Inverted Index

Samuel Zureick

Task 1: Inverted Index

Index Term Choices

The index terms that I chose to select were a combination of specific multi-word values and single words that do not appear in the NLTK stopwords. The inverted index specification said that it would be used to locate episodes containing specific characters and locations, and many of these were multi-word, so I felt it necessary to allow for these specific multi-word values to be included in my index. I used the NLTK tool MWE tokenizer to add these Simpsons-specific keywords to my index, which replaces whitespace with underscores in my usage to create pseudo-single-word indexing, allowing me to proceed with the task as I would with single word indexing. This approach is sufficient for this task, where we are given an extensive list of these specific items, but in the case where these are not provided, my method would only support single word indexing, which would split up terms that should be together. Furthermore, if we decided to add new Simpons episodes to the dataset with new locations or characters without updating the provided character and location files, these potentially poignant terms would be treated as single words and would not be tokenized consistently with the rest of the characters and locations. For example, Simpsons Episode 1 of season 29 takes place in Springfieldia, and Homer visits Springfieldia Human Power Plant- neither Springfieldia nor Springfieldia Human Power Plant appear in the location sheets provided, and Springfieldia Human Power Plant would be individually tokenized by whitespaces rather than treated as a single token with underscores instead of spaces, missing out on one of the main functionalities of the tokenization method. I chose to remove stopwords from the text in order to shorten runtime and space taken up by the index.

Pre-Processing Choices

I chose to use wordpunct tokenizer to split each space delimited substring of the input document into a list of initial tokens. Initially, there are no multi-word tokens that are generated- this actually occurs in the inverted-index step, because I didn't think that it made sense to attempt to tokenize these more complex terms before text normalization and removal of stop words- it would be easy for the multi-word tokenizer to miss over these specific tokens if the corpus text wasn't normalized, like I do before initializing the MWE tokenizer with my Simpsons character and location tokens.

After using wordpunct tokenizer, I use a list comprehension to accomplish a few tasks at once. The comprehension iterates through each token generated in the previous step and removes all non-alpha-numeric characters, followed by case folding. This token processing only takes place, however, in the case that the word does not appear in the stopwords. One way that I sped up the stopwords removal process was by using python's Counter feature- by creating a dictionary of each of the stopwords instead of keeping it as a list of words, the line of code <if token not in stopwords> operates with a O(1) hash lookup instead of an O(n)

list search- this alone brought my total program runtime from around 25 minutes to around 25 seconds.

One problem with my stopwords removal is that any multi-word tokens that I chose to index based on the character and location files that contain stopwords are tokenized without stopwords removed, meaning that “homer_the_thief” is a generated multi-word token but any text containing this will be tokenized as [“homer” , “thief”] after this pre-processing.

Inverted Index Contents

My inverted index contains all the Simpsons specific tokens and also includes all other words that were not in the stopwords. The index is structured as a dictionary, with the key being the token itself and the value being a list, composed of the document frequency for the term and postings list. The postings list is a list of tuples, with each tuple containing the docID that the term appears in and the location of the term in the text. Constructing this dictionary allows you to quickly check if a term appears in a specific document- simply referencing the key provides the value stored there with constant lookup time as a python dictionary is a hashed datatype.

In total with all of the provided documents, the inverted index stored postings for 13352 unique tokens, and each posting list contained an average of about 7 occurrences, yielding a total of about 93400 unique combinations of token, document ID, and position.

One challenge that I ran into while sorting the indexed terms was sorting first by the token alphabetically, then by the season number and then by the episode number. I managed to do this, but it was something that I overlooked initially, and depending on the structure of the document IDs that are held, the implementation of this sort might change. For example, in this implementation season 3 episode 11 (doc ID 3.11) occurs after season 3 episode 2 (doc ID 3.2), but a naïve sort resulted in 3.11 being placed prior to 3.2.

I have chosen to include the positions that each token occurs with the document ID in the postings list in order to allow for proximity search. Without the locations of the tokens in the text, I could still support a document-level Boolean query, but by including these I can see how the locations of each token found relate in each document. One downside to my implementation of this, however, is that the positional index for each term is generated after stopwords are removed, meaning that the locations stored in the postings list are not an accurate reflection of their location in the initial corpus. I believe that for an application such as this, my solution is still a valid solution because relative locations are still provided, allowing one to see how close specified tokens coexist in a document, and most people searching using proximity search are using this as a way to find documents mentioning a variety of combinations of characters, locations, and other terms, not as a way to find exactly where in the text a token is located.

Task 2: Proximity Search

My proximity search implementation starts with doing a small amount of pre-processing for the input terms. This is so that anything that the user inputs will be formatted in the same

way that the tokens stored in the inverted index will be. The pre-processing task is two steps, the first being removing all non-alpha-numeric characters and the second task being case folding. I do not attempt to use the multi-word tokenizer for multi-word search terms, as the specification asks for two single terms, however a user familiar with this implementation could put in specific multi-word terms separated by underscores, and this should yield appropriate results. For example, “Bart Simpson” would not be a valid term as input to the proximity search, but “Bart_Simpson” is, and the proximity search works as expected with this input.

The second step of my proximity search implementation removes the document frequency count from the postings list for each term to make array manipulation more readable.

Next, I calculate the intersection of the two term posting lists based on the document ID, so that only those potential matches based on document ID are stored in the postings.

After this, I use a list comprehension to find all elements in the list that have the same document ID and have index values that fall within the supplied window- this is accomplished using the python range function, and a pair is only returned if the position of one item falls within the range of another item in the postings list for the other index.

Positional indexing could be used as an alternative for multi-indexing for this task. One issue that could be encountered would be in the case of index terms that have a length greater than 2- my current implementation only supports proximity search for 2 terms. A way to work around this could be to write the proximity search function in such a way that it accepts multiple terms, but I would have to significantly restructure my algorithm as this would not be very efficient. Another solution would be to recursively call the method while iterating through the list. For example, for the character “Woman In Booth”, the algorithm could first perform a proximity search on “Woman” and “in”, average their positions, and then perform a proximity search on “booth” with the other term having the position calculated previously.