searchRit

# Documentation

**BACKEND:**

The backend comprises of three parts: KMeans Clustering Engine, Server serving the machine learning model and a Search Engine. The technology stack utilized for the development of back-end is: Python, Node Js/Express Js (Javascript).

As of late Python has been the choice for machine learning, data scientists and scientific computing community as it offers a wide array of libraries to carry out complex computations. It offers many great bindings for the purpose of developing models utilizing Clustering, Classification and other machine learning techniques. It is great for prototyping machine learning and natural language processing models fast and allows its users to concentrate on core areas of business processes and development of functionalities.

Python has a syntax which is closest to a readable pseudocode. Apart from the syntactical sugar that it offers, it is very much geared towards math and you get most of the libraries inclined towards mathematics and scientific computation by default. Since, we required a machine learning model, which essentially is a mathematical optimization only, makes Python an obvious choice here. Python also has to offer state of the art libraries for Numerical linear algebra (numpy), matrix algebra (numpy), Convex optimization, Statistical modeling, Natural language processing (Natural language toolkit) and Machine learning (Scikit-learn, Tensorflow) in general.

We incorporated the use of data frames, mathematical calculations, tokenization, stemming and machine learning functions. As we didn’t have predefined class labels and we wanted to find out what people were talking from the document we did resort to unsupervised learning and used simple K-means clustering to find what the documents were talking about. With the help of clustering we are able to show most talked about topics in the RIT community. To speed up the process the model has been persisted by pickling it and deployed by utilizing client-server architecture and remote procedure calls (RPC), this makes the system distributed and lets us segregate the logic for machine learning and clients utilizing this model, giving rise to Service Oriented Architecture (SOA). The Sever serving the machine learning model is acting as an intermediate between the Client App developed using Node JS/Express JS and persisted machine learning model.

**The search engine** is implemented in the lines of **OKAPI BM25** which calculates the relevance score using this:

$$\text{score}(D,Q) = \sum_{i=1}^{n} \text{IDF}(q_i) \cdot \frac{f(q_i, D) \cdot (k_1 + 1)}{f(q_i, D) + k_1 \cdot \left(1 - b + b \cdot \frac{|D|}{\text{avgdl}}\right)},$$

Image reference: https://en.wikipedia.org/wiki/Okapi_BM25

Where $f(q_i, D)$ is the term frequency for $q_i$ in the document D and |D| is length of the document which is counted from the number of words it has. AVGDL is average document length which is total length of all documents in words divided by number of documents. K1 and b are constants.
The Inverse document frequency here is calculated as:

$$\text{IDF}(q_i) = \log \frac{N - n(q_i) + 0.5}{n(q_i) + 0.5},$$

Image reference: https://en.wikipedia.org/wiki/Okapi_BM25

Where N is the number of documents in a particular collection and the n(qi) is number of documents containing qi.
After calculating the relevance score for documents containing query terms, the document is pushed into a resulting array which is sorted in descending order of relevance at the client side.

**DATA:**
The data obtained for the purpose of this application was extracted from the RIT Reddit discussion board(https://www.reddit.com/r/rit/). Each of the thread title and its corresponding description forms a single document for this application. One thousand such documents were scraped using a web browser automation tool selenium. The scraped data is stored in a tab delimited csv file so that it can be loaded into the backend system to perform natural language processing as well as clustering.

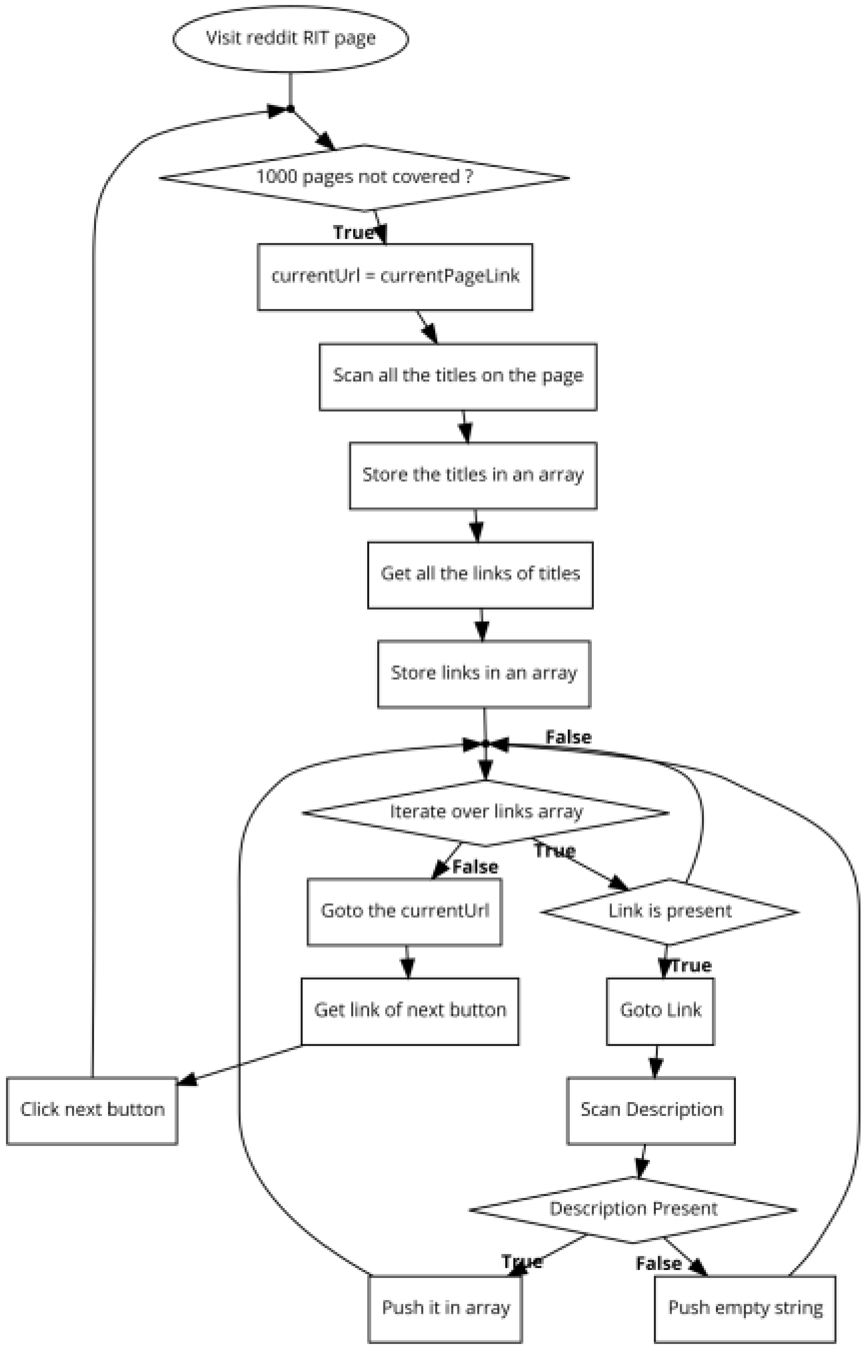


FIG: Flow diagram of data scraping

**BUSINESS LOGIC:**
The architecture is divided into client and server. The client here is the front-facing web application developed in Node Js and the server is Python code which runs the persisted machine learning model. The Python server serving the front-facing web application is binded on a port in a particular hostname. The client app when instantiates first calls that Python server using RPC. The server when gets the request, returns the generated machine learning model to the client that is web-app which displays the top terms at the front-end. The entire business logic can be viewed here under:

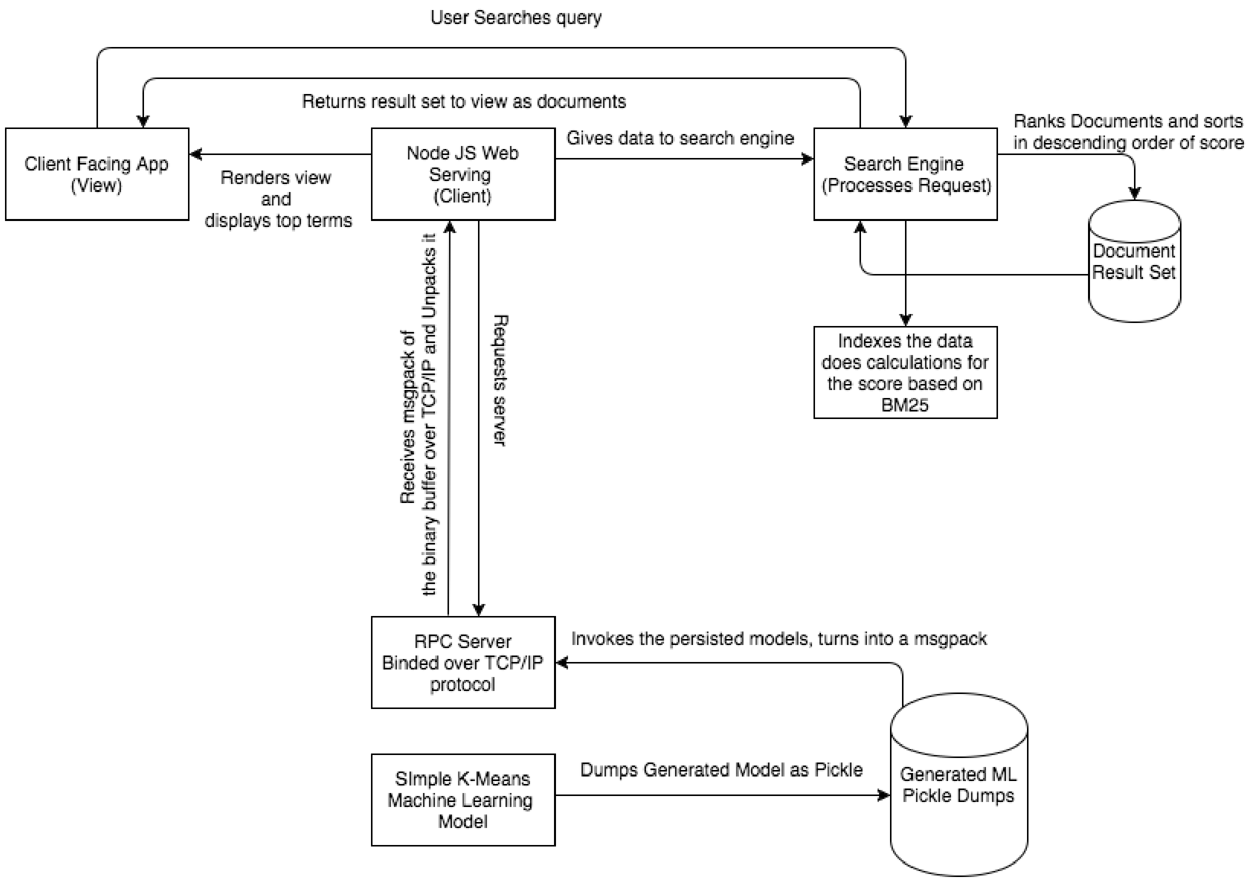

User Searches query
Returns result set to view as documents
Client Facing App (View)
Renders view and displays top terms
Node JS Web Serving (Client)
Gives data to search engine
Search Engine (Processes Request)
Ranks Documents and sorts in descending order of score
Document Result Set
Indexes the data does calculations for the score based on BM25
Receives msgpack of the binary buffer over TCP/IP and Unpacks it
Requests server
RPC Server Binded over TCP/IP protocol
Invokes the persisted models, turns into a msgpack
SImple K-Means Machine Learning Model
Dumps Generated Model as Pickle
Generated ML Pickle Dumps

FIG: Business logic of community information engine

The way this search engine works is, whenever a user types a query in the search box, the client application passes the request to the searcher class. When the request is received at the search engine, it calculates the relevance score of the document for the query terms and ranks them in descending order of their relevance score. This algorithm can implement the concept of single, bi-word and phrase query searches.

**USER INTERFACE:**
The front end of the application was developed using HTML, CSS, Javascript/JQuery and grid framework (twitter bootstrap). The main page of our GUI has a search box and a section displaying the most talked about terms at RIT.

The font size and the color of each term at the most talked about section dynamically vary as per their frequency of occurrence in the underlying documents. The words having the highest frequency appear big and bold in size compared to the rest and so on. Our UI also has an embedded RIT video, thereby enhancing the user experience of our application.

Whenever a term is searched for, the user is directed to a different page displaying all the relevant result one below the other in an array format. The screenshots of our UI and search results is as follows:

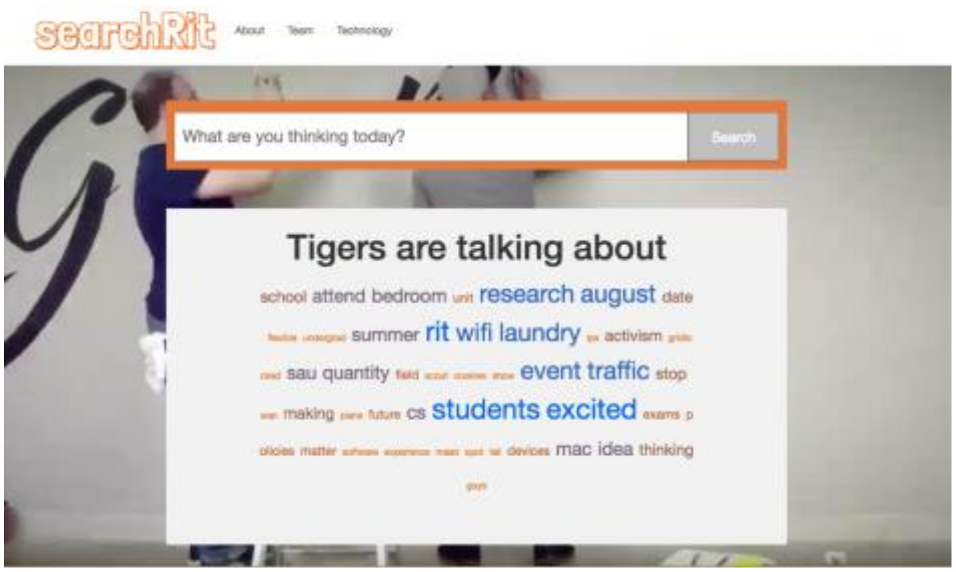
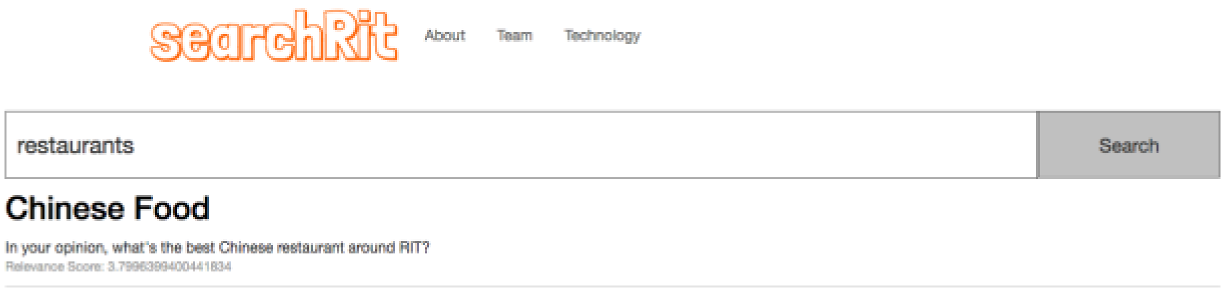

searchRit
About
Team
Technology
What are you thinking today?
Search
Tigers are talking about
school attend bedroom unit research august date
summer rit wifi laundry activism
sau quantity field event traffic stop
making future cs students excited exams
matter devices mac idea thinking

FIG: Entry page of the website



restaurants

Chinese Food

In your opinion, what's the best Chinese restaurant around RIT?
Relevance Score: 3.7996399400441834

FIG: Search results

**PACKAGE AND FILE INFORMATION:**

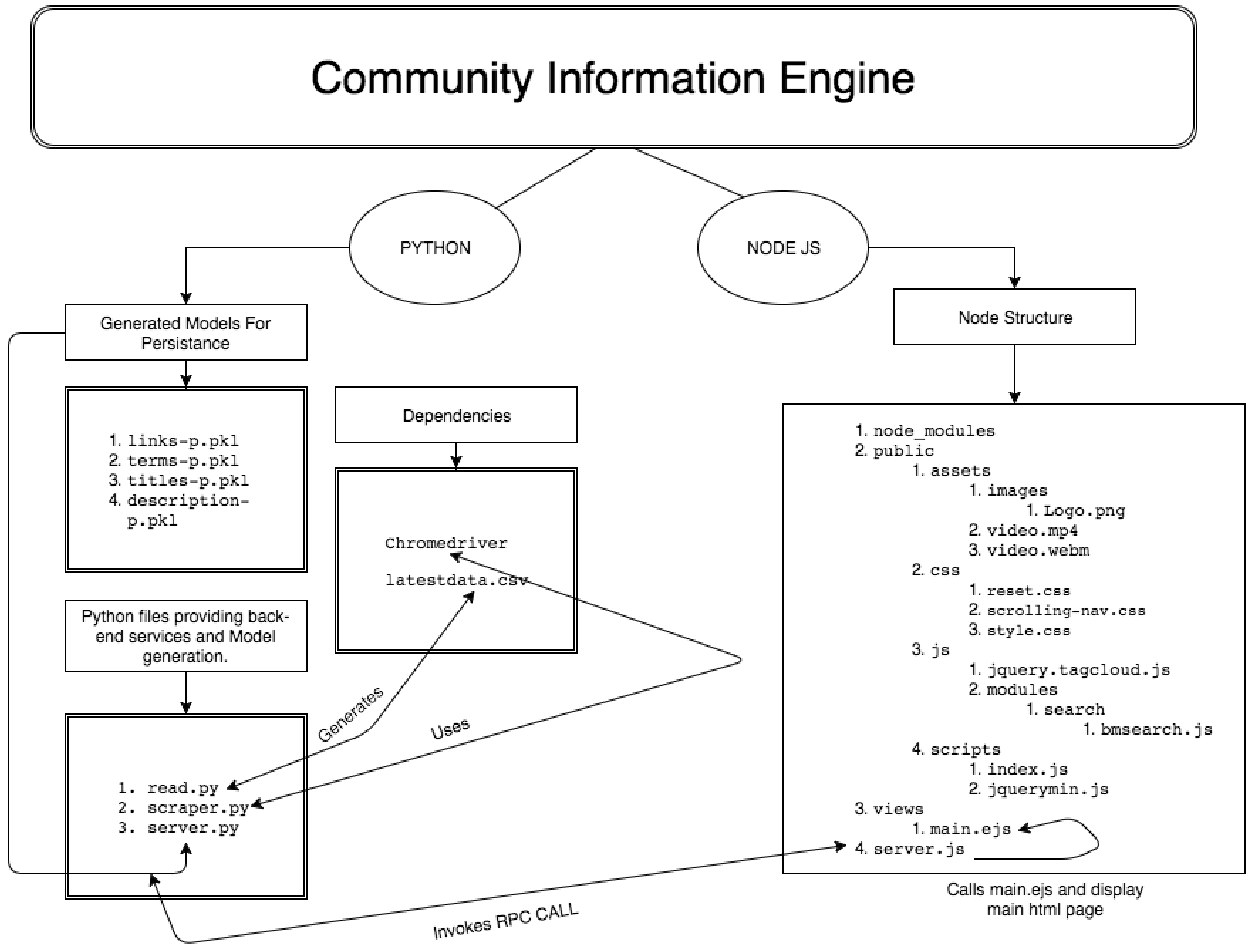


FIG: Package and file information

**REFERENCES:**

1. http://scikit-learn.org/stable/modules/clustering.html#clustering
2. http://scikit-learn.org/stable/modules/clustering.html#k-means
3. https://en.wikipedia.org/wiki/Okapi_BM25
4. https://xapian.org/docs/bm25.html
5. https://nlp.stanford.edu/IR-book/html/htmledition/okapi-bm25-a-non-binary-model-1.html
6. http://opensourceconnections.com/blog/2015/10/16/bm25-the-next-generation-of-lucene-relevation/
7. http://scikit-learn.org/stable/auto_examples/text/document_clustering.html
8. https://www.burakkanber.com/blog/machine-learning-in-other-languages-introduction/
9. http://brandonrose.org/clustering