Information Retrieval Assignment Documentation

Choosing a Data Structure for any information retrieval search engine is considered as a first step to develop a search engine. These data structures include Trie, Hash Tables, Suffix array, Suffix tree, Red-Black trees, B trees, BK trees, deterministic acyclic finite state automaton (DAFSA). A wise trade-off between speed and memory leads to the data structure which is required need for his Search Engine.

We have considered the following criteria while designing the systems:

- Spelling Corrector
- Suffix wildcard query
- Prefix wildcard query
- Good precision and recall of the returned documents.

Steps to make an effective Information retrieval Search Engine

1) Tokenization and Normalization:
NLTK's inbuilt tokenizer 'word_tokenize' is used. All the tokens are converted to lower case. Porter's stemmer and Snowball stemmer were explored.

2) Spelling Correction and Wild Card Queries:

If space is available, R-way tries may be used for completing the job with a constant number of character compares. For large alphabets, where space may not be available for R-way tries, TSTs are preferable, since they use a logarithmic number of character compares, while BSTs use a logarithmic number of key compares. Hashing can be competitive, but, as usual, cannot support ordered symbol-table operations or extended character-based API operations such as prefix or wildcard match.

To handle the wildcard queries one is left with data structures like DAFSA, R-way tries, TSTs, and other tree or graph data structures. Out of these Trie is the data structure delivering the comparable performance like the hash tables. But the **R-way trie wastes a lot of space** and is therefore discarded as an optimal data structure for handling wildcard queries. Therefore Ternary Search Tree is chosen as an optimal data structure to handle wild card queries.

Ternary tries are capable of handling search hit and insertion in comparable times to that of a Hash table but for the case of Search miss it is faster than the hash table and Ternary trie has the space complexity not greater than Hash Tables.

implementation	search hit	search miss	insert	space (references)
red-black BST	$L + c \lg^2 N$	$c \lg^2 N$	$c \lg^2 N$	4 N
hashing (linear probing)	L	L	L	4 N to 16 N
R-way trie	L	$\log_R N$	L	(R + 1) N
TST	L + ln N	ln N	L + ln N	4 N

Comparison of performance of various data structures for Information retrieval model.

Ternary search tries (TSTs)-To help us avoid the excessive space cost associated with R-way tries, an alternative representation of the ternary search trie (TST) is considered. In a TST, each node has a character, three links, and a value which is the weight of that word. In this case IDFs are considered as weight of a word. The three links correspond to keys whose current characters are less than, equal to, or greater than the node's character. In the TST, characters appear explicitly in nodes—we find characters corresponding to **keys only when we are traversing the middle links**.

Insertion- To insert a new key, we search, then add new nodes for the characters in the tail of the key. New letter is added to the middle node of the trie, and if it is occupied by any other letter then the new letter is compared with the letter of the middle node and if it is smaller than the middle letter then the new letter is added to the left side otherwise it is added on the right side of the node. At the end of the string weight of string is added which indicates the completion of string as well as its unique ranking in the corpus.

To handle suffix queries a reverse ternary trie is constructed which uses the same procedure of insertion, using the term reverse ternary trie may induce a confusion in reader's mind but it is creating ternary search trie for the reversed string.

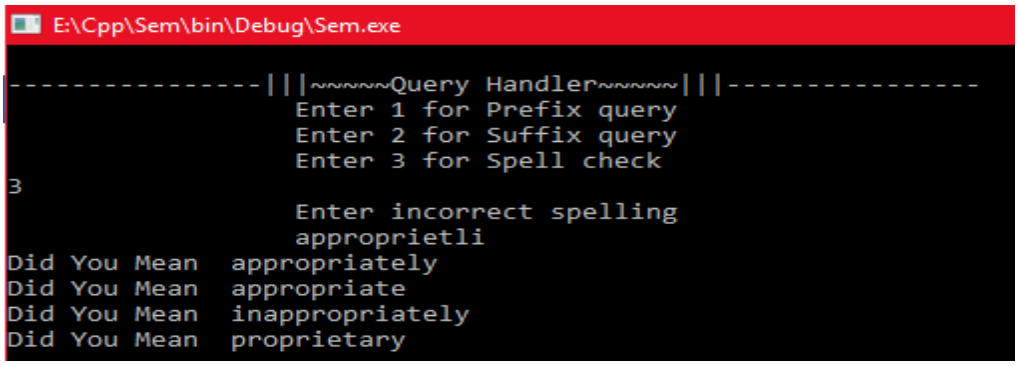
Search-To search, the first character in the key is compared with the character at the root. If it is less, we take the left link; if it is greater, we take the right link; and if it is equal, we take the middle link and move to the next search key character. In each case, this algorithm is applied recursively. In the process of searching if a null link is encountered than the process of searching is terminated. If the wildcard query gets over while searching then all the words which are having non-zero weight are returned.

Spelling correction- It is among the most important feature which a user expects from the search engine. Spelling correction could be context based spelling corrector or a simple spell corrector which retrieves the word having the smallest distance from the query word. In this assignment a method to calculate Levenshtein distance (LD) is used and the first four words having the smallest distance from the query word are retrieved.

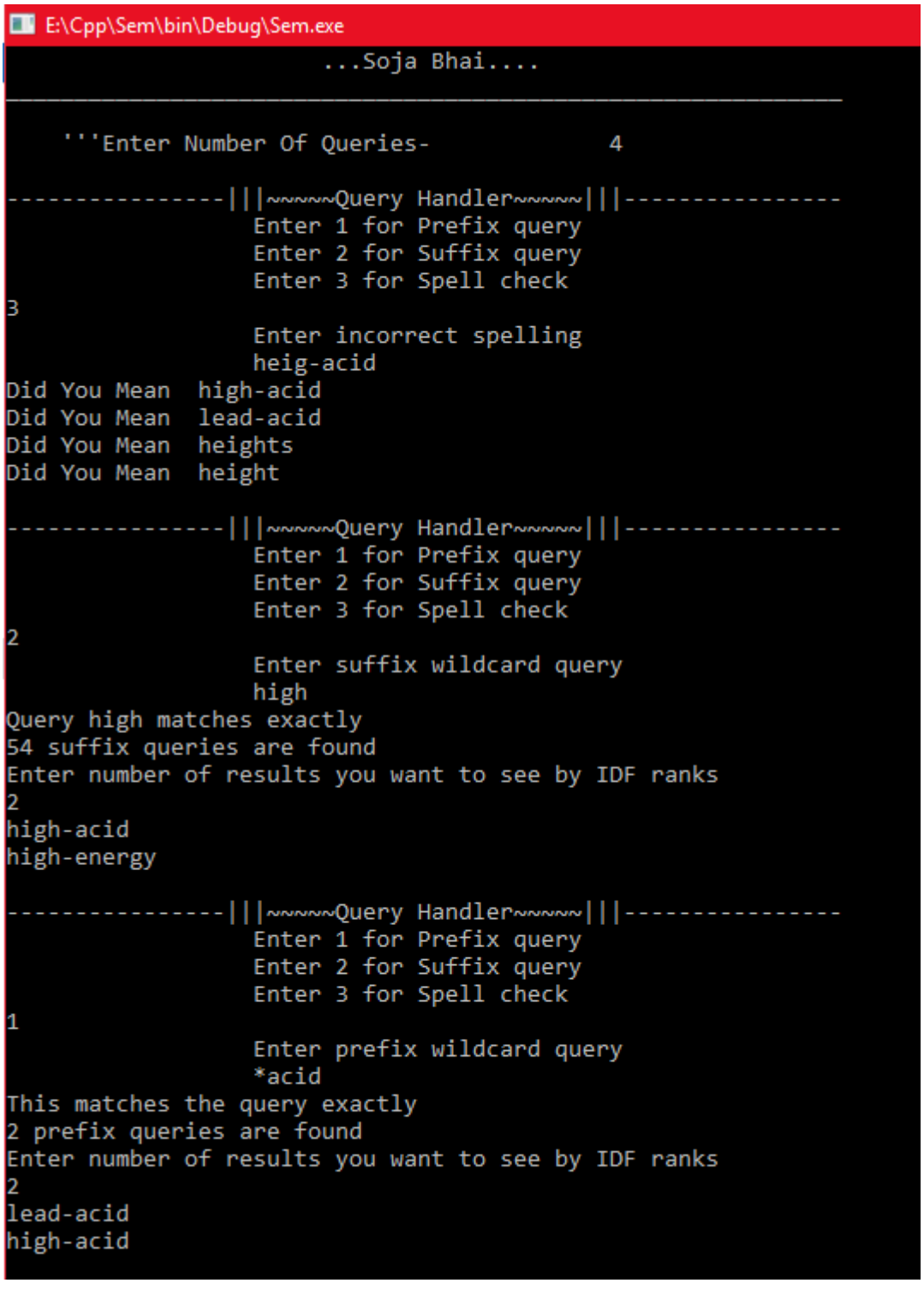
To calculate LD a method of Dynamic Programming is used which recursively computes the distance of query word with the words in the dictionary words. This method uses the technique of memoization, which makes it usable in real time scenario for the dictionary having a large amount of words.

A method of preprocessing the LD distance of all words in dictionary with each other is stored in the BK trees which could be used to compute the LD between the query word and the string having the smallest distance with that string.

The LD between two strings of length n can be computed in time of $O(n^{2-\varepsilon})$ where $\varepsilon > 0$ is a free parameter to be tuned.



Query Handler matching incorrect word approprietli to correctly spelled words



Query Handler performing Wildcard Query and Spell correction

3) Data structures and flow of code for the Vector Space Model:

- Inverted index: Python's dictionary (hash tables) having terms as keys and values as variable sized array for posting list.
- IDF values are stored in a dictionary with keys as terms and values as corresponding IDF.
- TF is stored in an index similar the data structure of inverted index, only instead of list containing doc Ids, it is a list of tuples [docID, TF(t,d)].
- Normalized document vectors are constructed and stored in a two dimensional numpy array.
- When a query is fed to the application it converts it into a normalized V dimensional vector implemented as a numpy array. An N (Number of documents) dimensional score array is created and initialized to zero. For each term in the query the posting list is fetched and the corresponding term in the score vector is updated incrementally according to the TF-IDF weights of the query term and document.

4) Retrieving more intuitive documents leveraging the word2vec model:

Even though the tf-idf scoring scheme gives results of modest relevance, it fails to take into account the context of the query. For example, if the query contains the term 'good' it will only search for that particular word and give zero weight to contextually similar words like 'positive', 'better', etc.

For this we have used the word2vec model to retrieve documents intuitively similar to the query. Word2vec models use neural networks to learn the vectors of words such that the vector captures the context of the word.

We used the gensim package which is a python library that uses deep learning to create word embeddings using the skip gram model and continuous bag of words model. After the model was trained on the corpus it could output top similar words for each term in the corpus. So, along with the query terms it also appends top two similar words for each query term.

Since the reuters data set is relatively smaller than what is required to train a very good deep learning model, the top similar words are not always acurate. So even if it increases the recall, precision might get reduced for some queries. Hence, inorder to give more weight to the original terms in the query, the weights of the additional terms is halved.