Information Retrieval

Programming Assignment 2

Daniel Erenrich & Daniel Rosenberg

Language Model

Our language model constructions precomputes all the structures we need to quickly perform our correction computation. We loop through each file in out data corpus, counting every word and bigram of words we see within a Counter. While we do this, we build a kgram (for k =3) indexing of the words we see for later use. We use the standard method provided in the set to compute our probability of a given query. We compute the mle's of the probabilities by dividing counts. The mle probability of word 2 given word 1 is the count of both divided by the count of the first one, while the probability of any given word is just its count divided by the number of words in the data set. We use N-gram Interpolation, as suggested, to account for bigrams that don't appear. We take the log of the probabilities when calculating the probability of seeing the entire query. Finally, we look at our empirical edits data to precompute counts. Our DamerauLevenshtein function extracts what corrections were made, and then counts the appearance of individual letters, as well as the operations done on the letters, which we will later use to calculate the empirical probabilities of different kinds of corrections.

Candidate Generation

The standard way of generating candidates as proposed by Norvig is to make all possible edits to the query and check which changes result in valid queries (all words appear in our dictionary). The problem with this method is that even for short queries it generates many queries. We implemented this method and some simple heuristics (not calling it on very probable words in the query) but it still generates many spurious candidates. This code is still in the corrector.py file.

Instead of trying to optimize this we switched to kgram candidate generation. During the language modeling phase we split each word into a series of kgrams (we use k=3) maintain a dictionary of kgrams mapping kgram strings to word indices (much like in indexing each word is assigned a numeric value). This index allows us to find words similar to a given string. To do this we split the string up into kgrams and then find all words which share at least some fraction (we use 50% but tuning this constant is a tradeoff between speed and accuracy) of kgrams with that word. Finally we ensure that the candidate is within a length of 2 of the original string and that it is within an edit distance of 2 of the original string (the length check is done because it is much faster to compute).

This method of candidate generation is very good for many words but is not complete for queries and nor does it handle short words well. For short strings(shorter than 4 characters) we drop back and look at all 1 character edits to the word. To handle queries of multiple words we produce candidate replacements for each word and then compute the cartesian product across all candidates. As an optimization for speed we only generate candidates for a word

already in the dictionary if that word occurs less than some percent of the time (we use 1e-6). This algorithm still cannot handle word merges. To deal with that we take the original query and expand it to many queries where in each case one of the spaces is deleted (this is efficient as long as the query does not have very many words).

We also handle the case where one word needs to be split into two. We try expanding the query by inserting spaces at every possible location (such that it generates 2 valid words). Note that this does not find splits such that it generates a word and a word where an additional edit is needed. Doing that optimization created far too many candidate queries and slowed the corrector down substantially.

For a given query we generally score between 1 and 300 candidate strings. Our program is still fairly slow though because of the time spent generating these candidates and the slowness of our scorer.

Candidate Scoring

Note that for our scoring process to work we had to write a DamerauLevenshtein function which returns what edits must be made to get one string from another. This function is the main bottleneck in our program and if it were rewritten we would get a large speed boost.

When we judge which candidate to choose from, we compute the probability of the chosen query, and the probability of the corrected typos for that query. We tuned mu to weight the relative importance of these factors. For the probability of the query, we add the log probabilities of each bigram in the sentence, along with the unigram probability of the initial word. To compute the edit probability under our uniform edit model, we count the number of corrections, and use a constant as the probability of making an edit. In our empirical cost version, we instead use the counts extracted from edits1.txt.gz in our model generation to calculate the probability of making particular edits. We then simply select the most probable candidate as our correction.