Design Document

This project is aimed towards creating a search engine for retrieving research papers matching a user provided query. The architecture of this application is discussed below.

Core

Core of this application is a flask server, which handles all sort of user requests and responses.
The server consist of following major parts,
1) REST API endpoints for providing responses for search suggestion and relevant document
2) Handling UI templating and user interaction using web-sockets for continuous bi-directional data flow.
3) Loading and querying data structures used for word completion and document retrieval

User Interface

The user interface presents a simple search field that provides the user with the functionality of word auto-completion. The User needs to enter the search keywords which are automatically pre-processed and relevant documents are fetched and presented to the user. The User can then view the document abstract, related keywords for each document and authors and subject. Some additional functionality that is provided to the user includes,

- Downloading the selected research paper directly
- Searching similar documents using the top keywords from the document.
- Search using a specific keyword from the document.

Data Structure

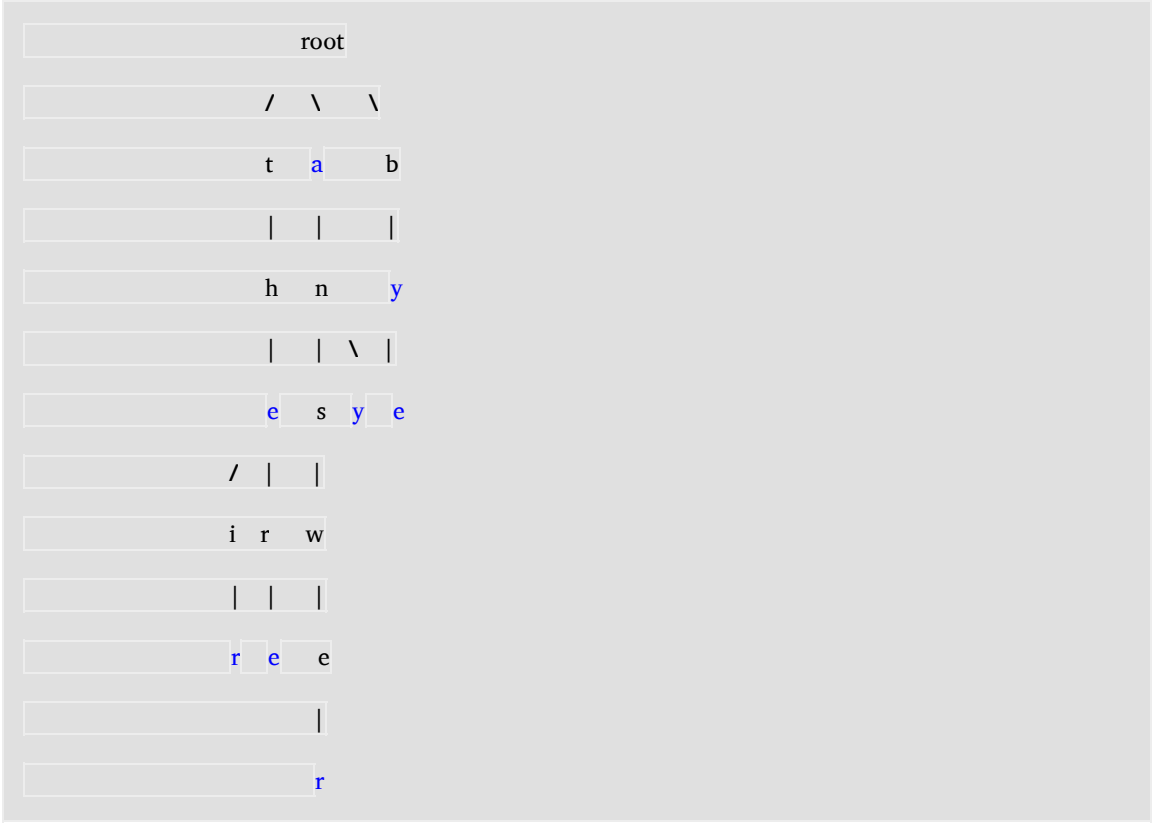
Data structure used for this project is a **Trie.**
Trie is an efficient information reTrieval data structure. Using Trie, search complexities can be brought to optimal limit (key length). If we store keys in binary search tree, a well balanced BST will need time proportional to **M * log N**, where M is maximum string length and N is number of keys in tree. Using Trie, we can search the key in O(M) time. However the penalty is on Trie storage requirements.

Every node of Trie consists of multiple branches. Each branch represents a possible character of keys. We need to mark the last node of every key as end of word node.

Inserting a key into Trie is simple approach. Every character of input key is inserted as an individual Trie node. Note that the children is an array of pointers (or references) to next level trie nodes. The key character acts as an index into the array children. If the input key is new or an extension of existing key, we need to construct non-existing nodes of the key, and mark end of word for last node. If the input key is prefix of existing key in Trie, we simply mark the last node of key as end of word. The key length determines Trie depth.

Searching for a key is similar to insert operation, however we only compare the characters and move down. The search can terminate due to end of string or lack of key in trie. In the former case, if the Leaf Node is reached, then the key exists in trie. In the second case, the search terminates without examining all the characters of key, since the key is not present in trie.

The following picture explains construction of trie using keys given in the example below,



Text Pre-processing

There are two different areas that need text pre-processing.
First is processing and cleaning the raw documents and making them more usable by removing the special characters (MATHJAX characters are extensively used at arxiv and they need to be removed before constructing the data structure as they are irrelevant).

Secondly these cleaned documents undergo further pre-processing steps such as stemming, lemmatization etc. All of these techniques are separately implemented and their performance is compared.

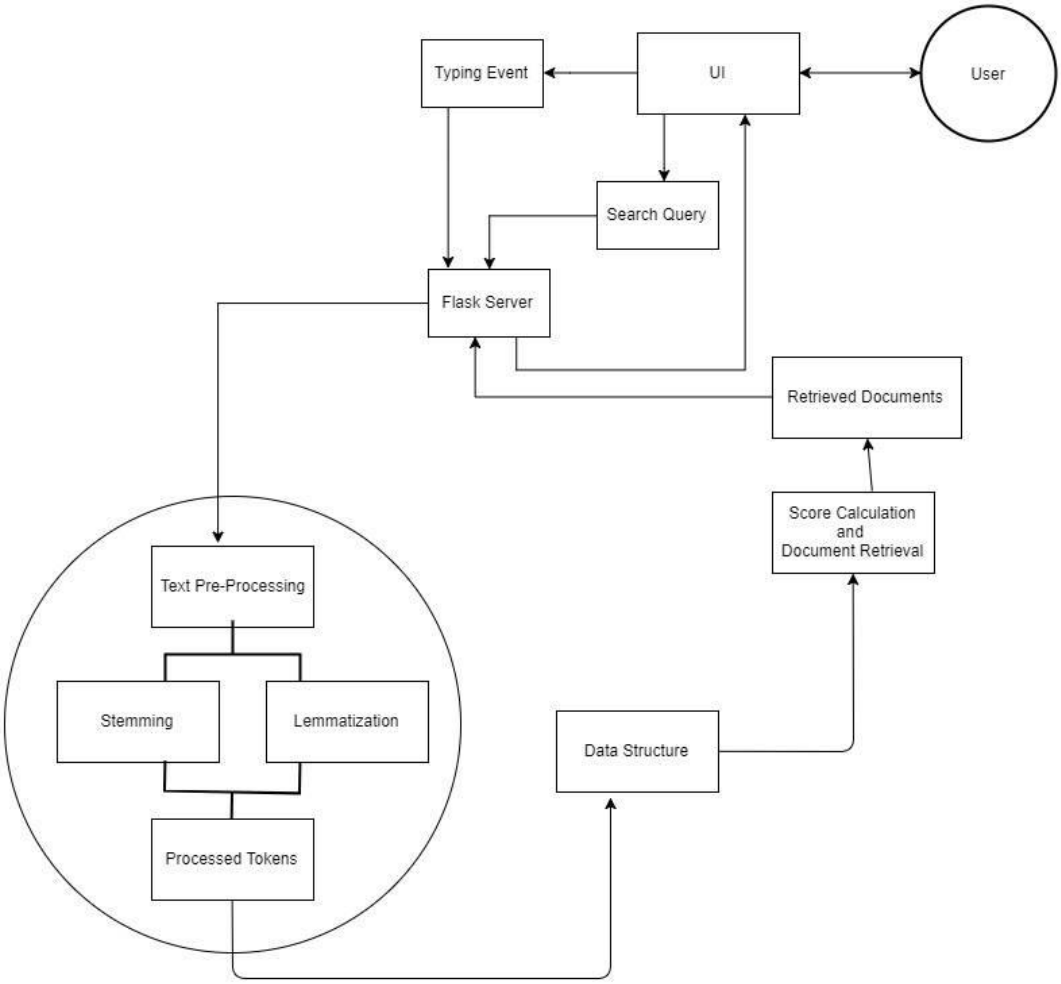
Below are some of the results,

-> Number of documents = 15686
-> Number of unique words after initial pre-processing = 38,773

S No.	Pre-processing Type	Time Taken
1	Only special character removal	41.67 sec
2	Special character removal + Stemming	146.13 sec
3	Special character removal + Lemmatization	67.14 sec

*time taken = document reading time + pre-processing + trie insertion

Implementation Logic



Data flow Diagram

Once the trie is created after the pre-processing step, it is saved to disk in pickled form. This trie is then used for querying and for word auto-completion.

While doing word **auto-completion**, for every character the user types, the partial string is sent to the trie. While traversing the trie, if at any node the number of leaf nodes below it is less than a certain **threshold**, the words corresponding to those leaf nodes is given as auto-complete suggestions to the user. This setup allows us to adjust the **precision and recall** of the auto-complete suggestions. If the threshold is set too high, the recall of the auto-complete system increases but the precision suffers. If the threshold is set too low, the recall suffers but the precision improves.

While **querying**, the trie is traversed from the root node to the leaf. While traversing, if the leaf node has been reached, the term exists in the trie and the inverse document frequency of the term and its term frequencies for all the documents is returned as a

Python dictionary. If, while traversing, a node has been reached which has no child node corresponding to the next character of the term, the term doesn't exist in the trie.

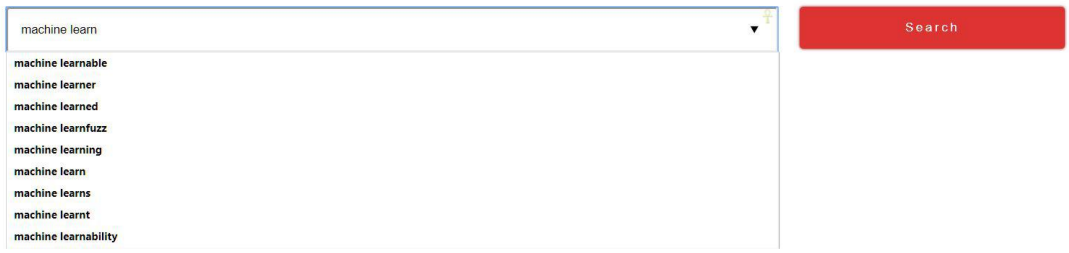
Score Calculation -

A zero matrix of size (Nx2) is initialized where N is the number of total documents. Tf-idf for each word in the query is calculated. On traversing the trie for each unique word in the query, the corresponding documents are selected and the score of each document is calculated corresponding to that term as follows,

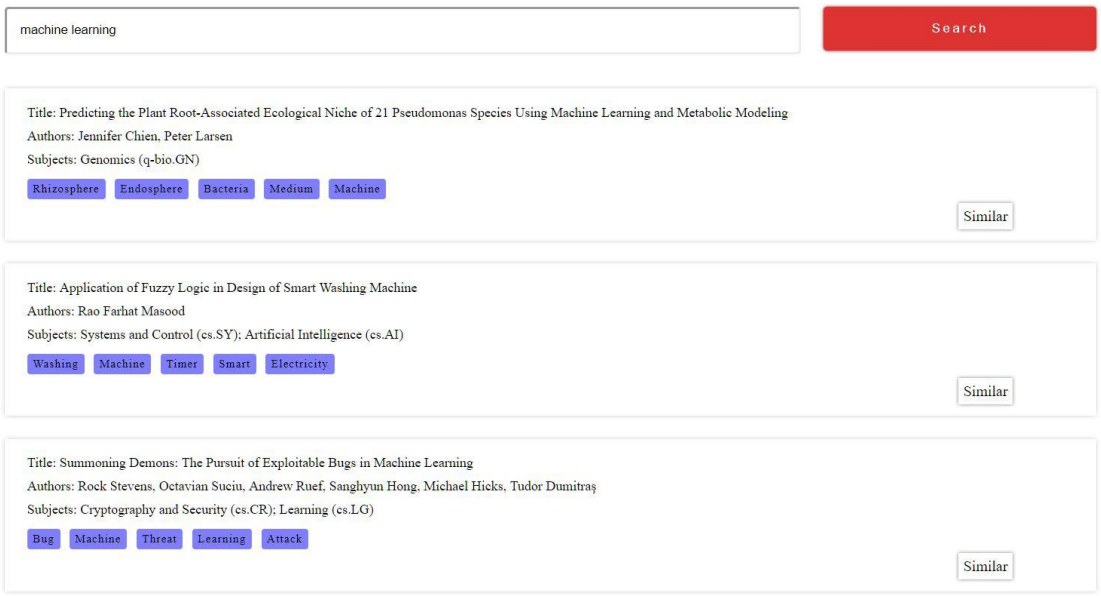
*Score = tf-idf of term in query * tf-idf of term in the document*

This score is updated in the matrix. This process is repeated for each unique term and finally the documents with the highest scores are retrieved and returned.

Screenshots -



Word Auto-completion



Retrieved Documents