3. (i) Implement (in any programming language) a bigram and a trigram language model for sentences, using Laplace smoothing (slide 8) or optionally (if you are very keen) Kneser-Ney smoothing (slide 46), which is much better. 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?) and you should do the same. Assume that each sentence starts with the pseudo-token **start** (or two pseudo-tokens **start1**, **start2** for the trigram model) and ends with the pseudo-token **end**. Train your models on a training subset of a corpus (e.g., a subset of a corpus included in NLTK – see http://www.nltk.org/). 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-vocabulary (OOV) words (in the training, development, test subsets) by a special token **UNK**. Alternatively, you may want to use BPEs instead of words (obtaining the BPE vocabulary from your training subset) to avoid unknown words. See Section 2.4.3 ("Byte-Pair Encoding for Tokenization") of the 3rd edition of Jurafsky & Martin's book (https://web.stanford.edu/~jurafsky/slp3/); for more information, check https://huggingface.co/transformers/master/tokenizer_summary.html.

(ii) Estimate the language cross-entropy and perplexity of your two models on a test subset of the corpus, treating the entire test subset as a single sequence of sentences, 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 $P(*start*|\ldots)$ or $P(*start1*|\ldots)$, $P(*start2*|\ldots)$ in the computation of cross-entropy and perplexity, since we are not predicting the start pseudo-tokens; but include probabilities of the form $P(*end*|\ldots)$, since we do want to be able to predict if a word will be the last one of a sentence. You must also count **end** tokens (but not **start**, **start1**, **start2** tokens) in the total length N of the test corpus.

(iii) Develop a context-aware spelling corrector (for both types of errors, slide 18) using your bigram language model, a beam search decoder (slides 20–27), and the formulae of slide 19. If you are keen, you can also try using your trigram model (see slide 28). As on slide 19, you can use the inverse of the Levenshtein distance between w_i, t_i as $P(w_i|t_i)$. If you are very

keen, you may want to use better estimates of $P(w_i|t_i)$ that satisfy $\sum_{w_i} P(w_i|t_i) = 1$. 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 $\log P(w_1^k|t_1^k)$.

You are allowed to use NLTK (http://www.nltk.org/) or other tools and libraries for sentence splitting, tokenization (including BPE tokenizers), counting *n*-grams, computing Levenshtein distances, but you should write your own code for everything else (e.g., estimating probabilities, computing cross-entropy and perplexity, beam search decoding). You may want to compare, however, the cross-entropy and perplexity results of your implementation to results obtained by using existing code (e.g., from NLTK or other toolkits).

Do not forget to include in your report:

- A short description of the algorithms/methods that you used, including a discussion of any data preprocessing steps that you performed.
- Cross-entropy and perplexity scores for each model (bigram, trigram) for sub-question (ii).
- Input/output examples (e.g., screenshots) demonstrating how your spelling corrector works, including interesting cases it treats correctly and incorrectly.