ice cream).

Results

When run with: **vtag.py ictrain.txt ictest.txt** outputs:

```
Tagging accuracy (Viterbi decoding): 90.91% (known: 90.91%
                                             novel: N/A))
```

When run with: **vtag.py entrain.txt entest.txt** outputs:

```
Tagging accuracy (Viterbi decoding): 92.34% (known: 96.05%
                                             novel: 53.89%)
```

For the ice cream data, the expected accuracy matches what the documentation provides so we known it is working correctly. For the english data set the end results were a bit unexpected. According to the documentation, the expected results should be:

```
Tagging accuracy (Viterbi decoding): 91.84% (known: 96.60%
                                             novel: 42.55%)
```

This means our results are 0.50% more accurate in overall, 0.55% less accuracte for known words, and 11.34% more accurate for novel words. The difference for the first two values could simply be attributed to rounding, log converting, or another simple logical error. However the novel word difference makes me believe there is some misinterpretation going on in my implementation. I was unable to locate where I was messing up in my program.

Part 4 - Improvement of Smoothing

Now that we have our basic add-one (Laplace) smoothing accuracy, we can improve our smoothing technique to increase performance. The new method of smoothing implemented is called the "one-count smoothing" method. Before we can start calculating new probabilities we need to define a couple new functions. Two of the new functions give us backoff probability estimates. As defined in the given specification:

$$P_{tt-backoff}(t_i|t_{i-1}) = \frac{c(t_i)}{n} \tag{3}$$

$$P_{wt-backoff}(w_i|t_i) = \frac{c(w_i)+1}{n+V} \tag{4}$$

These are implemented in the **vtag1c.py** source code as *ttBackoff* and *wtBackoff*. This Python program is the same as vtag.py except with the smoothing improvements. Now we needed 2 more dictionaries to keep track of singletons in the training file. A singleton was when there is only 1 of a specific tag,tag sequence or word/tag combination. In order to store the count of singletons for each tag, we create two dictionary data structures call *wtsing_count* and *ttsing_count*. When these are given a tag key value, it returns the number of

singletons for that tag. The values are obtained in our initial parse of the training data. Adding one on new entries and subtracting when another is found. Now with all the parts needed we can edit the original *ttMLE* & *wtMLE* functions. Again provided by the documentation we model our function after the equations:

$$P_{tt}(t_i|t_{i-1}) = \frac{c(t_{i-1}, t_i) + \lambda \cdot P_{tt-backoff}(t_i|t_{i-1})}{c(t_{i-1}) + \lambda} where \lambda = 1 + ttsing[t_{i-1}] \quad (5)$$

$$P_{wt}(w_i|t_i) = \frac{c(w_i, t_i) + \lambda \cdot P_{wt-backoff}(w_i|t_i)}{c(t_i) + \lambda} where \lambda = 1 + wtsing[t_i] \quad (6)$$

In this implementation we return normal probabilities from the backoff functions but return log-probabilities from the actual ttMLE and wtMLE functions to properly compute running totals.

Results

From earlier we know when ran with: **vtag.py ictrain.txt ictest.txt** outputs:

```
Tagging accuracy (Viterbi decoding): 90.91% (known: 90.91%
                                             novel: N/A)
```

With the one-count smoothing: **vtag1c.py ictrain.txt ictest.txt** we get:

```
Tagging accuracy (Viterbi decoding): 90.91% (known: 90.91%
                                             novel: N/A)
```

Which coincides with what we expect as the ic dataset has no singletons or novel words so they should be the same.

When run with: **vtag1c.py ic2train.txt ic2test.txt** outputs:

```
Tagging accuracy (Viterbi decoding): 90.91% (known: 90.32%
                                             novel: 100.00%)
```

Which matches the documentation as the correct values.

When run with: **vtag1c.py entrain.txt entest.txt** outputs:

```
Tagging accuracy (Viterbi decoding): 94.14% (known: 96.86%
                                             novel: 65.89%)
```

On the english dataset our overall accuracy increases from 92.34% $\Rightarrow$ 94.14%, a net +1.8. Known-word accuracy increases 96.05% $\Rightarrow$ 96.86% or net +0.81. Finally our novel-word accuracy goes from 53.89% $\Rightarrow$ 65.89% or a net +12.0 increase. Using the one-count smoothing method effectively increased the accuracy of our tagger across the board with the biggest improvement being in the novel-word accuracy.

Screenshots

Figure 1: ice cream 2 dataset

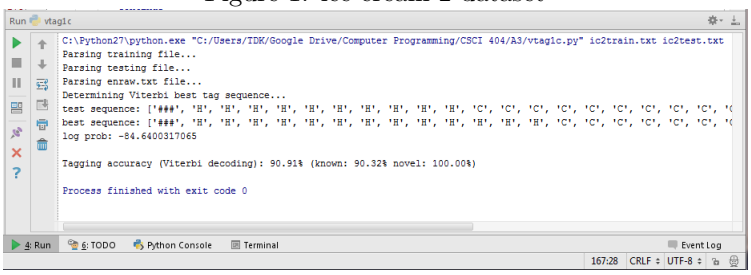
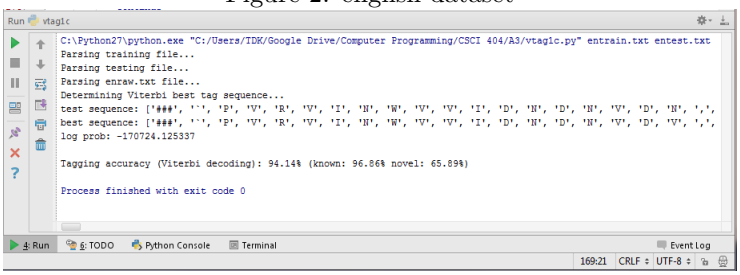


Figure 2: english dataset



Comments

- Included sources in .zip: vtag.py, vtag1c.py, report.pdf
- Initial run of either program may take a couple minutes
- Reference: hw3.pdf
- Reference: 4-pos-tagging.pdf
- Reference: one-count-smoothing.pdf
- Reference: http://en.wikipedia.org/wiki/Viterbi_algorithm
- Reference: http://www.katrinerk.com