Nithin Tumma (ntumma@college.harvard.edu)
Neel Patel (npatel@college.harvard.edu)
James Ruben (jruben@college.harvard.edu)
Eddy Grafstein (egrafstein@college.harvard.edu)

Annotated CS51 Project Final Specification
(Annotations are included in footnotes)

Overview

The main goal of our project is to create a means to succinctly organize information relevant to a certain topic. For example, given a specific focus, we are looking to find a way to create a "book" tailored to the topic based on information found in various articles and documents online, organized in order of relevance and breadth from wide general overviews to detailed explanations of complex subtopics. We look to use a (most likely open source) web crawler to gather appropriate articles and documents, and construct an algorithm to parse out a series of important key words about each overall topic from the entries we have collected. Then, we could use the Jaccard Coefficient to determine the overlap between each article and our predetermined keywords. By factoring in their lengths, we could determine the density of keywords on each page and produce a good estimate of how in-depth an article goes into each topic, and then sort each item based on this "breadth factor" to organize them in order of increasing complexity.

Feature List

1. We plan to implement an open source web crawler, such as PyGoogle (a python wrapper for the Google API), to gather search results regarding each topic.[1]
2. We will construct an algorithm to parse out a handful of keywords from all the search results on the topic, perhaps similar to the one described here:[2]
http://arxiv.org/pdf/1006.1184v1.pdf
3. After putting together the list of keywords, we will implement an algorithm using the Jaccard Coefficient to compare them to each article and determine the overlap, in a similar manner to that outlined here:[3]
http://www.iaeng.org/publication/IMECS2013/IMECS2013_pp380-384.pdf
4. Then we will come up with an algorithm to take this overlap data and figure out the density and distribution of the keywords, which will allow us to rank to pages based on their breadth/specificity
5. A secondary goal is to include a summarization functionality, perhaps using a python wrapper for the Open Text Summarizer (http://libots.sourceforge.net) which would generate a

---

[1] We ended up using functions from google.py instead of PyGoogle (which no longer exists), which performed the exact same function as a python wrapper for the Google API.

[2] The way we went about extracting keywords for each topic was by calculating the TF-IDF (term frequency-inverse document frequency) scores of each inputted word to determine how important that word is to the text, and selecting the 10 words with the highest scores to be our keywords.

[3] Instead of using the Jaccard coefficients method we just calculated the frequency of the word in the entire document, since most of the documents were pretty small we did not see a quantifiable performance gain from using the Jaccard method.

one or two line description of each entry in order to further simplify and streamline user experience.[4]

6. Another secondary goal is for the final list created to be put together into a sleek, user-friendly interface.[5]

Technical Specification

There are 3 distinct portions of the project: 1) parsing the keywords associated with the search query from its Wikipedia (or Wiki-esque) entry and determining the correlations between the words, 2) finding the web pages that include subsets of the keywords and ranking them by breadth (computed from the density distribution of the keywords), and 3) ordering the web pages based on their complexity/overlap.

We will have two teams: 1) James and Nithin, 2) Neel and Eddy, each tasked with building certain aspects of the overall project.[6] Initially, team 1 will focus on the first part of the project, while team two will focus on the second. After both are completed, they will work together on the final part.

To modularize, we will assume the format of the keyword list and correlation matrix, and generate some test data-sets so that Team 2 can begin working on implementing the search functionality and parsing of the web pages. In order to fine-tune the computation of complexity and overlap to sort the links, we will need a much larger dataset, so we will proceed after part 1 is complete. At that time, we envision that Part 2 will be almost complete, so both teams can work on the sorting algorithm. At this point, the difficulty will be in getting the search parameters in the algorithm to work correctly, which is just a lot of testing.

Pre-Processing: to obtain a corpus of texts for calculating tf-idf scores

- Randomly choose n texts from Wikipedia (n should be > 10,000)[7]
- For these combined texts, find the dictionary of each word that occurs
- For each word that occurs, count the number of texts that it occurs in. Each word will then have a number from 1 to n associated with it
- This associated number can now be used to find the idf, or inverse-document-frequency, of a word

Processing: to obtain a set of ordered and parsed links/texts for a given topic

---

[4] We ultimately ended up not implementing any summarization functionality—instead we just presented the links in order of increasing specificity from the broadest overviews to the most detailed articles. In addition, we made sure that any information that could be considered a prerequisite to understanding a later topic would be presented before that topic, maintaining a logical flow of learning for the user.

[5] The user interface we designed is a simple webpage that allows the user to perform a query, receive the list of resultant links, and view each page all in the same place. We also included a slider that allows users to filter for broader or more specific articles based on their needs.

[6] We kept this team-based system throughout, the only difference being that the teams ended up being 1) James and Neel and 2) Eddy and Nithin based on our work schedules.

[7] Instead of sampling random Wikipedia articles to determine the TF-IDF scores, we decided to use the Brown corpus which already separates out the words, and computed the IDF for each of the 53,000 words in the corpus.

- Get a single topic from the user
- For that topic, get the list of keywords from the Wikipedia article
    - Calculate the term-frequency inverse-document-frequency (tf-idf) score for each word in the article
        - Calculate tf based on the number of times the word appears
        - Calculate idf based on how common or rare the word is over a given corpus of documents
    - Rank the dictionary of words based on tf-idf scores; those with the highest tf-idf score will occur frequently in this document but be rare across other documents, meaning that they are important keywords in this document
- From the list of keywords, group keywords by correlation coefficient[8]
    - Calculate a correlation matrix for all of the key words
    - Calculate correlation based on two key words occurring in the same sentence
    - Group keywords that have a very high shared correlation value together, meaning that all keywords in a group tend to occur together in the document

Module Preprocess (only needs to run once, results can be stored on disk):

private function:

parseCorpus: returns list of all words in corpus[9]

idfWord (string word): returns IDF of word form the docs in corpus

public functions:

getIdf (void): returns dictionary of key - term in corpus, value - IDF of term

Module Keywords:

private functions:

wiki search: returns text of wiki article of search query (perhaps intelligently finds closest wiki article), throws error if wiki article is non-existent or too short.

wiki parser: returns most important words in text using standard NLP techniques[10]

public functions:

getKeywords (string query): returns tuple of (list of keywords)

- Calculate the term-frequency inverse-document-frequency (tf-idf) score for each word in the article
    - Calculate tf based on the number of times the word appears

[8] We decided not to group keywords together based on correlation coefficients because we thought that it would decrease the variability of our search results

[9] Using the Brown corupus (which is a huge set of documents), and for each document we calculated the frequency distrobution of each word.

[10] For wiki search and wiki parser we just used generalized versions of the functions—for search we just used a general search function that looked for a query on Wikipedia, and for parser we used a general function that given a URL would download the HTML, clean it of any markup, and return it as a set of words.

        - Calculate idf based on how common or rare the word is over a given corpus of documents
    - Rank the dictionary of words based on tf-idf scores; those with the highest tf-idf score will occur frequently in this document but be rare across other documents, meaning that they are important keywords in this document

getCorrelation[11] (string query, list keywords): returns list of lists of correlated words

- From the list of keywords, group keywords by correlation coefficient
    - Calculate a correlation matrix for all of the key words
    - Calculate correlation based on two key words occurring in the same sentence
- Group keywords that have a very high shared correlation value together, meaning that all keywords in a group tend to occur together in the document

2) Web Search/Parsing

Module Search

public functions:

getSearchResults (list relevantWords, list ignoredWords): returns a set of search results for a set of given keywords, but not including search results that contain words from the set of ignored words[12]

Export: links along with breadth/complexity score

3) Order Results

- rank: return ranked list of documents/links from the parsed search results based on the complexity and overlap of results[13]

- sort: sort the ranked list of documents/links based on overlap, so the list is in logical progressing order[14]

[11] Again, we ended up not determining the correlation between multiple keywords

[12] We used the Google wrapper to get the top 3 URLs of the searches that were formed by the term + two of the keywords for every (10 choose 2) subset of 2 keywords. This was the set of URLs that we processed.

[13] For rank we computed two different ranks: summarability and orderability. Summarability was determined by measuring the difference between the frequency distribution of the keywords in the linked article from the frequency distribution of the keywords in the original Wikipedia article. To determine the orderability score, we first computed the natural ordering of keywords based on what order they appear in the Wikipedia article, and then look at the frequency distribution of keywords in an article sorted by this natural order. Since it's sorted by natural ordering, we were able to find the midpoint of the frequency distribution to tell us if the article tends to focus on prerequisites or later dependent terms.

Module Order

private functions:

getGroupedKeywords (string query): returns a tuple of (list of grouped keywords), by first generating keywords, then generating the correlation coefficient matrix for that query and the given list of keywords. Finally, we would return a list of grouped sets of keywords.[15]

sort (set searchResults): unpacks a set of search results into a sorted linear list

public functions:

getOrderedResults (string query): based on the keywords.getKeywords function, gets the list of keywords based on a Wikipedia article, and groups together these keywords based on the getGroupedKeywords function. It then searches for each of these grouped sets of keywords based on the Search.getSearchResults function, with the argument of (current keywords, all non-current keywords) to prevent overlap. It then unpacks these search results into a sorted list, and returns it.[16]

Tasks to finish -

Week 1:

- Download Wiki corpus, find the 500 most common words, generate IDF dictionary composed of the key value pair (word, IDF score) for each word in the corpus
- Given a string of text, extract keywords and group based on correlation
- Given a query, get Wiki article text - throw error if of insufficient length/to few keywords
- Combine steps 2 and 3 with a wrapper function to return keywords for a query, with graceful degradation if no Wiki article exists
- Use test-driven development to write tests for: keyword extraction and getting Wiki article

Week 2:

- Implement open-source wrapper for Google Search (pyGoogle)
- Implement a function that searches for grouped sets of keywords based on the Search.getSearchResults function

---

[14] We sort the URLs by summarability first, and then within each group of 10 consecutive URLs (or bucket), and within that bucket refine the sorting by using the orderability scores. Therefore the buckets are sorted by summarability and each bucket is internally sorted by orderability.

[15] For the same reason as the correlation functions, we did not end up using grouped keywords so we did not use this function

[16] Since we did not take into account correlated keywords, the functionality of the getOrderedResults function was supplanted by the rank and sort functions as described above.

- Implement a function that can search for words in one list, but also ignore results that contain words from another list (which will allow us to use (search,ignore) lists and prevent overlap
- Implement a function to take a hierarchical set of search results and unpack them into a linear sorted list, based on breadth-first traversal

Week 3:

- After the core of the project is completed, write a Web interface for the core functionality
- Create a method to visualize the hierarchy of progress, from finding initial keywords to searching for text on each keyword to analyzing sub-keywords to ordering text results
- Ensure that all failures are handled on front-end

Version Control - git on Github

We've set up Github accounts (through their student programs) for our group. The main git functions will be pulling and pushing, with little branching as we will be working on modularized and non-overlapping sections of this project.