Homework II: Building an Apache-Solr based Search Engine and Ranking Algorithms for NSF and NASA Polar Datasets

1. Session with Dr. Burgess and Developing scientific questions

The hangout session with Dr. Burgess involved understanding the kind of questions that we can expect the researchers to query on our search engines. She asked us to look at answering questions that involved two or more independent events and fetching the related documents accordingly. Some examples of such questions included:

a. Effects of mineral ore mining by countries on melting ice sheets in various regions close to the arctic.
b. Changes in polar bear and other wildlife behavior due to melting ice sheets.

We also discussed the various fields that were specific to the polar data that we could use to index the entire collection of documents. Some of these fields included geographical parameters like longitude and latitude along with timestamps of the files in the repository.

2. Indexing system using Apache Solr and its ExtractingRequestHandler:

a. Install Solr – Done

b. Upgrading Solr – Built tika with the following support: GDAL, OCR, FFMPEG

c. Dump of the polar data was taken and posted to Solr.

Indexing for Content Based algorithm – The default process of indexing for Solr was used for the content based algorithm. This is because our algorithm uses fields that are already indexed by Solr (as indicated in the Schema.xml document).

Indexing for Link Based Algorithm – We used a Java plugin to index additional metadata from the polar repository. We observed that many of the documents had generalized data with respect to location (latitude/longitude) and timestamps such as date created and date last updated. Tika does not retrieve these fields because they are not present as metadata in the documents. Therefore, they had to be parsed separately from the documents using a program. **The same can be found in the folder that was submitted for this assignment.** The geospatial parameters for latitude and longitude (given the locations) are retrieved by CLAVIN, which was installed separately for this purpose. The fields are accordingly declared and defined in Schema.xml. Temporal data was also retrieved from this documents using the same plugin and used as an index field. The Java program that does the retrieval of these extra metadata fields is part of the submission made for this assignment.

3. Nutch indexing system to build up Solar Index

a. Tika instance was upgraded to include FFMPEG support.

b. Difference between what tika extracted in Nutch and what Solr extracted with respect to metadata:

Currently the fields that are indexed when using Nutch (integrated with Tika) are: id, title, segment, boost, digest, timestamp, type, date, content length, url, *_version_*.

The difference when using SolrCell to index the fields is that unless the above fields are not explicitly mentioned, Solr will not index them. Some of them are mentioned by default in Schema.xml.

One field that Nutch doesn't index, that SolrCell indexes, is content. The dynamic fields that are set in Schema.xml also influence the fields that are indexed by SolrCell. As a result, we get fields like attr_<html tags> that are indexed for HTML documents. Importantly, SolrCell also indexes the content of a document and saves it in the field called "text".

4. Designing algorithms

Content-based algorithm:

We developed our content-based relevancy using TFIDF and cosine similarity. The algorithm for the same is as follows:

1. Obtain the documents with their indexed fields from Solr. In this algorithm, we focus on the "content" field of the documents. We perform a clean up of the
2. For each document create a **Document-TF-HashMap** that stores the term-frequency of all its terms. The keys are the words and the values are the number of times the words appears in the document.
3. For the entire collection of documents create a **Document-IDF-HashMap** that stores the inverse-document frequency. The keys are the words (that appear at least once in at least one document) and the values are the logarithmic function of the number of documents that they appear in.

$$IDF_{term} = 1 + \log_e(N / N_{term})$$

   Where,

   N - No. of documents

   N_{term} – No. of documents that has *term* in it
4. Create a **Document-TFIDF-Hashmap** that stores the query-term and its TFIDF *mapping* for each document. In this HashMap, the document objects are the keys and the TFIDF mappings are the values.

5. Build a **Query-TFIDF-HashMap** that contains the TFIDF values for each distinct term in the query. The keys are the distinct words in the query and the values are its TFIDF values of the words.
6. Iterate through all the documents and compute the cosine similarity for each based on the **Document-TFIDF-HashMap** and **Query-TFIDF-HashMap** using the formula:

   Cosine Similarity = $\frac{\text{Dot product (Query, Document)}}{\text{ll Query ll} * \text{ll Document ll}}$

   where,

   a) Dot product – $x_1 * y_1 + x_2 * y_2 + \ldots + x_n * y_n$ (n is the number of words in the query)

   b) ll X ll - Sum of squares of all terms in X.

7. Sort the Documents based on their cosine similarities in a decreasing manner. The more relevant documents will appear at the beginning and the less relevant documents will be pushed to the end.
8. Display the documents whose cosine similarity is above a pre-established threshold.

   The Java program that implements this algorithm can be found in the folder that was submitted for this assignment.

Link-based algorithm:

The link based algorithm uses the properties of the polar data set in order to rank the documents based on certain “links”. This is similar to Google’s PageRank algorithm in that it uses common metadata features to represent in-links and out-links between documents as opposed to citations.

The extra metadata fields that we have indexed here are geospatial latitude and longitude of the document and its time stamp.

The algorithm first builds a graph where each node is representative of a documents and each edge between two nodes represents a common metadata field between the two. The edges are all considered to be bidirectional, as a result of which if two documents say X & Y had 2 common metadata fields, then there would be two directed edges from X to Y and two directed edges from Y to X.

Once the graph is constructed, we run the PageRank algorithm for each document, which is essentially a “vote” by all other documents to indicate the importance of that document. For every iteration through the list of documents, we establish an importance order based on the PageRank scores. When two consecutive iterations through the documents result in the same ordering of the documents, we break out of the loop.

The documents are then finally sorted based on their PageRank scores.

The algorithm is as follows:

1. Each document object stores the following information about the document that it belongs to: ID, Latitude, Longitude, Time Stamp, Title, Description, Links (total number of out links) & a Hash Map that maps the number of edges originating from this document to every other document.

2. Store all document objects in an array called **Document_List**. Create the graph that represents the collection of documents by comparing each document with every other document. If there are common metadata fields between the two documents, we establish an edge between the two.

3. Create a Hash Map called **PageRank_HashMap** that stores the keys as document objects and values as their PageRank scores. Initially all PageRank scores are 1. Initialize another hash map called **New_PageRank_HashMap** that stores

4. While true,
    a. For each document in *Document_List*
        i. Calculate its ***PageRank*** and store in ***New_PageRank_HashMap*** *using the formula:*

$$PageRank\_Score(x) = (1-d) + d(PR\_S(y1)/Links_{y1} + PR\_S(y2)/Links_{y2} + ... + PR\_S(yn)/Links_{yn})$$

Where, y1, y2….yn are all documents that have an in link from document x.

    b. Endfor
    c. If, order of documents in **New_PageRank_Hashmap** is the same as order of documents in **PageRank_HashMap,**
        i. Break out of while loop
    d. Else,
        i. PageRank_HashMap = New_PageRank_HashMap
c. Endwhile

5. Sort the **PageRank_HashMap**. The documents with the high PageRank scores are the ones that are most relevant.

The Java program that implements this algorithm can be found in the folder that was submitted for this assignment.

5. Developing a suite of queries that demonstrate answers to the scientific questions

You will find below, 5 scientific questions that we have framed in order to demonstrate the relevance of our algorithms. Given below each question is the query that should be entered while running the script mentioned in Question 6.

1. How is mineral ore mining in the arctic region affecting climate change?
    a. Query: Mineral ore mining climate change
2. How are activities undertaken in the name of national security by countries close to the Arctic Circle causing changes to the ice caps?
    a. Query: National security interest melting ice
3. What are the different geographical regions that are covered in the polar repository?
    a. Query: Geographical regions covered
4. How has the changes in melting ice caps caused the change in polar bear behavioral patterns?
    a. Query: Melting ice changes in polar bear behavior
5. How has the rise in sea level due to melting ice caps caused a shift in human habitation?
    a. Query: Sea level rise changes in human habitation

6. Program to run queries against Solr Index and output the results in an easy to read manner

We have a python program that prompts the user to enter the query, then runs that query (after building the relevant query string URL) against Solr, parses the JSON returned by the same and display it to the user in a readable manner.

The file name is **readUrl.py**.

Additional questions that were part of the assignment PDF:

1. How effective the link-based algorithm was compared to the content-based algorithm?
    a. Link-based algorithms are query independent whereas content-based algorithms are dependent on the terms of a user generated query.
    b. Link-based algorithms in our case use links based on certain common attributes between documents.
    c. Therefore, the effectiveness of the link-based depends on the nature of the documents in the repository. If the documents do not share too many common attributes or have similar and uniform links across the graph then the link-based algorithm will not be too effective. On the other hand, if the document links are skewed then there arises a natural ranking order of relevance, where more important documents tend to have more links originating from them.
2. What questions were more appropriate for the link-based algorithm compared to the content based ones?
    a. Questions that were more generic in nature (with respect to the polar database) were more appropriate for the link-based algorithm where as questions that were specific (and by that

related to only a handful of documents) were more appropriate for the content-based algorithms.

3. Describe the indexing process – what was easier – Nutch/tika + SolrIndexing (or) SolrCell?
    a. Nutch/tika indexing is done using the commande: **bin/nutch solrindex http://127.0.0.1:8983/solr/ crawl/crawldb -linkdb crawl/linkdb crawl/segments/.** Therefore, we need the segments that have been crawled using Nutch.
    b. SolrCell indexes documents either using curl commands or bin/post commands. The posting is done using HTTP post requests directly to the Solr core (collection 1 in our case). Once indexed, the core can be queried to using either the Solr graphical interface or by using the relevant query strings in the browser.

7. (Extra Credit) Developing a Lucene-latent Dirichlet allocation (LDA) technique for topic modeling.

For the Lucene- Latent Dirichlet Allocation (LDA) technique, we used the following idea :

1. We defined a set of topics, considering each topic to be a probabilistic distribution (Poisson distribution) over the vocabulary.
2. Then we assigned each word in the document to a related topic.
3. The input query is tokenized and the topics the tokens (or their combinations) correspond to are noted.
4. What percentage of each document corresponds to each topic, noted in the previous step, is calculated.
   To determine the percentage, for each topic, we measure the frequency of each word belonging to the topic and occurring in the document. Now, we divide the calculated frequency by the total number of words in the document and multiply it by 100.
5. The documents are sorted based on decreasing order of the percentages calculated in Step 4. The resulting documents are returned.

Difference observed: The documents returned after running the content-based and link-based algorithms cover a wide domain, whereas, those returned from the Lucene LDA technique belong to a much narrower domain.

Explanation: A possible explanation to the above observation could be that the algorithms listed in task #4 rank documents based on ideas such as the number of times the words in the queries appear in the document, location of the document, etc. However, the Lucene LDA technique ranks documents based solely on topics the words in the queries belong to (i.e. the topics of interest). Thus, in case of the LDA technique, all the documents that belong to the topics of interest are returned first instead of the documents that contain the query words most number of times. This explains why the documents that are semantically more relevant to the queries are given

precedence over those that merely contain the search keywords a lot of times.

8. (Extra Credit) Integrating the relevancy algorithm into Nutch.

1. In Eclipse, under the **File** menu, click on **Import**.
2. In the pop-up, select **Projects from Git** under the **Git** category. Click on **Next**.
3. In the next pop-up, click on **Clone URI** and then click on **Next**.
4. Now, in the text box for 'URI', enter the following - https://github.com/apache/nutch.git , which is the **Git repository for Nutch**. Click on **Next.**
5. In the **Branch Selection** pop-up, let all the files be selected (default). Click on **Next**.
6. In the **Local Destination** pop-up, enter the directory (in our case, /Users/JK/git/nutch) where you want the Nutch repository to be stored and click on **Next**.
7. Wait for the projects to download and select the **Import as general project** option. Click on **Next**.
8. Type the project name and click on **Finish**.
9. Now, in the **Package Explorer**, you should see a project named **nutch.**
10. Go to **nutch -> src -> java ->org ->apache ->nutch->scoring.** In this folder, you should see a file named **ScoringFilters.java** where you can add your link-based algorithm's code.

9. (Extra Credit) D3 visualization of the link-based relevancy

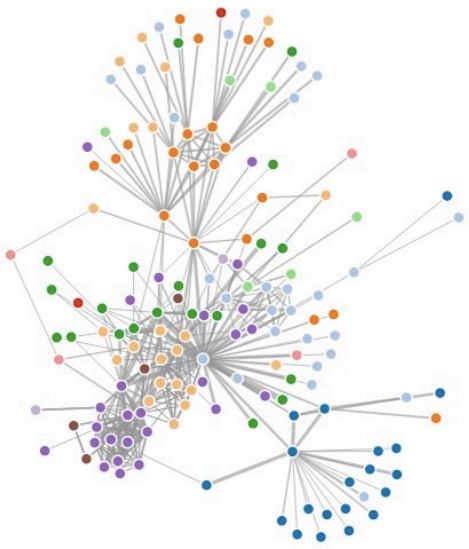

Fig. 9.1 - Force-Directed graph using D3

We created a D3 visualization of the link-based relevancy using D3.js. We generated a Force-Directed graph to represent the link-based relevancy on the documents we selected for testing.

We first took around 500 documents and ran the D3 code we had written to generate the graph. However, the resultant graph was too cluttered, making it difficult for us to understand it. Hence, we used 153 different files to generate the above graph.

We created our own json file which had two keys - **nodes** and **links**. The value to each key is an array of objects. The nodes had the following fields - **name** and **group** as the object properties. The name field represents the name/id of the documents that we used for testing and the group property represents the rank of the respective documents. Each rank is identified by a different color in the resultant graph. All documents with the same rank are represented with the same color.

Fig. 9.1 is a screenshot of the SVG we obtained of a Force-Directed Graph. The related documents can be seen in closer proximity, while unrelated documents are farther apart according to the ranking by our algorithm.

The other key (links) has the following object properties - **source** , **target** and **value**. The source and the target define the source document which is linked by an edge to a target document. The value field shows the amount of similarity between two documents. The more the value the thicker is the edge between two nodes. Thicker the edges, more is the similarity.

If the value field is 0, then the two documents are not similar and would not be connected by an edge.