CP 423 Text Retrieval & Search Engine – Assignment 2

Bryan Gadd (170451160), Herteg Kohar (190605160), Kelvin Kellner (190668940)

What each member contributed:

- Bryan Gadd (170451160)- Q3
- Herteg Kohar (190605160)- Q1, Q2, Q3
- Kelvin Kellner (190668940) - Q4

Q1:

How we understand the question:

- We read the documentation from nltk and decided on the proper functions and classes to utilize in order to complete the task at hand.

How we implement the question:

- Any functions required in the question were imported from the nltk module. These functions and classes included the RegexpTokenizer, PorterStemmer, and stopwords. All other functionality was done by using built-in methods and classes native to Python. For the tokenizing argument of the program the RegexpTokenizer class was used from the nltk module. For the stopwords option, the list of stopwords is downloaded using nltk and the list of tokens from the corpus is then filtered using the list of stopwords from nltk. The stemming option utilized the PorterStemmer class from nltk and by passing in the list of tokens from the corpus. The inverted index was created by tokenizing all the selected articles and then adding each of the tokens to the Counter to count each occurrence for the given article. After each of the article id and counts are appended to the inverted index for each token. To create the zipf plot for the corpus all the tokens are counted and then the token counts are sorted in reverse order. This is then plotted on a log-log plot based on the token's rank.

Any challenges/issues that we had:

- An issue that arose was the resources needed to try and process all the documents into one large corpus. There were memory errors which had occurred when trying to do this. To get around this threading was used in batches to speed up the processing as well as compute the processing without getting a memory error.

Q2:

How we understand the question:

- The slides contained information on how to compute the codes for each of the algorithms, gamma and delta respectively.

How we implement the question:

- The delta and gamma encoding utilized two helper functions for the unary and binary encoding. The unary encoding function made the string n-bits with leading 1s corresponding to the examples on the class slides.

Any challenges/issues that we had:

- It’s hard to understand whether the unary part of the code should have leading 0s or leading 1s. In the course notes the unary portion contains leading 1s however, from online resources and the assignment file itself contains leading 0s for the unary portion.

<u>**Q3**</u>:

How we understand the question:

- The textbook had a passage on how to compute the PageRank and implement the algorithm with the random surfer coefficient.

  $$PR(u) = \frac{\lambda}{N} + (1 - \lambda) \cdot \sum_{v \in B_u} \frac{PR(v)}{L_v}$$

  We compute the PageRank as normal and then multiply by a coefficient of 1 – lambda and add lambda over the number of documents.

How we implement the question:

- A class named Node was created in order to store the node’s id and the nodes it pointed to as well as the inbound edges from the other nodes in the graph, as well as its PageRank. The Web-Stanford.txt file was then parsed to get all the required data structures to compute the PageRank according to the command line arguments given by the user. We then iterate over all the nodes and compute each page rank respectively. At the end of this we check for convergence using the following equation.
- $||new - old|| < tau$
- If convergence is reached the designated node’s PageRank's are then displayed to the output as well as the total PageRank

Any challenges/issues that we had:

- One issue we had was understanding how the random surfer coefficient (lambda) affected the total PageRank of all pages. We were confused why the total was not close to 1, we thought we made a mistake in the code. Suddenly we had a eureka moment, the total PageRank was close to 0.75 and the random surfer coefficient was 0.25. We realized we had implemented it correctly after all! The PageRank equation above explains why: the factor (1-lambda).
- Old method for tracking nodes and link pointing used lists which required linear search which is an independent O(n) operation embedded within the O(n^2) loop creating a total runtime complexity of O(n^3). We then utilized data structures such as sets and dictionaries which have constant lookup times and brought our time complexity back down to O(n^2) making our program much more efficient.

<u>Q4</u>:

How we understand the question:

- The point of this question is to implement the Noisy Channel Model for spell-checking words. The model tokenizes a corpus of text and uses this as the vocabulary of known words. When a word is given to the program it should either 1) output the proportion of occurrences of tokens in the text which are the input words- the probability, or 2) output the most likely known words to be used as a spell correction of the given word. 1 and 2 are two different modes in the program, “proba” and “correct” accordingly. The program uses 1-edit distance and 2-edit distance to produce possible candidate words to suggest as corrections for the given word.

How we implement the question:

1. The program uses a condition to check if the corpus has already been tokenized. Question 1 uses the same corpus and the same processing, so there is a chance this step has already been done. Also, if the program is run more than once in a row there is no need to do this again. If the file is not found, a subsection of the Wikipedia dataset is read, tokenized, and saved to a file.
2. A word frequency vector is created with a count of occurrences for each token in the corpus. For each of the given words, the program:
    a. If the mode is “proba”, the probability is printed to the console (count of given word from the word frequency vector divided by total number of tokens in frequency vector).
    b. If the mode is “correct”, the best spell correction is printed to the console. Pattern:
        i. If the given word is a known word, the word is already correct, use it.
        ii. If there are 1+ known words that are 1 edit distance away from the given word, use the one with the highest probability. If there are none...
        iii. If there are 1+ known words that are 2 edit distance away from the given word, use the one with the highest probability. If there are none...
        iv. The algorithm does not have a correction for the given word. Use it as is.

Any challenges/issues that we had:

- No major issues occurred. Herteg’s work for A1 made processing the corpus super easy. The textbook and the online resource made the Noisy Channel Model easy to implement.