General instruction.

1. You are encouraged to use Python or Java. But other languages are also accepted.

2. You are encouraged to form 2 person teams on the programming assignment. Each team will only need to submit one copy of the source code and report.

3. You are allowed to make use of existing packages for some functionalities including handling file IO, tokenization and getting the raw counts of different tokens. Anything beyond this list will advance approval from the instructor. Also you need to clearly reference your source.

4. Your source code and report will be submitted through the TEACH site

   https://secure.engr.oregonstate.edu:8000/teach.php?type=want_auth

   In your report, please clearly indicate all your team members.

5. Your code should be clearly documented. Undocumented code can lead to lower score. Your report has one section that clearly describes how to run your code. The instructor may request a demo session in which you must demonstrate how your program works and how the results were produced.

6. Grading will depend on your code as well as your report. In particular, the clarity and quality of the report will be worth 20% of the grade. So please write your report in clear and concise manner. Keep it organized, and clearly label your figures, legends, and tables if any.

Language Modeling

Your tasks are to train a unigram (with MLE) and a bigram model (with MLE and Katz Back-off) on a training corpus of positive movie reviews and to test them on two test corpora: a test corpus of positive reviews, and one of negative reviews.

Training and test data: We have one training corpus, pos_train.txt, and two test corpora, pos_test.txt and neg_test.txt. Each line is a sentence. You need to surround each sentence by a start of sentence and end of sentence marker (e.g. `<s>...</s>`). Tokens (this includes words and punctuation marks, which you should treat like regular tokens) are separated by whitespaces.

Dealing with unknown words In testing stage, if we encounter words that have not appeared in the training corpus, they are called unknown words. Here is a simple strategy to deal with unknown words. At training stage, we will define a fixed vocabulary $\mathcal{V}$ based on the training corpus, which contains all the tokens in the training corpus that occur more than $k$ times (for this implementation please use $k = 1$). Replace all the words in the training corpus that appeared only once with a special token ‘UNK’ standing for unknown. Estimate the model treating ‘UNK’ as part of the vocabulary. At testing stage, any token outside of the fixed vocabulary $\mathcal{V}$ will be turned into the ‘UNK’ token.
**Bigram model: MLE and Katz Back-Off**  You will learn the bigram model from the training corpus using MLE with and without Katz Back-off. In particular, bigram MLE:

\[ q_{ML}(w|v) = \frac{\text{Count}(v,w)}{\text{Count}(v)} \]

To implement Katz Back-Off, please refer to Section 1.4.2 of Michael Collins’ note on language modeling. Set the \( \beta \) parameter to 0.5 for the tasks 1 and 2 below.

**Task 1: Generating sentences**  (15 pts) Please use your unigram and the basic MLE bigram language models to generate 10 sentences, and compare the probability assigned to these sentences by the unigram and the Katz bigram model.

**Task 2: Computing the perplexity of the test data**  (30 pts) the perplexity (normalized inverse log probability) of the two test corpora using the unigram model and the Katz bigram model. What do you observe from the results? What difference do you see between the two models? How does the result differ between the two test corpora? Please provide possible explanations for your observation.

**Task 3: Effect of the \( \beta \) parameter on the Katz bigram model**  (20 pts) Consider a set of different possible values for the \( \beta \) parameter (e.g., 0.3, 0.4, ..., 0.7). Plot the perplexity of each of the test corpus as a function of the \( \beta \) value. What observations can you make based on the results? Provide possible explanations for your observation.

**Hints for Implementation**  In order to avoid underflow, your implementation will have to use log-probabilities internally. That is, once you have computed the probability of each bigram, you should convert it to its logarithm. Make sure you always use the same base.

Note that storing all the possible bigram probabilities would require a lot of memory. One possible way to address this is to compute the probability of bigrams that don’t occur in the training data on the fly.