Art Search Engine Platform

Jiaxuan Lu  
Department of Computer Science  
New York University  
jl5006@nyu.edu

Nan Liu  
Department of Computer Science  
New York University  
nl1554@nyu.edu

Oukan Fan  
Department of Computer Science  
New York University  
of 394@nyu.edu

Abstract—The paper introduced an art search engine platform that differs from ordinary search engines, which provide users only directly related information about the users’ queries. The art search engine platform introduced in this paper provides users both the directly related introductory information of their queries and the necessary educational background information of the artworks and the artists of users’ interests to provide them with a more comprehensive and extensively broader understanding and knowledges of the artists and artworks that they search for.

Keywords — search engine, distributed indexer, MapReduce, tornado web framework, relevance analysis, information retrieval.

I. INTRODUCTION

Our project is composed of a web search engine, a MapReduce framework that we implement, a Distributed Indexer, a user interface, and some MapReduce programs that we used to extract relevant information from our data sets. We begin with the enwiki-latest-pages-articles-multistream.xml dump, which contains all the latest articles of Wikipedia, downloaded from Wikimedia Meta-Wiki. We then used the data sets from Museum of Modern Art in New York to limit the scope of our database by extracting all the related articles from the Wikimedia xml dump. After we parse the xml dump and preprocess our raw data, we perform relevance analysis on our data using our MapReduce Framework. We then have two approaches for building the indexer of the extracted Wikipedia articles. First, we build a distributed indexer using MapReduce to compute inverted index, inverse document frequency, and document store parallelly and store as Python dictionary. The second approach is that we use python sklearn library to compute TF-IDF beforehand and store as sparse matrix. Along with some other adjustments of our infrastructure, which will be addressed later, we ameliorate the performance of our search engine, which reduces the searching time from 13 seconds to about half a second. Python tornado web framework is deployed on servers that used for information retrieval and a web application is implemented for providing a user interface using CIMS Linux Servers provided by NYU.

II. MOTIVATION

Artworks and artists are always hard to be understood by non-professional people due to lack of comprehensive understanding of the background of the artworks or artists. Nevertheless, the mainstream search engines all provide only the most direct information based on users’ queries. So our project aims to develop a search engine platform that receives input information from users relates to any aspects of art, artists, or artworks, and gives feedback knowledges that are not only the information most directly related to users’ queries, but also the other relevant art information about the queries, which helps users to expand their knowledges related to their interests while deepening and widening their understanding about the artists or artworks they are curious about.

III. ARCHITECTURE

The very first step towards building up our search engine architecture is data preprocessing. The raw data is firstly parsed and preprocessed by our python reformatter program to partition our large input data set into small, HDFS-style blocks and format into structured data, which is a pickled structured data that stores the title, document id, and document text for each Wikipedia article as a dictionary for further processes. The all-article Wikipedia data set here serves as a main information provider, while artists’ and artworks’ names provided by MoMA open source data sets serve as a filter that we extract every page that matches a MoMA artist’s or a MoMA painting’s name to form our desired data store.

A preview of the search engine architecture of our first approach is presenting in the following diagram:
The preprocessed and partitioned data sets are passed to our MapReduce framework, where inverted index, inverse document frequency and document store MapReduce programs work parallelly to build our distributed indexer that is used to score each document. Inverted index MapReduce program serves the function of mapping every term that appears in our data sets to a list of tuples that stores every document, its id, that has the term in it with the term frequency in that document. Document store MapReduce program outputs pickled dictionaries that map each document id to a tuple that contains the document’s title and text body. IDF MapReduce program serves the function of outputting pickled dictionaries that store each term as key and term’s idf score as value. The IDF score for each term is computed as indicated below. The distributed indexer, including counting term frequencies, calculating IDF scores of each term, is based upon preprocessed document texts. The preprocessing of texts includes removing punctuations and special characters, removing stop words, stemmer, bigram collocations, etc.

\[
TF-IDF \text{ Term Weighting} \\
\quad w_{ij} = t_{fi,j} \cdot \log \frac{N}{n_i}
\]

\( w_{ij} \) weight assigned to term \( i \) in document \( j \) 
\( t_{fi,j} \) number of occurrence of term \( i \) in document \( j \) 
\( N \) number of documents in entire collection 
\( n_i \) number of documents with term \( i \)

Apart from building distributed indexer, the preprocessed data sets are passed to relevance analysis MapReduce program to count term frequencies in each document and outputs only the top five frequently used terms in each document. The results are stored as pickle-structured dictionaries with document id as the key for each dictionary, and the value is a list of tuples that zips each term with its frequency in the corresponding article.

Upon constructing distributed indexer and extracting relevant words. We built a search engine that uses the distributed indexer to search for every relevant word we get beforehand. The pickled preprocessed data processed by relevance analysis MapReduce program is passed to the relevance processing engine frontend server and every word serves as a query. For one document, five terms subject to it are searched using TF-IDF scoring strategy, as specified in the equation above, by accessing the distributed indexer. For each term, we only store 2 unique results, the document ids, that returned by our search engine. In total, we have ten related documents, stored by document id, linking to each document. This relationship is stored as another pickle structured file in the form of mapping one document id to a list of 10 distinct relevant document ids. The number of relevant documents can vary based on demand.

The user can input queries, either a term or a sentence, to the user interface of our art search engine platform. Based on the TF-IDF scoring strategy again, we retrieve the best match document based on users’ query using the distributed indexer. With this best matched document id at hand, we retrieve the 10 distinct relevant document ids we link to this best matched document previously.

Last step, we output the best matched document along with the 10 relevant documents to users. To retrieve the doc-
documents' title, url and content, the document servers search every document id in document store and the web application presents the results, including one directly related text and ten indirectly related texts, to users' queries, which would help users to better understand the artists or artworks by providing necessary related information.

When it comes to information retrieval, due to the unsatisfactory performance of our first approach, which takes about 13 seconds to search for one word, we tried the second approach with the hope of a better solution for a quicker searching experience for the users. Our second approach has the following architecture:

![Input Data](Artists from MoMA / Artworks from MoMA / Wikipedia dataset)

Reformmater

Preprocessed Data

Structured file: Dictionary

*title*, ...  
*doc_id*, ...  
*doc_text*, ...

Extraction of Document Text

List of Document text

['doc_text1', 'doc_text2', 'doc_text3', ...]

Generate

Inverted Optimization

Inverted Index

IDF

TF-IDF

Relevance Analysis

Obtained top 5 frequent terms from inverted index

Document Link

one document id linked to ten other related document

All necessary intermediate file is read into memory at this point

User

Input query

Figure 3.2 Optimized Architecture

The major change is that instead of building a distributed indexer, we calculate TF-IDF scores beforehand and store as sparse matrix, which would decrease indexing time and scoring time. Besides this major change, we also make some other adjustments to optimize the performance of our search engine.

The second approach uses the same way to filter and reformat our raw data. However, instead of partitioning our large input data set into small, HDFS-style blocks, we keep only one large preprocessed reformatted structured data as document store. From our comprehensive document store, we extract each document’s text as a string and construct a list of strings, which contains all the document texts in our data set.

From this list of text strings, using Python sklearn library, we can construct the inverted index, and TF-IDF weighting scores as sparse matrix, IDF as Python numpy array and a list of terms in our entire data set. The row of the inverted index matrix indexes each document and the columns represent all the words sorted by alphabetical order in our data set. The value of the inverted index matrix entries, referenced by a document and a term, stores the term’s number of occurrences in that document. The row of the TF-IDF matrix is also used to index each document, and the columns still represent all the words in the documents. The value of the TF-IDF entries, referenced by a document and a term, represent the TF-IDF scores of the term in that document. The storage of IDF scores for each term is a 1-D numpy array with one value for one term indexing by the word’s same index in the comprehensive words list.

The top 5 frequent words in each document are obtained by finding out the highest 5 numbers’ indices in each row of the inverted index matrix. Using these indices to refer back to the all-term list and then associate with each document id, we obtain the outputs of our relevance analysis.

Due to the size of our data set, which contains about 20,000 articles and 200,000 distinct terms, the size of the TF-IDF and inverted index matrices would be about 20,000 x 200,000, which is tremendous and would make writing to file impossible. Being aware of the fact that with the columns being the entire words list, there would be massive zeros in our matrices, because based on common sense, a document text would only have hundreds or thousands of words, which are far less than 200,000. Therefore, storing the matrices as sparse matrix would be a solution. We store our matrices as sparse matrix, using python scipy.sparse.csr_matrix, which only stores the indices of nonzero entries. When writing to files, storing data as sparse matrices reduces the file size from more than 10 GB to just under 100 MB.

With TF-IDF scores pre-calculated as one matrix, only one index server would be sufficient for our search engine. It reduces the score calculation time. Also, instead of building query vector and document vectors as in the first approach, during the calculation of final document scores, when more than one word is searched, we perform matrix multiplication using python numpy, which decreases the calculation time significantly for large scale data. Another modification of our search engine is that instead of reading files from disk, we read files into memory when our search engine starts and sends the data to index server and document server as arguments. Also, we add the feature to cache users’ search that
further reduces the searching time for second search, third search, etc.

Mechanism of constructing relevant document lists, document retrieval and user interface remain unchanged from the first approach. However, searching relevant terms to get relevant documents use the optimized indexing, which results in a huge amount of time saved. To emphasize, the second approach reduces the searching time for one query from 13 seconds to less than 1 second.

IV. LIST OF EXTERNAL SOFTWARE

Python
CIMS Linux server
Tornado web framework
Web application developed by Professor Matt Doherty

V. LIST OF WEB RESOURCE

https://www.wikipedia.org
https://meta.wikimedia.org/wiki/Main_Page
https://github.com/MuseumofModernArt/collection
http://www.robots.ox.ac.uk/~vgg/data/paintings
https://wikis.nyu.edu/display/NYUHPC/Clusters+-+Dumbo
http://www.nltk.org/
http://www.numpy.org/
http://scikit-learn.org/stable/

VI. PERTINENT SYSTEM FEATURES

Python libraries used: tornado, xml.ElementTree, sklearn, sklearn.feature_extraction.text, scipy.sparse.csr_matrix, nltk, nltk.corpus, nltk.collocations, nltk.tokenize, nltk.metrics, nltk.stem.snowball, numpy, mwparserfromhell, re, pickle, math, os, json, argparse, itertools, operator, sys, collections, uuid, socket, logging, urllib, hashlib, getpass, subprocess, csv.

VII. RUNNING EXAMPLE

1. Successful example

The following figure is a screenshot of one successful search of our art search platform:

![Figure 7.1 Successful Example](image)

The searching time as shown in figure 7.1 is 0.022 seconds, which demonstrates the quick response time of our search engine.

The query in figure 7.1 is Guernica, which is one of the most well known paintings of Pablo Picasso, that depicts the crudeness and misery of Spanish Civil War. Our search engine aims to provide the user with relevant information and aspects of art in an educational purposes. In the figure, the first result that is returned to the user, Pablo Picasso, the painter of Guernica, with no doubt is a relevant page that the user would be interested in. The second article, Still Life with Old Shoe, by reading the Wikipedia page, is a 1937 oil painting by Joan Miró, and now is part of the permanent collection of the Museum of Modern Art in New York City. The painting was painted in the context of Spanish Civil War and Joan Miró depicts his anguish and fear in it. Obviously, it is very relevant to what the user is searching for, Guernica. And this result would introduce the user with the information of another painting that shares the same theme with Guernica. The third result, which is Spain, again would provide the user with some relevant information that he or she would interested in.

Therefore, this example demonstrates the success of our search engine platform in terms of quick response time and relevant analysis. Moreover, it also demonstrates that we achieve our goals that aligns with our motivations in first place.
2. Failed example

Our search engine is based on mostly MoMA and Wikipedia painting and sculpture datasets, so it runs successfully when the query is related to these two areas but normally fails to find anything if user attempts to search other things like musician, computer science, etc. beyond the areas art. However, this issue has been included into account when we are developing our search engine. We set up user-interface in our search engine that allow users to provide their contribution to our dataset if they failed to find anything satisfying like Wikipedia do. In the future work, we can expand our dataset by processing both the user-contributed information to and downloading more public dataset from other museums and institutions.

The following is a screenshot of the search engine of our first approach:

Figure 7.2 Initial Searching Time

Figure 7.2 demonstrates that with our first indexing mechanism, the responding time is 12.389 seconds, which obviously is not a pleasant user experience and is not a practical search engine in real world.

Following figures are presentation of some other failure examples of our platform:

Figure 7.3 Failure Example 2

The screenshot is taken before we handle the case of the query words not in our data sets at all. We have handle this case in our search engine’s final version. The point that we would like to address in this example is that, with our first approach, even there is no results for the query words, the notification of this no-result feedback would be given to users in 12.389 seconds. It, by no means, would serve as an acceptable feature of an ideal search engine.

Figure 7.4 Failure Example 3

Figure 7.4 demonstrates a previously unhandled case that our search engine does not take into account the case that a user under two or more words that are the same. And it is fixed in our final version platform.

VIII. ISSUES

We discuss problems we encountered during implementing our project, solving approaches we used, and outcome we obtained in this section.

1) Data Set Range. The data set we used is collections in MoMA, which are mainly painting or sculpture art, so only limited information about other art forms are contained and artworks and artists in other museums and galleries are improper queries and hard to presented their relevance. Currently our search engine is designed to record the user searched queries without any result for future expansion of our dataset.

2) Data Set Precision. Since the pages are filtered from the wikipedia dataset by fuzzy match of titles of wiki-pages and names of artists or artworks in MoMA collection, there are some art-irrelevant documents in our data set and some valuable artworks are not obtained, such as whose names are "untit-
led". We have not yet found out an effective solution to overcome this problem.

3) **Analysis Model Precision.** Our relevant analysis model treats each word as an independent unit. However, in reality words tightly related to each other or consist in one phrase should be treated as the same query. For example, for query "garden", it should be equivalent to "monet’s garden" in art search engine, and the results are the series paintings of Monet instead of the gardens nearest to the users returned by regular search engine. We have extracted some pairs of words that have strong associations, such as "new" and "york" which both represent "new york", but failed to use these phrase in analyzing the article texts due to time limited.

4) **Analysis Algorithm Performance.** The relevance extraction procedure took us days to run at the beginning of the relevance analysis section. Firstly we attempted to implement a MapReduce interface for index servers of relevance processing engine to run the procedure in parallel so as to speed up the process of extracting relevance documents, but it failed. Then we referred to the concept of distributed systems, running the procedure on different machines at the same time and introducing cache mechanism to avoid wasting time on repeating works. The time was greatly reduced but still not satisfying. So we changed the architecture of TF-IDF calculation section and optimized the process of retrieving information in index servers. This approach works and decreases the running time from 9 days to 40 minutes.

5) **Duplicate Results.** In the experimental result of relevance extraction, it is common that some of the most relevant pages of an article are the article itself for the reason that this document is exactly the one who scores highest under TF-IDF model for one of the most frequent terms in it. We added a function to remove duplicates to avoid same documents occur multiple times in one retrieve so that more information could be offered.

6) **Unrestricted Criteria.** During relevance extraction process, we found that some terms that have top frequencies in a document do not actually linked to articles that are most relevant to the query. We tried to fetch some obviously unrelated and nonsense words from our data set and import them into the "non-refer repository". As a result, the problem is partially solved. We are also seeking a more intelligent method to find and update the meaningless terms during processing data.

**IX. EVALUATION**

For the first approach of data retrieval, we used multiple index handler website for processing the reformatted data. Then the TF-IDF scores are calculated in every index handler for every document. Based on the top two result of every term in every document of every index handler, we need to combine them to rank documents in order to and retrieve documents. And the data is read from disk by index servers and document servers of our search engine platform, which also makes the process of information retrieval much slower.

For the second approach, only one index server exists for reading in all the preprocessed data. Also, instead of implementing vector multiplication, we implement matrix multiplication, which performs better when the query consist of two or more words. By reading in texts, we generate the matrices whose columns are document ids and rows are terms, the values are the term frequencies in inverted index and TF-IDF scores in TF-IDF. Besides, When starting the frontend of the search engine, the data is read from disk in advance to memory and passed to index servers and document servers as arguments for data retrieval, which accelerate the process of indexing retrieval. This is because when users use our search engine to look for information of their interests, the application has already start working, and the data has already been read to memory. The above factors, along with the additional cache features to store users’ previous searching results, work together that make the search engine performs much more faster than the first approach.

The first approach is abandoned for its unbearable long searching time, while the second approach is finally applied to our application for its quick results-return performance. We kept both implementations in our source code.

**X. FUTURE WORK**

Under our current TF-IDF model, articles with rich terms have higher exposure. We are planning to optimize the model by designing different layers of terms with corresponding weights and classifying terms in an article into different grades based on their ratios of frequency in this article. Thus, short but more relevant articles are more likely to be retrieved.

The result of our search engine is limited to the size of data, both in range and precision. For now only collections in MoMA are included in our data set. Furthermore, we rely on structured data offered by art institutions and only large-scale, powerful art groups have the ability to collect and maintain data. We are going to introduce crawler and data cleaning techniques, combined with the queries recorded, to collect and append resources into our database gradually to provide more accurate result and more information related to other classifications, especially artworks and artists who are underrepresented and have few chances to be exposed to public and collected in exhibitions.

We currently design the architecture with only one index server. We would modify the process of TF-IDF pre-calculation to enable the configuration of the numbers of index servers and capability of running various servers in parallel in corresponding to the grow of data set.

Due to project time limitation, the relevancy between articles is analyzed based on the evaluation of single term instead of phrase. To update our project with more precise results, we would like to use strong associated words integrated with the Word2Vec model to assess analogy terms and phrases that can be treated as same.

Additionally, the Natural Language Processing technology can be used in both data filtering process to improve the con-
centration of art related data and relevance analysis section to increase the accuracy of result.

We would like to add some intriguing additions to arouse users’ search frequency, including, without limitation, presenting related photographs or paintings together with the wiki-snippets, paging the results to show more information, authorizing users rights of appending or modifying articles’ contents.

We will make our project open sourced for further implementation, and also considering using it in other areas that public are not familiar with but interested in, like Peking Opera, martial arts.

ACKNOWLEDGMENT

We would like to acknowledge our professor Matt Doherty and two Teaching Assistants of course "Search Engine Architecture" for giving us advices and feedbacks on selecting project topic, fixing bugs during our platform development. We would also like to acknowledge the technicians of NYU HPC and CIMS of Courant Institute of Applied Mathematics for the guidance and service during our usage of the server. We are grateful to Wikipedia and MoMA for providing data set.

REFERENCES