EECS 767

INFORMATION RETRIEVAL

Progress Report

By: FiniteLoop Squad

Ron Andrews, Nidhi Midha, Blake Bryant

Table of Contents

[1. Introduction 3](#_Toc508444053)

[2. Programming Platform and Version Control Selection 4](#_Toc508444054)

[3. Data Structures 4](#_Toc508444055)

[3.1. Pre-Processing to Processing Interface Data Structures 4](#_Toc508444056)

[3.2. Processing to Query Processing Interfaces Data Structures 5](#_Toc508444057)

[3.3. Query Processing to HMI Data Structures 6](#_Toc508444058)

[4. Ingest 6](#_Toc508444059)

[5. Preprocessing 7](#_Toc508444060)

[5.1. Tokenization 7](#_Toc508444061)

[5.2. Indexing 7](#_Toc508444062)

[6. Processing 8](#_Toc508444063)

[7. Query 8](#_Toc508444064)

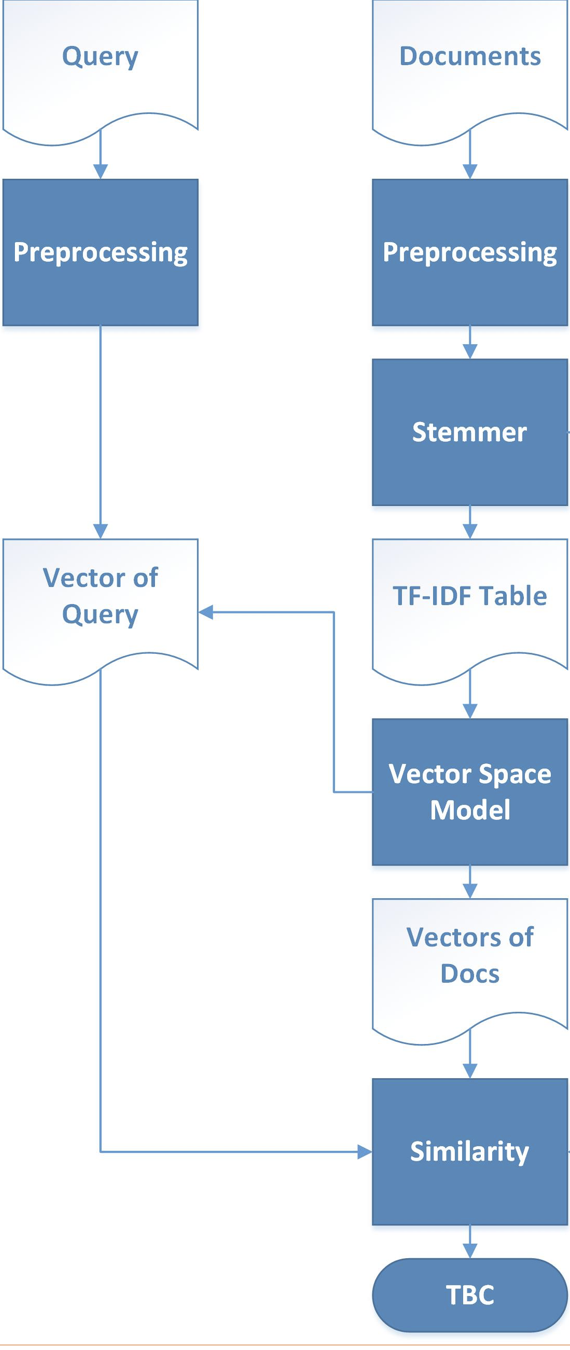
[8. Appendix 9](#_Toc508444065)

[8.1. Group Log 9](#_Toc508444066)

# Introduction

Our project goal is to create a simple Information Retrieval System (Search Engine) using Vector Space Model. We would add optimization techniques such as term proximity, Relevance Feedback, and test the product on a practical data set obtained by crawling a specific domain with Niche Crawler.

In phase 1, we have finished the implementation of document processing, inverted index and vector space model. We are currently working over adding term proximity as a consideration of displaying the results. Figure, below, shows our current progress.



# Programming Platform and Version Control Selection

In review of the various programming options available, we focused on those languages which were most capable, natively for this project. Specifically, looking at those languages which supported complex functions such as cosine similarity, web compatibility (Common Gateway Interface, CGI, or apache server module based), and of course, familiarity. After consideration of various options, such as *R*, *Perl*, *C++*, and *Python*, we selected *Python* as our language of choice.

With the current versions available for *Python*, we initially selected to go with version 3.6, being the latest available. As we worked through the various modules of our search engine (pre-processing, processing, query, and human machine interface (HMI), we ran into a few challenges. The Natural Language Toolkit (NLTK) that we selected to facilitate the stop list and lemmer was compatible with *Python* 3.5, not 3.6. Additionally, the Electrical Engineering and Computer Science (EECS) student web server currently provides access to *Python* 2.7 and 3.5. Our conclusion was to go forward with Python 2.7 as it was common to our individual environments as well as the web server. Additionally, we elected to use the EECS web server CGI capability for hosting our search engine.

For our collaboration environment, we set up a GitHub repository specifically for our **FiniteLoop** **Squad** to work and share. In the environment, we are able to coordinate our code development efforts as well as documentation.

# Data Structures

In order to pass the data structures between our modules, we are leveraging a *Python* module called *shelve*. This native module enables us to pass the raw data structures by way of a binary file stored on the file server. The following sub-sections provide the data structures passed between the modules.

# Pre-Processing to Processing Interface Data Structures

Pre-Processing provides three data structures to the Processing module:

1. Document Key Matrix
2. Term Incidence Matrix, with Frequency
3. Term Proximity Matrix

The Document Key provides a listing of the details for each document as a *Python* dictionary where the document name is the key and the details is a list of values. The details for each document include the document identifier and current location of the document. The order of the list of documents matches the order of the documents in the Term Incidence Matrix.

The generalized data structure looks like the following, in *Python* terms:

doc\_key = [

{ DocName1: [DocID1, DocLocation1] },

{ DocName2: [DocID2, DocLocation2] },

…,

{ DocNamen: [DocIDn, DocLocationn] }

]

The Term Incidence Matrix provides a listing of each term and its occurrence in the corpus as a *Python* dictionary where the term is the key and the document incidence with frequency is a list. Each document incidence list is aligned in order with the list provided in the doc\_key data structure.

The generalized data structure looks like the following, in *Python* terms:

index = [

{ Term1: [tf1, tf2, …, tfn] },

{ Term2: [tf1, tf2, …, tfn] },

…,

{ Termm: [tf1, tf2, …, tfn] }

]

The Term Proximity Matrix provides a dictionary of each term, as the key, and a list of tuples as the value. The tuples identify the document and offset from the beginning of the document. Offsets are based on word distance from the beginning of the document after the tokenization and stop word parsing is complete.

The generalized data structure looks like the following, in *Python* terms:

proximity = {

Term1: [ [DocID, Prox], [DocID, Prox], … [DocID, Prox] ],

Term2: [ [DocID, Prox], [DocID, Prox], … [DocID, Prox] ],

…,

Termm: [ [DocID, Prox], [DocID, Prox], … [DocID, Prox] ]

}

# Processing to Query Processing Interfaces Data Structures

Processing provides three data structures to the Query Processing module:

1. Document Key Matrix
2. Normalized Vector Space Model (VSM)
3. Term Proximity Matrix (InWork)
4. Term Index Look-Up Dictionary (InWork)

The Document Key Matrix is forwarded, unaltered from what was received from the Pre-Processing module.

The Normalized VSM provides an alphabetically sorted list of vectors (lists). Each vector is in order as identified by the Term Index Look-Up Dictionary and each vector is in order of the Document Key Matrix. The vectors provide the normalized Term Frequency – Inverted Data Frequency (TF-IDF) weight of the term for each document.

The generalized data structure looks like the following, in *Python* terms:

docVector = [

[WT1,D1, WT1,D2, …, WT1,Dn],

[WT2,D1, WT2,D2, …, WT2,Dn],

…

[WTm,D1, WTm,D2, …, WTm,Dn]

]

The Term Proximity Matrix provides a listing similar to the Normalized VSM where instead of the weights, it provides a list of the proximities.

The generalized data structure looks like the following, in *Python* terms:

proxVector = [

[ [P1T1,D1, P2T1,D1, …, P*i*T1D1], [P1T1,D2, P2T1,D2, …, P*i*T1D2], …, [P1T1,D*n*, P2T1,D*n*, …, P*i*T1Dn] ],

[ [P1T2,D1, P2T2,D1, …, P*i*T2D1], [P1T2,D2, P2T2,D2, …, P*i*T2D2], …, [P1T2,D*n*, P2T2,D*n*, …, P*i*T2Dn] ],

…,

[ [P1T*m*,D1, P2T*m*,D1, …, P*i*T*m*D1], [P1T*m*,D2, P2T*m*,D2, …, P*i*T*m*D2], …, [P1T*m*,D*n*, P2T*m*,D*n*, …, P*i*T*m*Dn] ]

]

The Term Index Look-Up Dictionary provides a dictionary of each term, as the key, and an index into the Normalized VSM and Term Proximity Matrix lists for efficient look-up of the vectors needed in searching for results based on a query.

The generalized data structure looks like the following, in *Python* terms:

termIndex = {

Term1: i1,

Term2: i2,

…,

Term*m*: i*m*

}

# Query Processing to HMI Data Structures

The data passed to the HMI is done so directly with the CGI script importing the Query module directly for dynamic processing and results. The Query module provides the following data and data structures to the HMI for display to the user:

* Total Number of Results found
* Time taken to process query and return results
* List of results, ordered by relevance

The list of results is provided to the HMI as a list of document entries. Each entry contains a list of parameters to display to the user.

The generalized data structure looks like the following, in *Python* terms:

results = [

[ DocName1, DocLocation1, Rank1, Summary1],

[ DocName2, DocLocation2, Rank2, Summary2],

…,

[ DocName*n*, DocLocation*n*, Rank*n*, Summary*n*]

]

# Ingest

Ingestion is performed by functions contained within the ingest.py file which operate on files stored within a local directory. The path to the directory is currently configured with a static path hard coded within the ingestion function. A function was created to allow for user entry of a different path location, however this functionality is currently deemed unnecessary and has been disabled via inline comments.

The native python library “urllib2” is used to read files and provide compatibility for processing various document formats including html. Each document within the specified directory is loaded into memory as a stream of raw characters and stored in an array, called “data,” with each document representing a single index within the array.

A dictionary called “doc\_key” is created to store the filename, document ID and file path for each document ingested. The document filename is used as the key for the dictionary while the value for each key is represented as an array containing the document ID and file path.

# Preprocessing

# Tokenization

Each document, stored as an index within the data array created during ingestion, is processed by the function “func\_tokenize” within the ingest.py file. The Python Natural Language Tool Kit (NLTK) (<https://www.nltk.org/>) is used to create stop word lists and a stemming function within func\_tokenize. This requires the installation of NLTK as well as downloading the stop word list prior to running the ingestion program. NLTK version 3.2.5 (the latest version as of this report) requires either Python version 2.7 or 3.5. Python 3.5 is not the most current release of Python 3 and may require some systems to install an earlier version of Python.

Preprocessing within func\_tokenize consists of 4 steps executed on each index of the data array:

1. HTML tags are removed from the data stream via regular expression pattern matching.
2. The data stream is converted to lower case, punctuation is removed, and the stream is split into tokens via the Python string.split() method.
3. Stop words contained within the NLTK stop word list are removed from the list of tokens.
4. Each token within the data stream is processed by the Porter stemmer provided by the NLTK library. There is a known issue when processing

Unicode characters with the NLTK Porter stemmer. Because of this, words containing Unicode characters are currently dropped in the stemming phase.

# Indexing

The number of documents processed during ingestion is used to determine the dimensionality of arrays within the terms dictionary. Each data stream, corresponding with a separate document, is parsed for unique terms. If a new term is discovered in a document, the term is added as a key to the dictionary called “terms” with an initial value of an array with multiple indices corresponding with the number of documents processed. The index corresponding to the document, wherein the term was observed, is then incremented by 1. Subsequent observations of the term within the same document, or future documents, results in incrementing the value stored in the term array at the index corresponding with the id of the document being parsed. This results in an array reflecting the term frequency for each term observed within each document.

Concurrently, an additional dictionary called “proximity” is created to record positional data pertaining to each occurrence of a term within the documents parsed. Similar to the “terms” dictionary, unique terms are used as key values in the dictionary. Values within the “proximity” dictionary are represented as an array containing tuples consisting of the document ID and the position the term appeared within the document. A new tuple representing the document id and term position is appended to the term array each time the term is observed within the data stream. This data structure is described in further detail within the “Data Structures” section of this report.

Finally, data is exported to a file called “ingestOutput.db” using the Python shelve library. The “terms” dictionary is exported as “index”, the “doc\_key” dictionary is exported as “doc\_key” and the “proximity” dictionary is exported as “proximity” within the output file.

# Processing

The processing module reads in the output file from the ingest function in order to acquire the pre-processing data structures. It then generates the TF-IDF, normalizes the vectors and stores the VSM for the query module. Additionally, processing generates a proximity matrix, similar to the VSM data structure and also stores it, along with the doc\_key structure into the output shelve file for use by the query function.

To accomplish this, the index data structure (the term incidence matrix) is sorted and then walked through to evaluate for the Document Frequencies of each term in order to generate the VSM. It does so by first calculating the non-zero indices in the document arrays along with calculating the idf (log n/df), where n is the length of the array (# of documents). With this data, the non-normalized weights are calculated, |Wi| = sqrt(sum(idf2)), and then added to the VSM (docVector). Finally, the module goes through the VSM and normalizes the document vectors and stores the data structure in the shelve output file for the query function.

The generation of the proximity file is currently in work.

# Query

The dictionaries “doc\_vector” and “doc\_key” are passed to query module using a file processingOutput.db which is Python shelve library. “doc\_vector” dictionary provides the alphabetically sorted list of vectors (lists). Each vector is in order as identified by the Term Index Look-Up Dictionary and each vector is in order of the Document Key Matrix. The vectors provide the normalized Term Frequency – Inverted Data Frequency (TF-IDF) weight of the term for each document. Weights in this dictionary are compared with another vector “query\_vector”.

Search query inputted by user is first preprocessed by removing the stop words and processing by the Porter Stemmer provided by NLTK library. Once processed, query tokens are stored in the query vector, whose weights are compared with the document vectors to calculate the cosine similarity.The cosine similarities are calculated using the method similarity() which sorts the similarity, and ranks the results in descending order on the basis of doc\_key associated with each document vector. The ranks are stored in queryOutput.db, which would be passed to CGI to display the results.

Current work which is going on includes getting the proxVector from processing module, and re-rank the top 10 documents obtained from cosine similarity, on the basis of proximities.

# Appendix

# Group Log

* 29 Jan 2018: First meeting between Nidhi and Ron - discussed separation of functionality and preferred coding language for project. Reviewed language alternatives (java, python, R, etc.) - looking for the language with the most to offer natively
* 01 Feb 2018: Notified professor on formation of group
* 05 Feb 2018: First meeting of group as three members, discussed approach to project, discussed initial outline of code (draft created by Blake). Coordinated module responsibilities and discussed I/O between modules. Selected Python for language. Started GitHub repository for collaboration environment
* <insert entries from GitHub log here>
* 15 Feb 2018: Notified professor of additional team member and team name: FiniteLoop Squad
* <insert entries from GitHub log here>
* 19 Feb 2018: Met to review initial code stubs and discuss next steps - working to complete simple search to reflect (and be able to check against) steps outlined in class quiz (taking in simple docs and matching with query)
* <insert entries from GitHub log here>
* 26 Feb 2018: Met to go over ingest functionality and output - need to modify processing to accept minor change in ingest output format. Discussed front end GUI - coordinated CGI script and environment on EECS *peoples* server
* <insert entries from GitHub log here>
* 5 Mar 2018L Met to go over current status, the progress report and functionality. Discussed implementation for term proximity ranking and feedback relevance. Team discussed data structures to pass for proximity and how feedback relevance should work in our search engine. Also assigned sections for working on the progress report.