cs769, Spring 2011: Problem Set 2

Total points: 100

Due date: Friday, 4th of March

Submit write-up and code via handin: `handin -c cs769-1 -a hw2 -d <DIRECTORY>`. If submitting a late assignment, use `-a hw1-lateN` where `N` is the number of days late.

Late policy: 0–24 hours late ⇒ –10%. 24–48 hours late ⇒ –30%. 48–72 hours late ⇒ –50%. Thereafter no credit. You may discuss problems at a high level with classmates, but you must solve and code them yourselves.

Question 1 (50 points)

For this problem we will consider the supervised prediction of part-of-speech tags. You will be implementing a maximum likelihood estimator (MLE) and viterbi decoder for a bigram HMM. Use `orwell-train2.txt` as your training data (used to estimate the emission and transition parameters) and use `orwell-test2.txt` as testing data. Train the model parameters using the following formulas:

$$P(t|t') = \frac{count(t', t)}{count(t')}$$

$$P(w|t) = \frac{count(t, w) + \delta}{count(t) + |V| \cdot \delta}$$

Predict the most likely tag sequence on each sentence in the test data, using the Viterbi decoding algorithm (lecture 15, page 4 for details). Report the average per-word tag accuracy on the test-set when training with the first 1000, 2000, 3000, 4000, 5000 sentences, respectively, and finally with all sentences. Plot the resulting "learning curve."

Implementation Notes

- Only use smoothing for emission probabilities. The size of the vocabulary: $|V| = 9712$ and use $\delta = 0.001$.
- Hint: When training on the first 3,000 sentences, your test-set accuracy should be 92.035%.
- Don't forget that the first tag of each sentence always transitions from a special begin tag "START" and the final tag always transitions to a special end tag "END." When calculating accuracy do not include these tags (since they are always known).
- It's probably a good idea to implement viterbi in the log-domain to avoid underflow.
- My python implementation is 98 lines of code.

Question 2 (50 points)

For this problem we will consider the *unsupervised* prediction of part-of-speech tags. In this case we will only use the file `orwell-train2.txt` (since our scenario is unsupervised, we do not have separate training and testing sets). Please implement the bigram version of the Gibbs sampling algorithm for the

Bayesian HMM. We will assume that the set of allowable tags for each word in the corpus is known ahead of time. In practice, for each word w, simply collect the set of tags T_w observed with word w in `orwell-train2.txt`. Gibbs sampling will proceed as follows:

Initialization: For each position in the corpus i, randomly assign a tag $t_i \in T_{w_i}$
Sampling: Iterate through the corpus some number of times:

- For each position in the corpus i:
  - Resample $t_i \sim P(t_i|\mathbf{t}_{-i}, \mathbf{w}, \alpha, \beta)$, from the set of allowable tags T_{w_i}.

Recall from lecture that:

$$P(t_i|\mathbf{t}_{-i}, \mathbf{w}, \alpha, \beta) \approx \frac{p_1 \cdot p_2 \cdot p_3}{Z}$$

$$p_1 = \frac{count(t_i, w_i) + \beta}{count(t_i) + |W_{t_i}| \cdot \beta}$$

$$p_2 = \frac{count(t_{i-1}, t_i) + \alpha}{count(t_{i-1}) + |T| \cdot \alpha}$$

$$p_3 = \frac{count(t_i, t_{i+1}) + \alpha}{count(t_i) + |T| \cdot \alpha}$$

where $|W_{t_i}|$ is the total number of words that are allowed to take tag t, $|T|$ is the total number of tags (12 for this data-set), and the counts range over the current sample excluding the i^{th} tag and word. (Note that we are ignoring the indicator variables $I(t_{i-1} = t_i ...)$ from lecture, as these do not make much difference).

For each tag position i, keep a count over all the sample values of t_i. After each iteration through the corpus, output the tag accuracy of the current sample, as well as the accuracy given by selecting the most frequently sampled tag per position. Graph both of these results over 35 rounds. What happens to the sample accuracy after the first few rounds? What about the most frequently sampled tag accuracy? Why does the latter perform better than the former? Why does this unsupervised method perform better than the supervised method in problem 1? (Hint: It really doesn't, but is an artifact of the way we set up the two problems).

Implementation Notes

- Here is my python code for sampling from (a possibly unnormalized) list of probabilities:

```
def sample(probs):
    prob_sum = sum(probs)
    rand = random.random() * prob_sum
    cumulative = 0.0
    for (i,p) in enumerate(probs):
        cumulative += p
        if rand < cumulative:
            return i
```

- Use $\alpha = 0.001, \beta = 1.0$

- The initial random accuracy should be about 90%.
- After one round the sample accuracy should be about 94%.
- My implementation is 115 lines of code in python.