COMS W4705, Fall 2017 : Problem Set 1

- Total points: 100
- Due date: Analytical problems (questions 1–3) due Monday 25th September, 5pm. Programming problems due Monday 2nd October, 5pm.
- Late policy: 5 points off for every day late, 0 points if handed in after 5pm on 28th Sept. for the analytical problems, Oct. 5th for the programming problems.

Analytical Problems

Question 1 (15 points)

In lecture we saw a method for language modeling called *linear interpolation*, where the trigram estimate $q(w_i \mid w_{i-2}, w_{i-1})$ is defined as

$$q(w_i \mid w_{i-2}, w_{i-1}) = \lambda_1 \times q_{ML}(w_i \mid w_{i-2}, w_{i-1}) + \lambda_2 \times q_{ML}(w_i \mid w_{i-1}) + \lambda_3 \times q_{ML}(w_i)$$

Here $\lambda_1, \lambda_2, \lambda_3$ are weights for the trigram, bigram, and unigram estimates, and q_{ML} stands for the maximum-likelihood estimate.

One way to optimize the λ values (again, as seen in lecture), is to use a set of validation data, in the following way. Say the validation data consists of n sentences, $S_1, S_2, \ldots, S_n$. Define $\mathrm{c}'(w_1, w_2, w_3)$ to be the number of times the trigram w_1, w_2, w_3 is seen in the validation sentences. Then the λ values are chosen to maximize the following function:

$$L(\lambda_1, \lambda_2, \lambda_3) = \sum_{w_1, w_2, w_3} \mathrm{c}'(w_1, w_2, w_3) \log q(w_3, |w_1, w_2)$$

Question: show that choosing λ values that maximize $L(\lambda_1, \lambda_2, \lambda_3)$ is equivalent to choosing λ values that *minimize the perplexity* of the language model on the validation data.

Question 2 (10 points)

In the lecture we saw an improved method for linear interpolation where the λ values are sensitive to the number of times the bigram (w_{i-2}, w_{i-1}) has been seen; the intuition behind this was that the more frequently this bigram has been seen, the more weight should be put on the trigram estimate.

Here we'll define a method that is similar in form to the method seen in lecture, but differs in some important ways. First, we define a function $\Phi(w_{i-2}, w_{i-1}, w_i)$ which maps trigrams into "bins", depending on their count. For example, we can define Φ as follows:

$$\begin{array}{lll}
\Phi(w_{i-2}, w_{i-1}, w_i) = 1 & \text{If} & Count(w_{i-2}, w_{i-1}, w_i) = 0 \\
\Phi(w_{i-2}, w_{i-1}, w_i) = 2 & \text{If} & 1 \leq Count(w_{i-2}, w_{i-1}, w_i) \leq 2 \\
\Phi(w_{i-2}, w_{i-1}, w_i) = 3 & \text{If} & 3 \leq Count(w_{i-2}, w_{i-1}, w_i) \leq 5 \\
\Phi(w_{i-2}, w_{i-1}, w_i) = 4 & \text{If} & 6 \leq Count(w_{i-2}, w_{i-1}, w_i)
\end{array}$$

The reason is that the probability distribution is invalid since lambdas for each bucket don't sum to 1. You can refer to the derivation for bigram buckets in language modelling notes; if we follow the same derivation for trigram, we notice that that the bucketing depends upon w and hence it cannot be taken out of the summation. This does not guarantee lambdas for each bucket to sum to 1.

$$\sum_{\underline{w\in\mathcal{V}'}} q(w \mid \underline{u}, \underline{v})$$

$$= \sum_{w\in\mathcal{V}'} [\lambda_1 \times q_{\mathsf{ML}}(w \mid u, v) + \lambda_2 \times q_{\mathsf{ML}}(w \mid v) + \lambda_3 \times q_{\mathsf{ML}}(w)]$$

$$= \lambda_1 \sum_w q_{\mathsf{ML}}(w \mid u, v) + \lambda_2 \sum_w q_{\mathsf{ML}}(w \mid v) + \lambda_3 \sum_w q_{\mathsf{ML}}(w)$$

$$= \lambda_1 + \lambda_2 + \lambda_3$$

$$\boxed{= 1} \checkmark$$

The trigram estimate $q(w_i \mid w_{i-2}, w_{i-1})$ is then defined as

$$\begin{aligned} q(w_i \mid w_{i-2}, w_{i-1}) &= \lambda_1^{\Phi(w_{i-2},w_{i-1},w_i)} \times q_{ML}(w_i \mid w_{i-2}, w_{i-1}) \\ &\quad + \lambda_2^{\Phi(w_{i-2},w_{i-1},w_i)} \times q_{ML}(w_i \mid w_{i-1}) \\ &\quad + \lambda_3^{\Phi(w_{i-2},w_{i-1},w_i)} \times q_{ML}(w_i) \end{aligned}$$

Notice that we now have 12 smoothing parameters, i.e., λ_j^i for $i = 1 \ldots 4$ and $j = 1 \ldots 3$.

Question: Unfortunately this estimation method has a serious problem: what is it?

Question 3 (15 points)

We are going to come up with a modified version of the Viterbi algorithm for trigram taggers. Assume that the input to the Viterbi algorithm is a word sequence $x_1 \ldots x_n$. For each word in the vocabulary, we have a *tag dictionary* $T(x)$ that lists the tags y such that $e(x|y) > 0$. Take K to be a constant such that $|T(x)| \leq K$ for all x. Give pseudo-code for a version of the Viterbi algorithm that runs in $O(nK^3)$ time where n is the length of the input sentence.

Programming Problems

Please read the submission instructions, policy and hints on the course webpage.

In this programming problem, you are going to build a trigram HMM tagger for named entities. The joint probability of a sentence $x_1, x_2, \cdots, x_n$ and a tag sequence $y_1, y_2, \cdots, y_n$ is defined as

$$p(x_1 \cdots x_n, y_1 \cdots y_n) =$$

$$q(y_1|*, *) \cdot q(y_2|y_1, *) \cdot q(\texttt{STOP}|y_{n-1}, y_{n-2}) \cdot \prod_{i=3}^{n} q(y_i|y_{i-1}, y_{i-2}) \cdot \prod_{i=1}^{n} e(x_1|y_1)$$

$\star$ is a padding symbol that indicates the beginning of a sentence, `STOP` is a special HMM state indicating the end of a sentence.

We provide a training data set `ner_train.dat` and a development set `ner_dev.key`. Both files are in the following format (one word per line, word and token separated by space, a single blank line separates sentences.)

```
U.N. I-ORG
official O
Ekeus I-PER
heads O
for O
Baghdad I-LOC
. O
```

```
Senior O
United I-ORG
Nations I-ORG
arms O
official O
```

⋮

There are four types of named entities: person names (PER), organizations (ORG), locations (LOC) and miscellaneous names (MISC). Named entity tags have the format I-TYPE. Only if two phrases of the same type immediately follow each other, the first word of the second phrase will have tag B-TYPE to show that it starts a new entity. Words marked with O are not part of a named entity.

The script `count_freqs.py` reads in a training file and produces trigram, bigram and emission counts. Run the script on the training data and pipe the output into some file:

```
python count_freqs.py ner_train.dat > ner.counts
```

Each line in the output contains the count for one event. There are two types of counts:

- Lines where the second token is `WORDTAG` contain emission counts $Count(y \rightsquigarrow x)$, for example

  ```
  13 WORDTAG I-ORG University
  ```

  indicates that `University` was tagged 13 times as `I-ORG` in the the training data.

- Lines where the second token is n-`GRAM` (where n is 1, 2 or 3) contain unigram counts $Count(y)$, bigram counts $Count(y_{n-1}, y_n)$, or trigram counts $Count(y_{n-2}, y_{n-1}, y_n)$. For example

  ```
  3792 2-GRAM O I-ORG
  ```

  indicates that there were 3792 instances of an `I-ORG` tag following an `O` tag and

  ```
  1586 3-GRAM O I-ORG I-ORG
  ```

  indicates that in 1586 cases the bigram `O I-ORG` was followed by another `I-ORG` tag.

Question 4 (20 points)

- Using the counts produced by `count_freqs.py`, write a function that computes emission parameters

$$e(x|y) = \frac{Count(y \rightsquigarrow x)}{Count(y)}$$

- We need to predict emission probabilities for words in the test data that do not occur in the training data. One simple approach is to map infrequent words in the training data to a common class and to treat unseen words as members of this class. Replace infrequent words ($Count(x) < 5$) in the original training data file with a common symbol `_RARE_`. Then re-run `count_freqs.py` to produce new counts. Submit your code.

- As a baseline, implement a simple named entity tagger that always produces the tag $y^* = \arg\max_y e(x|y)$ for each word x. Make sure your tagger uses the _RARE_ word probabilities for rare and unseen words.

  Your tagger should read in the counts file and the file `ner_dev.dat` (which is `ner_dev.key` without the tags) and produce output in the same format as the training file, but with an additional column in the end that contains the log probability for each prediction. For instance

  ```
  Nations I-ORG -9.2103403719761818
  ```

  Submit your program and report on the performance of your model.You can evaluate your model using the evaluation script: `python eval_ne_tagger.py ner_dev.key prediction_file`. In addition, write up any observations you made.

Question 5 (30 Points)

- Using the counts produced by `count_freqs.py`, write a function that computes parameters

  $$q(y_i|y_{i-1}, y_{i-2}) = \frac{Count(y_{i-2}, y_{i-1}, y_i)}{Count(y_{i-2}, y_{i-1})}$$

  for a given trigram $y_{i-2}\ y_{i-1}\ y_i$. Make sure your function works for the boundary cases $q(y_1|*, *)$, $q(y_2|*, y_1)$ and $q(\texttt{STOP}|y_{n-1}, y_n)$.

  Write a program that reads in lines of state trigrams $y_{i-2}\ y_{i-1}\ y_i$ (separated by space) and prints the log probability for each trigram. Submit the program.

- Using the maximum likelihood estimates for transitions and emissions, implement the Viterbi algorithm to compute

  $$\arg\max_{y_1 \cdots y_n} p(x_1 \cdots x_n, y_1 \cdots y_n).$$

  Your tagger should have the same basic functionality as the baseline tagger. Instead of emission probabilities the third column should contain the log-probability of the tagged sequence up to this word. Submit your program and report on the performance of your model. In addition, write up any observations you made.

Question 6 (10 Points)

Improve the way your HMM tagger deals with low-frequency words by grouping them based on informative patterns rather than just into a single class of rare words. For instance, such a class could contain all capitalized words (possibly proper names), all words that contain only capital letters and dots (possibly abbreviations for first names or companies), or all words that consists only of numerals. You can again replace words in the original training data and generate new counts by running `count_freqs.py`. Report on the classes and patterns you chose and how these methods impact the performance of your named entity tagger. Submit your program.