Initialization

We begin our algorithm by constructing an NGram object that is a modified version of Group 3's N-Gram program. We train this N-Gram off a PlainTextCorpus using a document we found to serve as our corpus. The document is well over a million words with excerpts from Gutenberg, news articles, and the British National Corpus. We found that training the model on more corpra produced better results but drastically reduced the time because of how much was needed to process. With a PlainTextCorprus and chosen corpus we reduced the time needed significantly and produced similar results for suggesting words. The initialization of the NGram includes creating a list of all "correct" words using the words corpus from NLTK. Lastly in this initialization of the NGram, we construct counters for the unigram and bigram. We chose to only go to bigram and not a trigram or beyond since we found almost no noticeable difference in quality of output, but significant time spent on initialization. We also chose to not store and load the NGram but rather construct it at the beginning since it took similar time to load it from a file.

Algorithm

The first thing we do in our algorithm is to tokenize the sentence passed into the check function. We strip out the punctuation to make it tokenized on purely whitespace. Next, we loop over every token and check if it's valid. Validity is checked using the original word, Porter Stemmer, Snowball Stemmer and the WordNet Lemmatizer. If any these versions of the word are found in the words corpus then we consider it to be correctly spelled. If the word is misspelled, then we first retrieve all candidates based on pure edit distance by calculating all the permutations and then filtering the list down to valid words only. We chose to either calculate all words two edit distance away or quit if we get above a certain threshold for number of words that are one edit distance away. We found 25 words to be a pretty good threshold for finding the edit distance candidates. Next, we construct a bigram and use it to calculate possible candidates based on the NGram. We filter out any words that are both in the edit distance candidates and the NGram's candidates and return those. If there is no overlap then we sort the two sets of candidates, take the top five from each, combine them and then sort it again. We sort the NGram candidates on edit distance and the edit distance candidates on NGram frequency. We then pick the top five candidates as our final 5. In filtering the NGram candidates we make sure to only pick candidates within a certain threshold for edit distance which we chose to be 3. We cache the result to use in case the misspelled word is seen again. One issue we found is that for misspelled words like "uall", the possible candidates with the same edit distance is so numerous that it is hard to find the correct one, especially if it never appears in the corpus. The sorting function used by Python sorts items with the same sort criteria is random order and so the correct word can sometimes not appear within the top 5, even though on some runs it does.