Exercises on *n*-gram language models, spelling correction, text normalization

Ion Androutsopoulos, 2018–19

Submit as a group of 3–4 members a report for exercise 4 (max. 5 pages, PDF format). Explain briefly in the report the algorithms that you used, how they were trained, tuned etc. Present in the report your experimental results and demos (e.g., screenshots) showing how your code works. Do not include code in the report, but include a link to a shared folder or repository (e.g. in Dropbox, GitHub, Bitbucket) containing your code.

1. (i) Using Laplace smoothing (slide 8) or optionally better estimates (e.g., K-N smoothing, slide 38) and (possibly a subset of) an English corpus (e.g., the English part of Europarl, available from http://www.statmt.org/europarl/), estimate the probabilities of all the word bigrams of the five candidate word sequences t_1^4 of slide 14. (ii) Using a bigram language model and the bigram estimates of the previous question, estimate the probability of each one of the five candidate sequences t_1^4 of slide 14. (In practice, *n*-gram language models compute the sum of the logarithms of the *n*-gram probabilities of each sequence, instead of their product. Why?) (iii) Using as $P(w_i|t_i)$ the inverse of the Levenshtein distance (e.g., as computed by http://www.let.rug.nl/~kleiweg/lev/) between w_i, t_i (or better probability estimates, which satisfy $\sum_{w_i} P(w_i|t_i) = 1$) and the formulae of slide 17, estimate $P(t_1^4|w_1^4)$ for each one of the five candidate sequences t_1^4 of slide 14. You may want to add a special *end* token at the end of each sentence. You may also want to use:

$$\hat{t}_1^k = \underset{t_1^k}{\operatorname{argmax}} \lambda_1 \log P\left(t_1^k\right) + \lambda_2 \log P(w_1^k|t_1^k)$$

to control (by tuning the hyper-parameters λ_1, λ_2) the importance of the language model score $\log P\left(t_1^k\right)$ vs. the importance of $\log P(w_1^k|t_1^k)$.

2. [Optional.] (i) Check the Levenshtein distance calculations of the table on slide 45. (ii) Implement (in any programing language) the dynamic programing algorithm that computes the Levenshtein distance (slides 40–45); your implementation should print tables like the one of slide 45 (without the arrows). (iii) Optionally extend your implementation to accept as input a word *w*, a vocabulary *V* (e.g., words that occur at least 10 times in a corpus), and a maximum distance *d*, and return the words of *V* whose Levenshtein distance to *w* is up to *d*.

3. Calculate the entropy in the cases of slide 24 (e-mail messages). Hint: for $P(C) \rightarrow 0$, use L'Hôpital's rule.

4. (i) Implement (in any programming language) a bigram and a trigram language model for word sequences (e.g., sentences), using Laplace smoothing (slide 8) or optionally (much better in practice) Kneser-Ney smoothing (slide 38). Train your models on a training subset of a corpus (e.g., from the English part of Europarl). Include in the vocabulary only words that occur, e.g., at least 10 times in the training subset; use the same vocabulary in the bigram and trigram models. Replace all out-of-vocabular (OOV) words (in the training, development, test subsets) by a special token *UNK*. Assume that each sentence starts with the pseudo-token *start* (or the pseudo-tokens *start1*, *start2* for the trigram model) and ends with *end*.

(ii) Check the log-probabilities that your trained models return when given (correct) sentences from the test subset vs. (incorrect) sentences of the same length (in words) consisting of randomly selected vocabulary words.

(iii) Estimate the language cross-entropy and perplexity of your models on the test subset of the corpus, treating the entire test subset as a single sequence, with **start** (or **start1**, **start2**) at the beginning of each sentence, and **end** at the end of each sentence. Do not include probabilities of the form $P(*start*|...)$ (or $P(*start1*|...)$ or $P(*start2*|...)$) in the computation of perplexity, but include probabilities of the form $P(*end*|...)$.

(iv) Optionally combine your two models using linear interpolation (slide 10) and check if the combined model performs better.

You are allowed to use NLTK (http://www.nltk.org/) or other tools for sentence splitting, tokenization, and counting *n*-grams, but otherwise you should write your own code.