EECS 767

INFORMATION RETRIEVAL

Progress Report

By:

FiniteLoop Squad

Ron Andrews, Nidhi Midha, Blake Bryant

Table of Contents

[Abstract 3](#_Toc508177227)

[Software Design 3](#_Toc508177228)

[Programming Platform and Version Control Selection 3](#_Toc508177229)

[Data Structures 4](#_Toc508177230)

[Pre-Processing to Post-Processing Interface Data Structures 4](#_Toc508177231)

[Post-Processing to Query Processing Interfaces Data Structures 5](#_Toc508177232)

[Query Processing to HMI Data Structures 5](#_Toc508177233)

[Ingest 6](#_Toc508177234)

[Preprocessing 6](#_Toc508177235)

[Post-Processing 6](#_Toc508177236)

[Query 6](#_Toc508177237)

[Appendix 6](#_Toc508177238)

[Group Log 6](#_Toc508177239)

# Abstract

<nidhi>

# Software Design

sdsd

# 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, post-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 Post-Processing Interface Data Structures

Pre-Processing provides three data structures to the Post-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] },

…,

{ DocName*n*: [DocID*n*, DocLocation*n*] }

]

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: [t1,f, t2,f, …, t*n*f] },

{ Term2: [t1,f, t2,f, …, t*n*f] },

…,

{ Term*m*: [t1,f, t2,f, …, t*n*f] }

]

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:

term\_prox = {

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

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

…,

Term*m*: [ [DocID, Prox], [DocID, Prox], … [DocID, Prox] ]

}

## Post-Processing to Query Processing Interfaces Data Structures

Post-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, W T1,D2, …, W T1,D*n*],

[W T2,D1, W T2,D2, …, W T2,D*n*],

…

[W T*m*,D1, W T*m*,D2, …, W T*m*,D*n*]

]

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

<blake>

# Preprocessing

<blake>

# Post-Processing

<ron Andrews>

# Query

<nidhi>

Vectorization

Document Vectorization

# Appendix

## Group Log

* 29 Jan 2018: First meeting between Nadhi 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.