EECS 767

INFORMATION RETRIEVAL

Progress Report

By: FiniteLoop Squad

Ron Andrews, Nidhi Midha, Blake Bryant

Table of Contents

[1. Introduction 3](#_Toc508453004)

[2. Programming Platform and Version Control Selection 4](#_Toc508453005)

[3. Data Structures 4](#_Toc508453006)

[3.1. Pre-Processing to Processing Interface Data Structures 4](#_Toc508453007)

[3.2. Processing to Query Processing Interfaces Data Structures 5](#_Toc508453008)

[3.3. Query Processing to HMI Data Structures 6](#_Toc508453009)

[4. Ingest 6](#_Toc508453010)

[5. Preprocessing 7](#_Toc508453011)

[5.1. Tokenization 7](#_Toc508453012)

[5.2. Indexing 7](#_Toc508453013)

[6. Processing 8](#_Toc508453014)

[7. Query 8](#_Toc508453015)

[8. Appendix 9](#_Toc508453016)

[8.1. Group Log 9](#_Toc508453017)

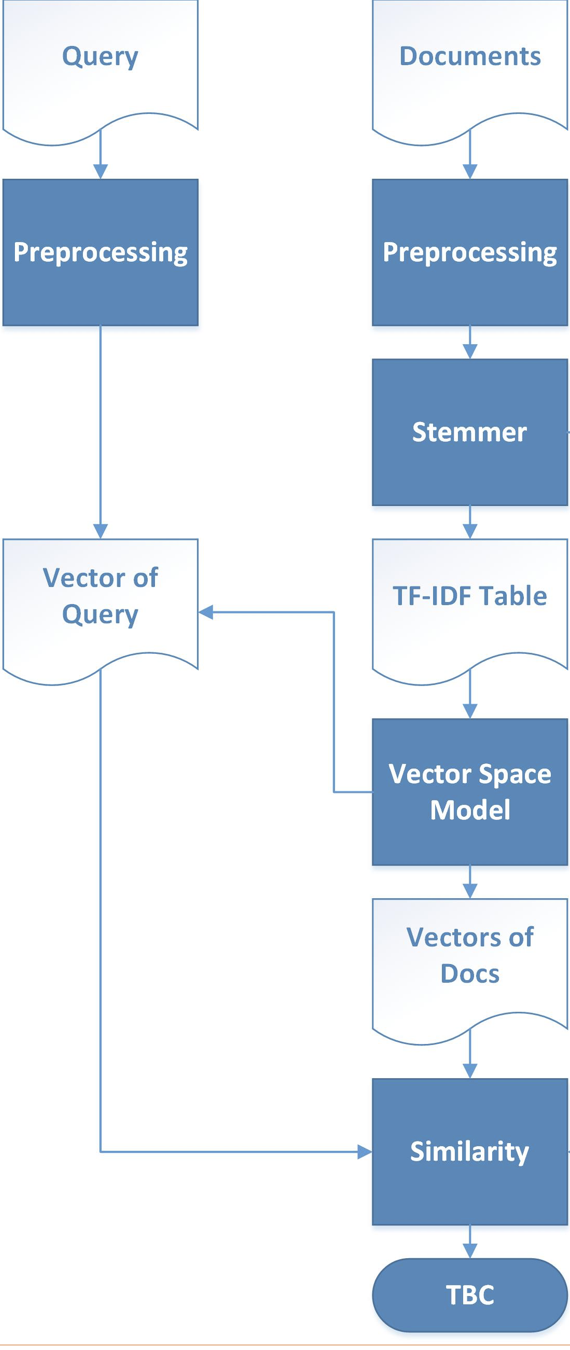
[8.2. GitHub Contributions 10](#_Toc508453018)

[8.3. GitHub ChangeLog 10](#_Toc508453019)

# 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. As a test we utilized a control group of data files, based on the class quiz covering VSM. The text data files are as follows:

test1.txt: Shipment of gold damaged in a fire

test2.txt: Delivery of silver arrived in a silver truck.

test3.txt: Shipment of gold arrived in a truck.

test4.txt: Truck arrive damaged.

# Pre-Processing to Processing Interface Data Structures

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

1. Document Key Matrix
2. Term Incidence Matrix, with Frequency
3. Term Proximity Matrix
4. Title mapping
5. Number of documents

The Document Key provides 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, current location on the local filesystem (cache), and the URL [[1]](#footnote-1)of the document. The document identifier provides the sorting order of the documents and is also the index in the Term Incidence Matrix.

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

doc\_key = {

|  |  |  |  |
| --- | --- | --- | --- |
| DocName1: | [DocID1, | DocLocation1, | DocURL1], |
| DocName2: | [DocID2, | DocLocation2, | DocURL2], |
| …, |  |  |  |
| DocNamen: | [DocIDn, | DocLocationn, | DocURLn] |

}

As a practical example, the contents of the **doc\_key** using the control group is:

doc\_key = {

|  |  |  |  |
| --- | --- | --- | --- |
| 'test1.txt': | [3, | '/EECS767/FiniteLoopSE/test1.txt', | 'no\_url'], |
| 'test3.txt': | [2, | '/EECS767/FiniteLoopSE/test3.txt', | 'no\_url'], |
| 'test4.txt': | [0, | '/EECS767/FiniteLoopSE/test4.txt', | 'no\_url'], |
| 'test2.txt': | [1, | '/EECS767/FiniteLoopSE/test2.txt', | 'no\_url'] |

}

The Term Incidence Matrix provides 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] } |

]

As a practical example, the contents of the index using the control group is:

Index = {

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 'truck': | [1, | 1, | 1, | 0], |
| 'arriv': | [1, | 1, | 1, | 0], |
| 'damag': | [1, | 0, | 0, | 1], |
| 'fire': | [0, | 0, | 0, | 1], |
| 'silver': | [0, | 2, | 0, | 0], |
| 'gold': | [0, | 0, | 1, | 1], |
| 'deliveri': | [0, | 1, | 0, | 0], |
| 'shipment': | [0, | 0, | 1, | 1] |

}

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] ] |

}

As a practical example, the contents of the proximity using the control group is:

defaultdict = {

|  |  |  |  |
| --- | --- | --- | --- |
| 'truck': | [ (0, 0), | (1, 4), | (2, 3) ], |
| 'arriv': | [ (0, 1), | (1, 2), | (2, 2) ], |
| 'damag': | [ (0, 2), | (3, 2) ], |  |
| 'fire': | [ (3, 3) ], |  |  |
| 'silver': | [ (1, 1), | (1, 3) ], |  |
| 'gold': | [ (2, 1), | (3, 1) ], |  |
| 'deliveri': | [ (1, 0) ], |  |  |
| 'shipment': | [ (2, 0), | (3, 0) ] |  |

}

The title map provides a dictionary of document names and document titles, specifically for HTML pages by leveraging the contents of the title[[2]](#footnote-2). This information is metadata used to provide ‘summary’ text displayed in the results of the search.cgi script.The generalized data structure looks like the following, in *Python* terms:

title\_map = {

|  |  |
| --- | --- |
| DocName1: | DocTitle1, |
| DocName2: | DocTitle2, |
| …, |  |
| DocNamen: | DocTitlen, |

}

As the control group does not contain any html pages, a relative example is not provided and left to the reader to imagine.

The number of docs is provided to ensure that there is a check between ingest and processing that the number of documents expected is there number of documents processed. This variable is simply an integer stored in the database file:

num\_docs = x

As in the case of the control group, the database reflects:

num\_docs = 4

# Processing to Query Processing Interfaces Data Structures

Processing provides the following data structures to the Query Processing module:

1. Document Key Matrix
2. Term Proximity Matrix
3. Term Index Look-Up Dictionary

Additionally, the processing module archives the normalized Vector Space Model (VSM) in a processingArtifacts.db database file.

1. Normalized Vector Space Model (VSM)

The Document Key Matrix is forwarded, unaltered from what was received from the Pre-Processing module, see Section 3.1, Pre-Processing to Processing Interface Data Structures.

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]

]

As a practical example, the contents of the proximity using the control group is:

docVector = [

[0.356, 0.858, 0.0, 0.0, 0.0, 0.0, 0.0, 0.356],

[0.118, 0.0, 0.57, 0.0, 0.0, 0.0, 0.805, 0.118],

[0.27, 0.0, 0.0, 0.0, 0.65, 0.65, 0.0, 0.27],

[0.0, 0.378, 0.0, 0.755, 0.378, 0.378, 0.0, 0.0]

]

The docVector, is used to create a loop up table for the query module (envoked by search.cgi) to quickly locate terms from the query and perform the cosine similarity process. This resulting data structure is a dictionary of terms such that each term contains the document arrays of normalized weights for that term.

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

termDict = {

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

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

…,

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

termDict

---------------------

{'truck': [0.356, 0.118, 0.27, 0.0], 'arriv': [0.356, 0.118, 0.27, 0.0], 'damag': [0.858, 0.0, 0.0, 0.378], 'fire': [0.0, 0.0, 0.0, 0.755], 'silver': [0.0, 0.805, 0.0, 0.0], 'gold': [0.0, 0.0, 0.65, 0.378], 'deliveri': [0.0, 0.57, 0.0, 0.0], 'shipment': [0.0, 0.0, 0.65, 0.378]}

title\_map

---------------------

{}

doc\_key

---------------------

[{'test4.txt': [0, '/home/terrapin/EECS767/FiniteLoopSE/test4.txt', 'no\_url']}, {'test2.txt': [1, '/home/terrapin/EECS767/FiniteLoopSE/test2.txt', 'no\_url']}, {'test3.txt': [2, '/home/terrapin/EECS767/FiniteLoopSE/test3.txt', 'no\_url']}, {'test1.txt': [3, '/home/terrapin/EECS767/FiniteLoopSE/test1.txt', 'no\_url']}]

proxVector

---------------------

[[[1], [2], [], [], [], [], [], [0]], [[2], [], [0], [], [], [], [1, 3], [4]], [[2], [], [], [], [1], [0], [], [3]], [[], [2], [], [3], [1], [0], [], []]]

termIndex

---------------------

{'truck': [0.125, 7], 'arriv': [0.125, 0], 'damag': [0.301, 1], 'fire': [0.602, 3], 'silver': [0.602, 6], 'gold': [0.301, 4], 'deliveri': [0.602, 2], 'shipment': [0.301, 5]}

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],

…,

[ DocNamen, DocLocationn, Rankn, Summaryn]

]

# 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 proximity file provided by ingest contains a dictionary of tuples where each tuple indicates the term and offset.

|  |  |
| --- | --- |
| n - # of terms, m = # of documents, p = # of term occurrences | |
| ***Prepare index, document key, and proximity key from ingest*** | **O(n log n + 2n + m)**  **=~ O(n log n)** |
| *sortedTerms = sorted(list of dictionary keys)* | O(n log n)[[3]](#footnote-3) |
| *sortedTermIndex = array of term hashes based on sortedTerms* | O(n) |
|  |  |
| *sortedDocs = array of document hashes based on doc index id* | O(m) |
|  |  |
| *sortedProximity = array of term prox hashes based on sortedTerms* | O(n) |
|  |  |
| ***Generate TF-IDF*** | **O(n x 2m)** |
| *For each word in termIndex* | O(n) |
| *DF = sum of all non-zero indices in the doc array* | O(m) |
| *Calculate IDF* |  |
| *Calculate weights for word in each document* | O(m) |
|  |  |
| ***Normalize Vectors*** | **O(n x m)** |
| *Create doc length (unit) for each document* | O(m) |
| *For each word in termIndex* | O(n) |
| *Normalize each weight for each document* | O(m) |
|  |  |
| ***Process proximity*** | **O(m x p)** |
| *For each document* | O(m) |
| *For each term occurrence (tuple)* | O(p) |
| *Process tuple and append to term index array* | O(1) |
|  |  |
| Totals  O(n log n) + O(n x 2m) + O(n x m) + O(m x p)  O(n log n) + O(n x m) + O(m x p) | O(nm) or O(mc)  Where c is # of total words in corpus |

# 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

# Manifest & Installation

The FiniteLoop Squad Search Engine consists of the following manifest:

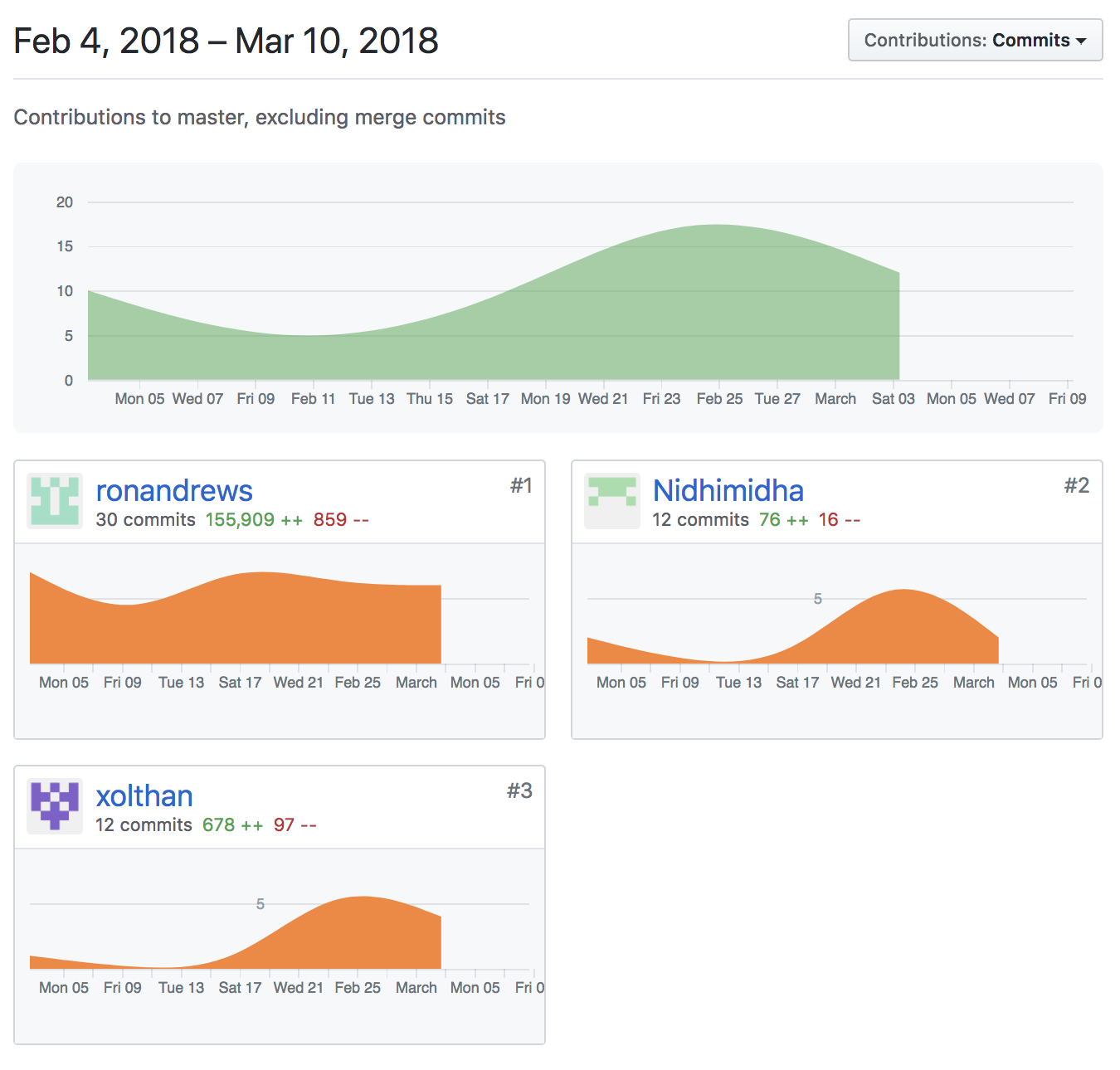
* **FiniteLoopSquad**[[4]](#footnote-4)
  + **cached\_docs** – directory where pages are downloaded to by the niche web crawler
  + **cgi-bin** – directory containing the cgi script
    - *search.cgi* – cgi (common gateway interface) script used to host search engine parsing of the query against the corpus
  + *index.html* – base page for the FiniteLoop Squad Search Engine – passes query to search.cgi
  + *Makefile* – makefile script for setting up and executing the offline components as well as configuring for web based access
  + **OUTPUT** – directory for storing shelve data structure files to make available between modules
    - *ingestOutput.db* – shelve data structure output by ingest.py
    - *processingOutput.db* – shelve data structure output by processing.py
    - *queryOutput.db* – shelve data structure output by query.py
  + *src* – directory containing all of the source code (excluding the cgi script
    - *ingest.py* – python script for ingesting source files, outputs ingestOutput.db for the processing module
    - *processing.py* – python script for processing data structures from ingest, output processingOutput.db for the query module
    - *query.py* – python script for searching corpus using the output from the processing module
    - *seeShelve.py* – python script for printing out the contents of the shelve data structure files, parses all db files located in the OUTPUT folder

The Makefile provides the necessary functions to ensure that the environment will work correctly, provided the user is in a Linux-based environement running python 3.5. The FiniteLoopSE search engine was developed to be executed in the EECS environments and run in a users student web page area via the *people.eecs.ku.edu* web hosting. Due to the limitations on cycle servers being able to access external pages, other than a few git sites, we ran the niche web crawler on a student workstation within the ITTC domain and transfer the resulting cached documents to the EECS domain for ingest and processing.

# 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
* 15 Feb 2018: Notified professor of additional team member and team name: FiniteLoop Squad
* 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)
* 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
* 5 Mar 2018: 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.

# GitHub Contributions



# GitHub ChangeLog

commit a721d2f522ca97e7d36157a094b08e7b02bf5e65 (HEAD -> master, origin/master, origin/HEAD)

Author: Ron <ron@merehuman.net>

Added termIndex and proxVector to Processing

commit a255d61ad5aafa0064315d40929fc10441d4ede4

Merge: be5e8be 46e6696

Author: Ron <ron@merehuman.net>

Merge branch 'master' of https://github.com/ronandrews/eecs767

commit be5e8be73360f6f19ece98b344fcf860a3a0529c

Author: Ron <ron@merehuman.net>

Formatting, seeShelve, and Template

commit 46e669667f0b0d930e7bf426352619cb4bd6dfeb

Author: Nidhimidha <36167384+Nidhimidha@users.noreply.github.com>

Add files via upload

commit e89295ef9d3cf1612a262c140aa077fa2d5f5f20

Author: Ron <ron@merehuman.net>

Updated Report - Processing

commit a366aed0cacadf00fe280893448b8b52b18324ed

Author: Nidhimidha <36167384+Nidhimidha@users.noreply.github.com>

Add files via upload

commit 2781573d1709f8f1e926c46282e5ae82c820fa7f

Author: xolthan <xolthan@gmail.com>

Blake updated ingest and preprocessing sections

commit 85734d01e8da44db0aa2eef0829945d07ea80791

Author: xolthan <xolthan@gmail.com>

Now exports data via shelve to ingestOutput.db

commit 5de02bf3a740bec963c999ce34cf7e9e0260b188

Author: Ron <ron@merehuman.net>

Added Data Structures Info

commit 745d3dd3c2a64ec7d6e67b730c0b62c764bfce28

Author: Ron <ron@merehuman.net>

Notes in the template

commit 87ecf7b5e105b182f87f98a9dcd47e3cb10bb124

Merge: 53280a4 62732de

Author: Ron <ron@merehuman.net>

Merge branch 'master' of https://github.com/ronandrews/eecs767

commit 53280a435c4ea6f231fb5c72de19b0d7530e80f0

Author: Ron <ron@merehuman.net>

Update Assignments in Report

commit 62732de61d23c605b2742dda99f41529a1cb379b

Author: xolthan <xolthan@gmail.com>

deals with unicode exception and added proximity

commit fc78449ddb2e071a8c4b2a14166fa6e02c5d70d2

Author: xolthan <xolthan@gmail.com>

index.py rewritten for Python 2.7

commit b307fc441c18720e717c93a0f17519f2969d43da

Author: xolthan <xolthan@gmail.com>

Removed decoding in Ascii

commit 0d88284104221772834c420fd77dd9c245fe18e9

Author: xolthan <xolthan@gmail.com>

Add files via upload

commit 6b13d0628fb0d142247cf87bf88cc08f5c4a2cc7

Author: Ron Andrews <35875316+ronandrews@users.noreply.github.com>

Removing Duplicate file

commit a3a4db30ac695daa923e88b95070eeedd522000c

Author: Ron <ron@merehuman.net>

Updated Processing to Match Input format

commit 5c48fb0d503168a8e1677d2788e09b640dcce0f5

Author: Ron Andrews <35875316+ronandrews@users.noreply.github.com>

26 Feb Meeting Update

commit 16b72200d3828d5a791ee3fd31d2fca605a6ae62

Author: xolthan <xolthan@gmail.com>

Add files via upload

commit c8f8d6be309c92574f8fe2d60ed66398be24dc29

Author: xolthan <xolthan@gmail.com>

Add files via upload

commit 60969adee0988b86a7b4168fd6a7f84cfe5018f1

Author: Ron <ron@merehuman.net>

d

commit aa1a3f14db9555106af3fd58fb7dc024b9049b9e

Author: Ron <ron@merehuman.net>

oops

commit 408df3b350596ed86c3ac2bbd808891219001df3

Author: Ron <ron@merehuman.net>

updates

commit 441578f442f55a74ed745b06bb4cfb32193dea70

Author: xolthan <xolthan@gmail.com>

Add files via upload

commit 27e9f904aad15f3daa5c8d5343bd44ad8a256f57

Author: Nidhimidha <36167384+Nidhimidha@users.noreply.github.com>

Update query.py

commit 70dc48c19ebc57532a794520265560864acc02df

Author: xolthan <xolthan@gmail.com>

Add files via upload

commit c0a95246cc14d60d038cabb23433195e1514e623

Author: xolthan <xolthan@gmail.com>

Add files via upload

commit 7cdad81e5c0197e0ea2048008b64d50e02a30679

Author: Nidhimidha <36167384+Nidhimidha@users.noreply.github.com>

Update query.py

commit 6b118fef8efb19fd4e274393ae7f09bcef9dae78

Author: Nidhimidha <36167384+Nidhimidha@users.noreply.github.com>

Update query.py

commit e11905626a621d97c32532b2243d9c220aad0351

Author: Nidhimidha <36167384+Nidhimidha@users.noreply.github.com>

Add files via upload

commit c254401baee72274194b53c7563ea6c658203231

Author: Nidhimidha <36167384+Nidhimidha@users.noreply.github.com>

Update query.py

commit afbe2f079b4fc71d545ca1b98b51e68a7bb8b71d

Author: Nidhimidha <36167384+Nidhimidha@users.noreply.github.com>

Delete FinalReport\_v1.docx

commit 7c9b6e45d96b064a7f162c57fc02d7e762e4a2b5

Author: Nidhimidha <36167384+Nidhimidha@users.noreply.github.com>

Create FinalReport\_v1.docx

commit ab28e88c5757c0cc61c85cc98911ffc1a7b52a33

Author: Nidhimidha <36167384+Nidhimidha@users.noreply.github.com>

Update query.py

commit 1a6744815b08a3d570376f029e9e7d144e28c171

Author: Ron Andrews <35875316+ronandrews@users.noreply.github.com>

Create GroupLog file

commit 48d89594777dcc0a660d01200ecb0ced442a648f

Author: Ron <ron@merehuman.net>

Updated processing output

commit 26c0046b5d285063e789cc7f211b76dc5d005062

Author: Ron <ron@merehuman.net>

Changed to LL

commit ca30d784f2d87a30f2941038f2293eba99894bfb

Author: Ron Andrews <35875316+ronandrews@users.noreply.github.com>

Minor update - some notes

commit 7c6fdd439365195f1430fc904b566836a1e14155

Author: Ron Andrews <35875316+ronandrews@users.noreply.github.com>

Added text from blackboard assignment

commit a67949ab63086b16805aeb97f96d26082b135140

Author: Ron Andrews <35875316+ronandrews@users.noreply.github.com>

oops - useless

commit 3f07b2589313ba430107c2b9124f86968513553e

Author: Ron Andrews <35875316+ronandrews@users.noreply.github.com>

oops - useless....

commit 51c067a63eab2f2b0ee008ff2af360346b2f676f

Author: Ron <ron@merehuman.net>

Added initial CGI script

commit c8b9469b4cc84b07a749ff70bd25fd5c1ea65506

Author: Ron <ron@merehuman.net>

Ported processing to python3

commit 222ff5ed7e47bdda4f1f5fb1c911e0f54c30a8a0

Author: Ron <ron@merehuman.net>

Expanded the ingest files

commit 5b2102a07c6c5730308adf68b816b5574edf2ffa

Author: Ron <ron@merehuman.net>

Adding docsnew.zip

commit e960ae2a0c677cb42444fbde866390a7f1ff7f6e

Author: Ron <ron@merehuman.net>

Processing Update

commit a18ee3be8edb86e7fba1caa9db153bbd602c69e9

Author: Ron <ron@merehuman.net>

Processing Update

commit 4bf8f08391b78c07a08fa14a74b69726c1e950fc

Author: Ron <ron@merehuman.net>

Updates

commit 45af2b3ed7190b2f04d24226fa01e186a13617d9

Author: Ron <ron@merehuman.net>

Initializing CODE folder

commit d4b458a48d076f02d1a88fc9a4a14fc4b8341691

Author: Ron <ron@merehuman.net>

Presentation Material

commit 522ca614cb7719f7b5e06d1333bec9d466f4f151

Author: xolthan <xolthan@gmail.com>

Add files via upload

commit e5cfc6ca08ea434a7f4c4fbf763debd124020fd5

Author: Nidhimidha <36167384+Nidhimidha@users.noreply.github.com>

Update References-Links

commit c928cfed29d90805c1c87c06f553210ab8f6aecc

Author: Ron Andrews <35875316+ronandrews@users.noreply.github.com>

Create References-Links

commit 32692d6249f56fcbbac7581fdb77326d18a74a84

Author: Nidhimidha <36167384+Nidhimidha@users.noreply.github.com>

Create HomePage

commit 8b7eeb65ff87929abb8347b2fd00727698a85bf0

Author: Ron Andrews <35875316+ronandrews@users.noreply.github.com>

Initial commit

1. For instances where a URL isn’t available, such as for local test files, a default value of ‘*no url*’ is used [↑](#footnote-ref-1)
2. Extracts text between <title>some text</title> [↑](#footnote-ref-2)
3. Python sort function is a hybrid of merge sort, average performance O(n log n) [2] [↑](#footnote-ref-3)
4. Bold indicates folder, italics indicates file [↑](#footnote-ref-4)