EECS 767 Information Retrieval

Simple Web Search

Final Project Report

Bijal Parikh

Ragaprabha Chinnaswamy

Sirisha Thippabhotla

Contents

Introduction

Simple Web Search is a web application used for information retrieval from text data. Its features include a ‘**Search’** functionality which can be used to search for relevant documents based on the users search terms. It also contains a **‘Provide Feedback’** feature which allows the user to mark search results relevant to their needs and provides a refined search result. The search results are computed using an inverted index and ranked based on a tf-idf score computed using cosine similarity. The application updates its document base by crawling the web for documents. The crawler is a fast performing multithreaded crawler and crawls documents in the domain pointed to by the user.

Programming Platform

Simple Web Search is built entirely using Python 2.7 in the backend. The front end is rendered using HTML and exchange of data between the front end and the search engine is done using Django framework. Below table lists the important architecture details:

Language	Python 2.7
Framework	Django
UI	HTML + Django Template language + Bootstrap
Inverted Index Files	Stored as txt files
URL parsing during crawl	BeautifulSoup
Tools	Pycharm
Stemmer	PorterStemmer implementation from snowball stemmer library in Python

Architecture

Simple Web Search is composed of several independent components which work in phases. They are: a. Index generation b. Search Engine c. Document crawling

Index Generation

- The search application makes use of an inverted index for choosing the set of documents to return to the use.
- This inverted index is generated using a '1 time operation' and needs to be updated as and when new documents are added to the system.
- Document parsing, removal of stop words, stemming the document is done before index generation, the detailed description of which is provided in the later sections
- Inverted Index are stored in 2 flat files in our system.
    - InvertedIndexAtoM.txt
    - InvertedIndexNtoZ.txt

Search Engine

- Once the index is created, we run the search server which facilitates user search
- The inverted index stored in 2 files is read into cache and used for serving user query.
- When the user enters a search query, the query is run through a parser, stop words are removed and words are stemmed
- The inverted index is then used from the cache to fetch the list of candidate documents and ranking is provided based on the tf-idf score of the documents.
- An important feature of our search engine is that it '**caches the processed search query and its result'**
- The above means that if a user were to search for 'Acadia National Park' and then 'Park Acadia National' our search engine will consider both the queries to be same and return the result documents from the cache rather than compute the results again!
- Our search engine also provides a feature called 'Provide Feedback'. There is a checkbox next to each search result and the users can check the checkboxes for the relevant documents and click Provide Feedback. The search engine will then refine the results using Rochhio algorithm and display the results.

Document Crawling

- Simple Web Search uses a multithreaded crawler to crawl for documents based on the given domain.
- The crawled documents are manually added to the repository and then index generation is run again on the documents for updating the inverted index.

Algorithm, Flowchart and Implementation

Index Generation

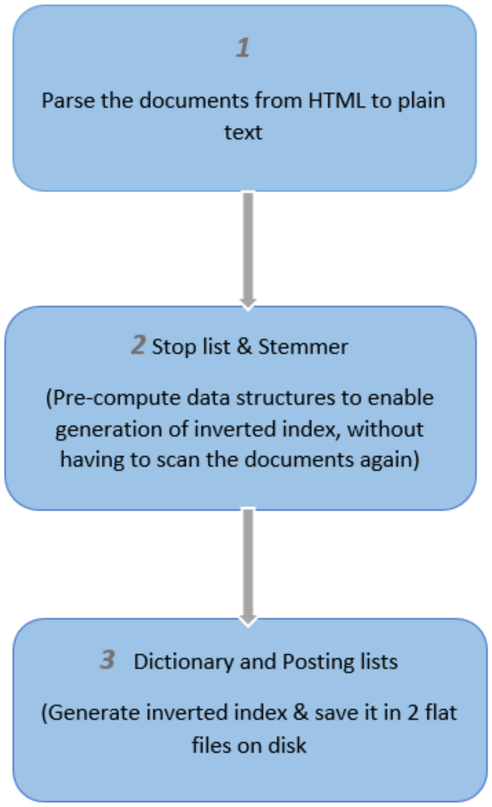

1
Parse the documents from HTML to plain text
2 Stop list & Stemmer
(Pre-compute data structures to enable generation of inverted index, without having to scan the documents again)
3 Dictionary and Posting lists
(Generate inverted index & save it in 2 flat files on disk

Figure 1

- All the original files are stored in a folder on the server at location **websearch/originalFiles/**
- The very first step comprises of parsing the HTML documents to extract text information and this is achieved by using regular expression for pattern matching and removing those patterns. The output files are stored at **websearch/parsedFiles/**
- The files are now read serially, reading one line at a time for a file and:
    - Check if the word is a stop word
    - If not, then stem the word
        a. Update the vocab_df_map, vocab_tf_map, doc_tf_list to update tf, df information for the word
            - **vocab_df_map - <String, Set>**
                - Contains mapping of terms and corresponding set containing doc-ids
                - The size of the set is used to obtain the document frequency(df)
                - The Individual document ids have been used for generating posting lists later on.
            - **vocab_tf_map- <String, Integer>**
                - A map that stores overall frequency of the term in the system (different from term frequency).
            - **doc_tf_list – [List(Map<String, Integer>)]**

   - A list of map containing all term frequencies per document. (DocId is index position – 1) Map at each position contains term present in document and its term frequency

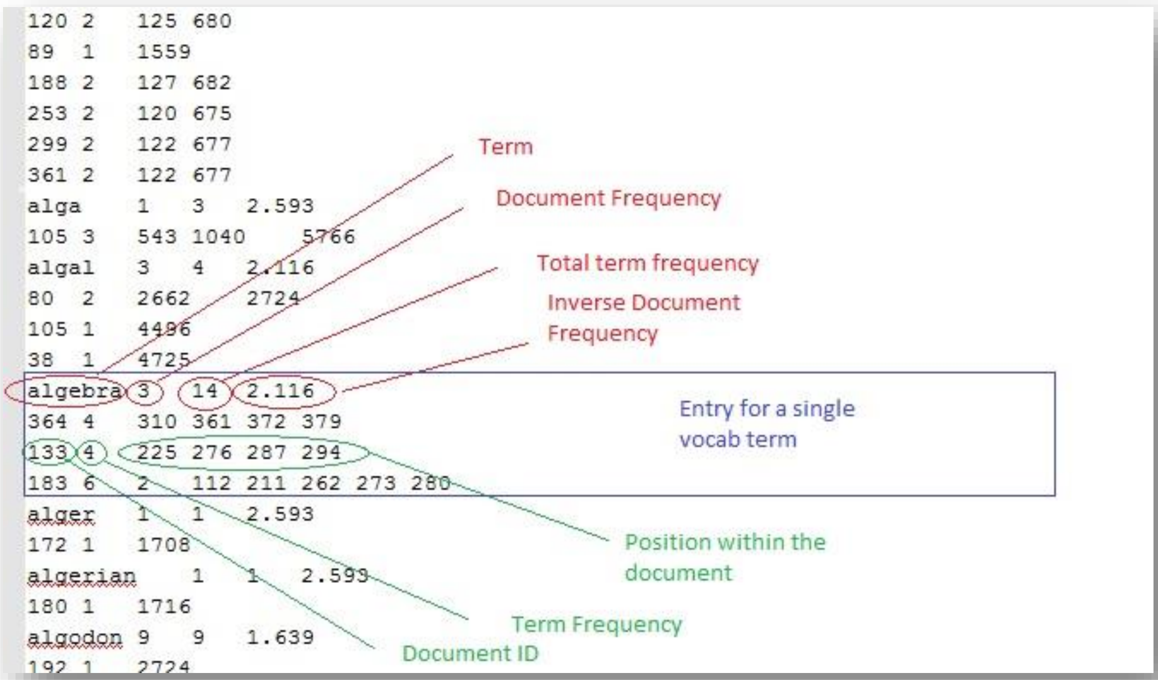
- The inverted index is stored in 2 user defined data structures

  InvertedIndexNode : <Term, docFreq, overallTermFreq, postingListHead>

  Postinglisthead is the head pointer for the posting list (linked list).

  PostingListNode : <docid, docTermfreq, next>

  next is the pointer to next node of the posting list.
- The terms in vocab list are sorted.
- For every term in the vocab list, a new InvertedIndexNode is created which contains a head node of linkedlist of PostingListNode
- The inverted index information is written down in 2 flat files InvertedIndexAtoM.txt and InvertedIndexNtoZ.txt, so that it can read in the future by the search engine.



```
120 2   125 680
89  1   1559
188 2   127 682
253 2   120 675
299 2   122 677
361 2   122 677
alga    1   3   2.593
105 3   543 1040    5766
algal   3   4   2.116
80  2   2662    2724
105 1   4496
38  1   4725
algebra 3   14  2.116
364 4   310 361 372 379
133 4   225 276 287 294
183 6   2   112 211 262 273 280
alger   1   1   2.593
172 1   1708
algerian    1   1   2.593
180 1   1716
algodon 9   9   1.639
192 1   2724
```

Term
Document Frequency
Total term frequency
Inverse Document Frequency
Entry for a single vocab term
Position within the document
Term Frequency
Document ID

Figure 2

Search Engine - Caching

- When we start the search engine, it reads and stores a bunch of things in its cache
  - InvertedIndex information
  - Tf-idf matrix normalized data
  - Set of stop words

Search Engine – Query Processing

- When the user first enters a query, the query is parsed. All the stop words are removed, remaining words are stemmed, and query terms are sorted and stored in a list.
- The query is then used for fetching candidate document list and computing the ranked result
- An important feature of our search engine is that is **caches the parsed sorted query terms and corresponding search results**.
- Eg: is user searches for 'Acadia National Park' the search engine will compute the result and store it in query_cache
- If the user again searches for 'National Park Acadia' the search engine will parse and sort query terms and fetch the stored results from the cache rather than compute it again!

Search Engine – Vector Space Model

- Constructing the V*N dimensional vector space
  - The matrix of size V*N is created, where V is the total number of terms in the query and N is the total number of documents
  - The matrix is populated using the Inverted Index generated earlier:
    - a. For every term in the query, a new vector (temp_tfidf_list) of size N has been created.
    - b. Initially all values of the vector is set to zero
    - c. The idf values of the terms are fetched from the InvertedIndexNode and updated in idf_list
    - d. The inverted index is binary searched for the query term, followed by the linked list traversal of PostingListnode.
    - e. For each PostingListNode, the tf-idf values has been calculated using the idf_list entry and term frequency of the PostingListNode and the corresponding document ID is added to the candidate document set
    - b. When the end of PostingListNode is reached, temp_tfidf_list will be appended to the matrix
    - c. Time complexity: O (V*C) [V=Total number of terms in the query and C= Total number of candidate documents for each term]

Fetching magnitude and normalization:

1. The magnitude of the documents is fetched from the flat files
2. For every document ID in the candidate document list, the tf-idf values are normalized for query terms using the magnitude fetched earlier.

- Similarity Measure
    a. The cosine similarity will be computed between query and each candidate documents and similarity_map will be updated.
    b. The documents will be ranked (descending order) based on the similarity measure and ranked documents are returned
    c. Time complexity: O (C*V) [V=Total number of unique terms in the collection and C= Total number of candidate documents for each term]

Search Engine – Display of Results

- The engine results the list of docids in the descending order of the similarity score
- The django views refer to file mapping.txt which has a mapping of document Ids and the document names. It used this file to fetch the names which are then displayed a hyperlink on the UI
- The UI displays the results, checkbox for providing feedback and the similarity score of the document

Search Engine – Relevance Feedback

- There is an option to provide feedback on search results which will enable the search engine to refine the results and send it back
- Users can select the checkbox for all relevant documents and click on Provide Feedback
- Search engine will use rocchio algorithm with alpha = 1, beta = 0.5 and gamma =0.2 to refine the results
- The implementation is as follows:
    1. We store the search results of the query entered previously (a feature of our query cache!)
    2. We even have access to previous query vector, so that is no re computed
    3. Out of all the document present in the result buffer, the checked documents are considered relevant, remaining non-relevant
    4. We compute the document vectors for relevant documents and sum (difference) them up based on relevant and non-relevant.
    5. We compute the updated similarity and return the results

Document Crawler

- The document and the number of threads are defined
- The main spider is called only once to do initial setup and to get initial set of links to begin with
    1. Initial Setup includes:
        a. Instantiating the class variables inside constructor
        b. Creating a local folder and two text files inside the local folder
    2. It crawls the homepage to get initial set of URL's
        a. It collects the links using beautifulSoup package only if the document content is HTML, else it will return an empty set
        b. It calls the HTML parser to parse the HTML content and send the parsed content for indexing
        c. Add collected links to URLFrontier only if it is not in URLFrontier or Crawled URL's sets
        d. Remove the homepage from the URL frontier and add it to the crawled URL
        e. Save copy of the 2 sets (URL Frontier and crawled URL) in two flat files located in the local folder
- The worker threads are created (number of worker threads are user defined) and the target of the threads are set to the function work where the threads can get the next job (URL) in the queue.
- The worker threads crawl the specified URL to get the set of links (The crawling process is similar to the crawling process of the main spider)
- The worker threads will keep on crawling for specified number of times or until the queue is empty.

Term Proximity

- The inverted index has been updated to record the positions of the terms in each document and stored as flat file.
- Once the server is started, the content of the flat files are read into inverted index data structure.
- The design of term proximity algorithm is as follows:
    - The algorithm fetches the idf values of each term in the query and stores it in idf_map <string, float>
    - The map is sorted based on idf value to find the two terms with highest idf values (Higher the idf more important the term is)
    - The candidate documents along with positions of the terms in those documents are fetched from inverted index data structures and stored in term1_candidate_document-<DocId, PositionList> and term2_candidate_document-<DocId, PositionList>
    - The final candidate documents are selected from the above two sets only if the selected terms are present in a document. This is done by taking the intersection of the two document sets.

- The naïve approach is used to calculate the distance between two selected terms. The maximum distance is set to 25.
    - Time complexity**: O (d*n*m)** [d=Total number of candidate documents which has both the query terms and n= Total number of positions for term1 and m= Total number of positions for term2]
- The candidate documents with its distance is stored in diff_map-<docId,distance>
- The distance of each candidate document is then converted into weight scores and stored in weightage_map-<docId,weight>
    - The weight is calculated using the below formula: **Weight = (26- distance) / 25**
- The final score is calculated using cosine similarity as well as the term proximity weight score and stored in tot_sim_map -<docId, similarity>
    - The final score is calculated using the below formula: **Final Score = ((0.7 * (cosine similarity score)) + (o.3 * (term proximity weight score)))**
- The documents are ranked based on the new scores and the results are returned

Results/Evaluation

- The basic version of our search engine gives search result in 0.2 to 0.3 seconds

- If we search for the same query again, ignoring the order of the term, then our search engine uses query caching to return results in approximately 0.002 seconds, using query caching

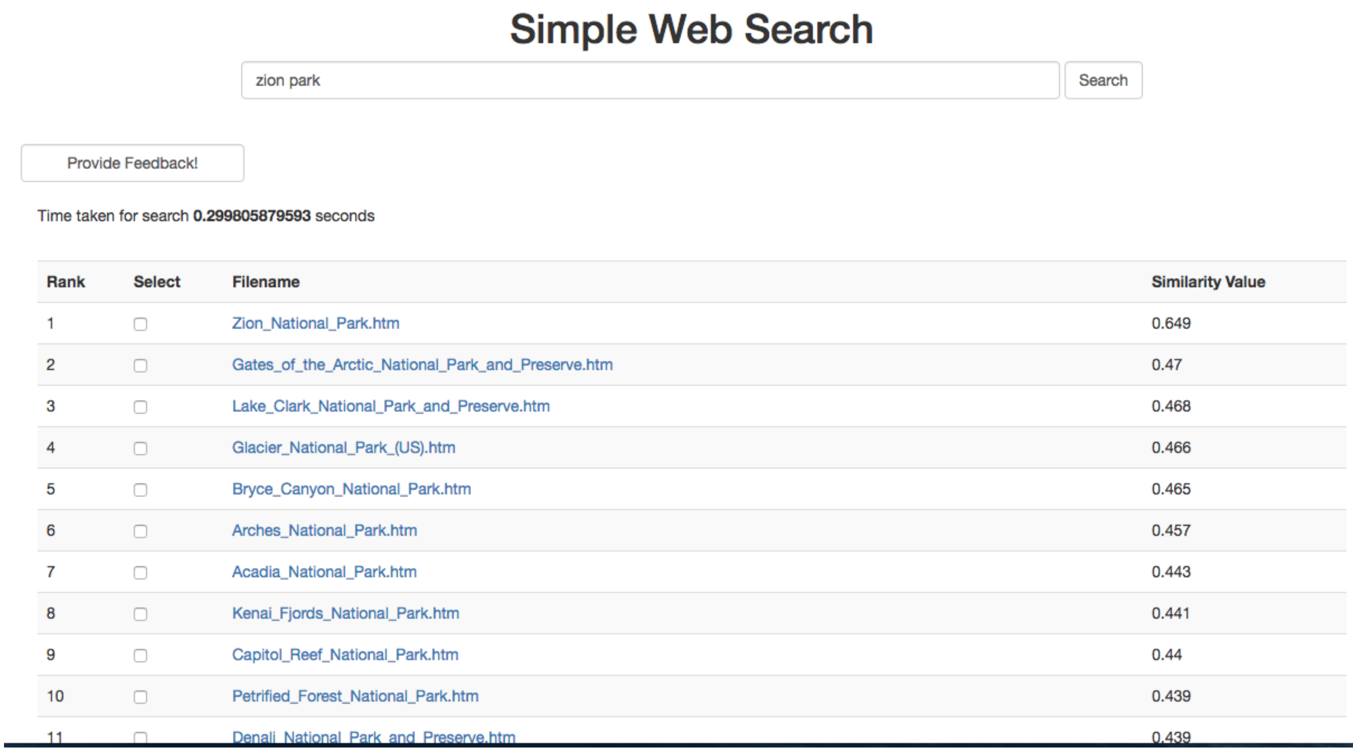
- Our relevance feedback is highly effective. It ranks all documents marked relevant higher than other docs and we see an increase in similarity score for relevant documents as well.

- Only flip side for relevance feedback is, it takes a small performance hit, as we need to compute document vectors for all documents marked relevant

- Using our version of term proximity, increases the effectiveness of our result tremendously. Only flip side is a slight performance hit

- Screenshots are provided below for each flow to show our results.

1. **Search result for Zion Park**



Simple Web Search

zion park

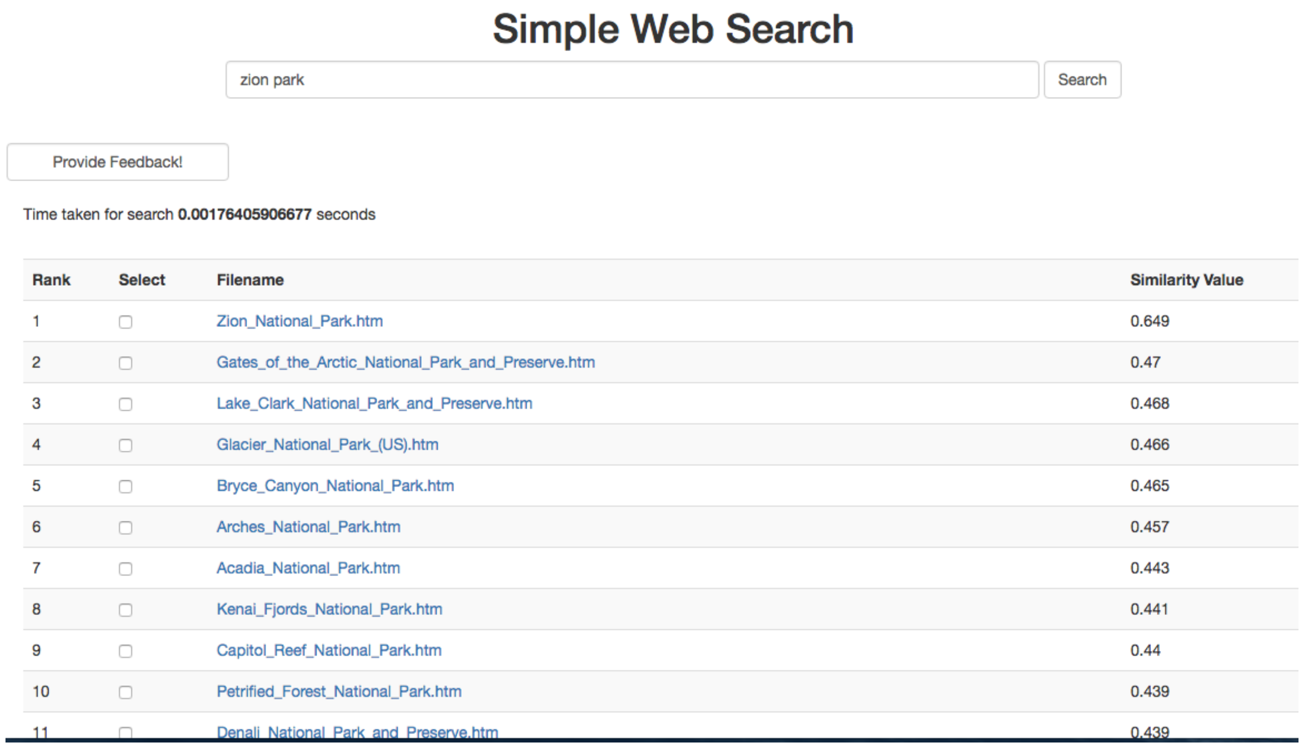
Search

Provide Feedback!

Time taken for search **0.299805879593** seconds

Rank	Select	Filename	Similarity Value
1	☐	Zion_National_Park.htm	0.649
2	☐	Gates_of_the_Arctic_National_Park_and_Preserve.htm	0.47
3	☐	Lake_Clark_National_Park_and_Preserve.htm	0.468
4	☐	Glacier_National_Park_(US).htm	0.466
5	☐	Bryce_Canyon_National_Park.htm	0.465
6	☐	Arches_National_Park.htm	0.457
7	☐	Acadia_National_Park.htm	0.443
8	☐	Kenai_Fjords_National_Park.htm	0.441
9	☐	Capitol_Reef_National_Park.htm	0.44
10	☐	Petrified_Forest_National_Park.htm	0.439
11	☐	Denali_National_Park_and_Preserve.htm	0.439

2. **Searching again with query caching**



Simple Web Search

zion park

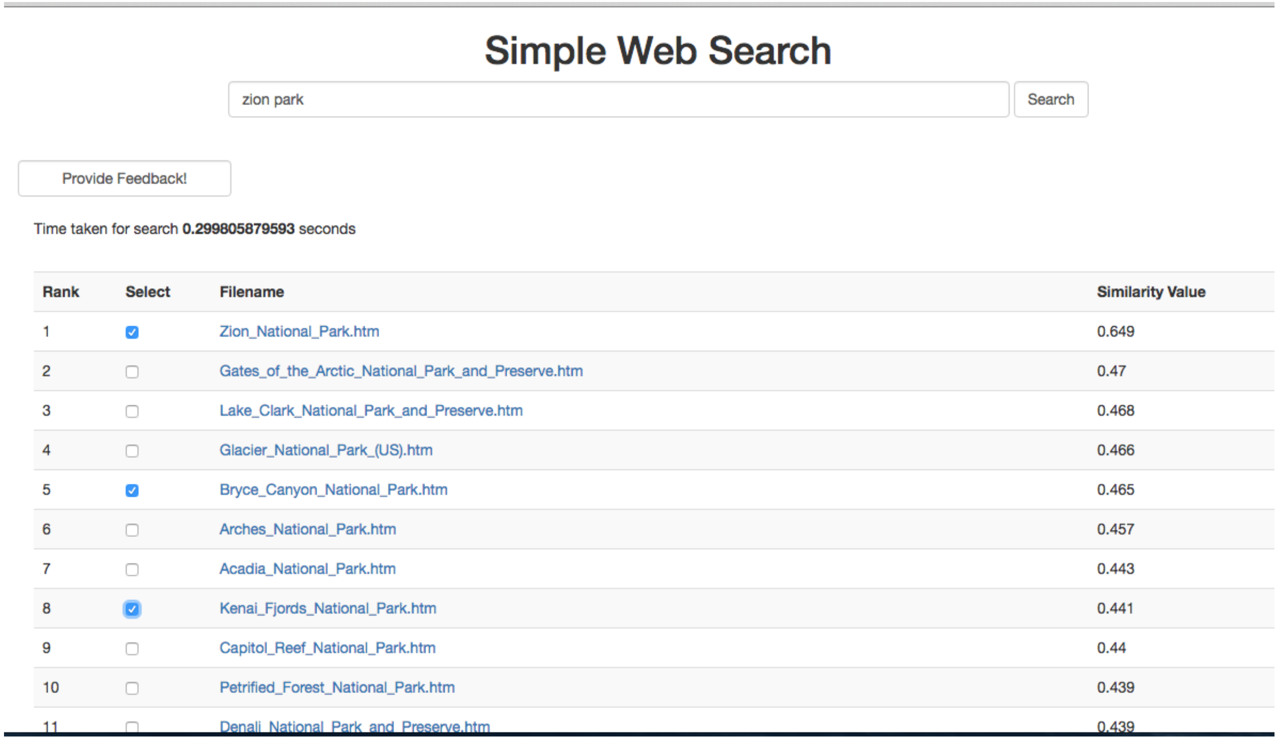
Search

Provide Feedback!

Time taken for search **0.00176405906677** seconds

Rank	Select	Filename	Similarity Value
1	☐	Zion_National_Park.htm	0.649
2	☐	Gates_of_the_Arctic_National_Park_and_Preserve.htm	0.47
3	☐	Lake_Clark_National_Park_and_Preserve.htm	0.468
4	☐	Glacier_National_Park_(US).htm	0.466
5	☐	Bryce_Canyon_National_Park.htm	0.465
6	☐	Arches_National_Park.htm	0.457
7	☐	Acadia_National_Park.htm	0.443
8	☐	Kenai_Fjords_National_Park.htm	0.441
9	☐	Capitol_Reef_National_Park.htm	0.44
10	☐	Petrified_Forest_National_Park.htm	0.439
11	☐	Denali_National_Park_and_Preserve.htm	0.439

3. Marking documents for Relevance Feedback for Zion Park



Simple Web Search

zion park

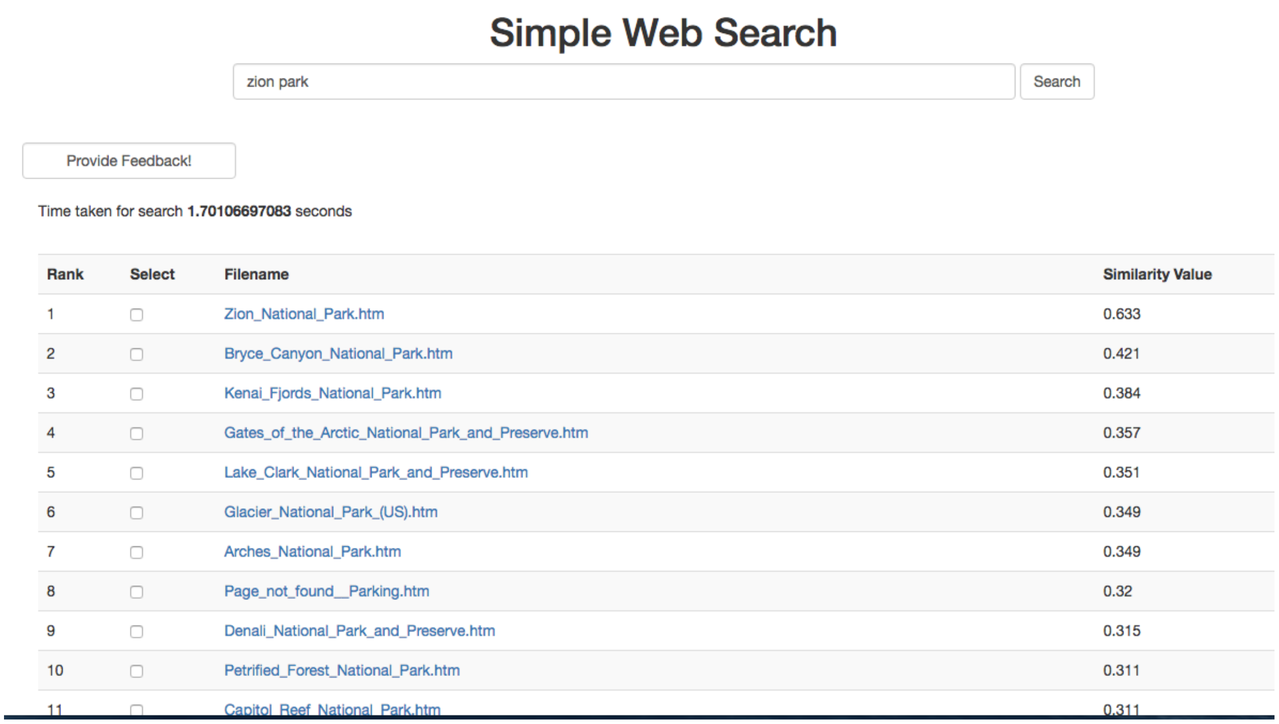
Search

Provide Feedback!

Time taken for search **0.299805879593** seconds

Rank	Select	Filename	Similarity Value
1	✓	Zion_National_Park.htm	0.649
2	☐	Gates_of_the_Arctic_National_Park_and_Preserve.htm	0.47
3	☐	Lake_Clark_National_Park_and_Preserve.htm	0.468
4	☐	Glacier_National_Park_(US).htm	0.466
5	✓	Bryce_Canyon_National_Park.htm	0.465
6	☐	Arches_National_Park.htm	0.457
7	☐	Acadia_National_Park.htm	0.443
8	✓	Kenai_Fjords_National_Park.htm	0.441
9	☐	Capitol_Reef_National_Park.htm	0.44
10	☐	Petrified_Forest_National_Park.htm	0.439
11	☐	Denali_National_Park_and_Preserve.htm	0.439

4. Providing Feedback



Simple Web Search

zion park

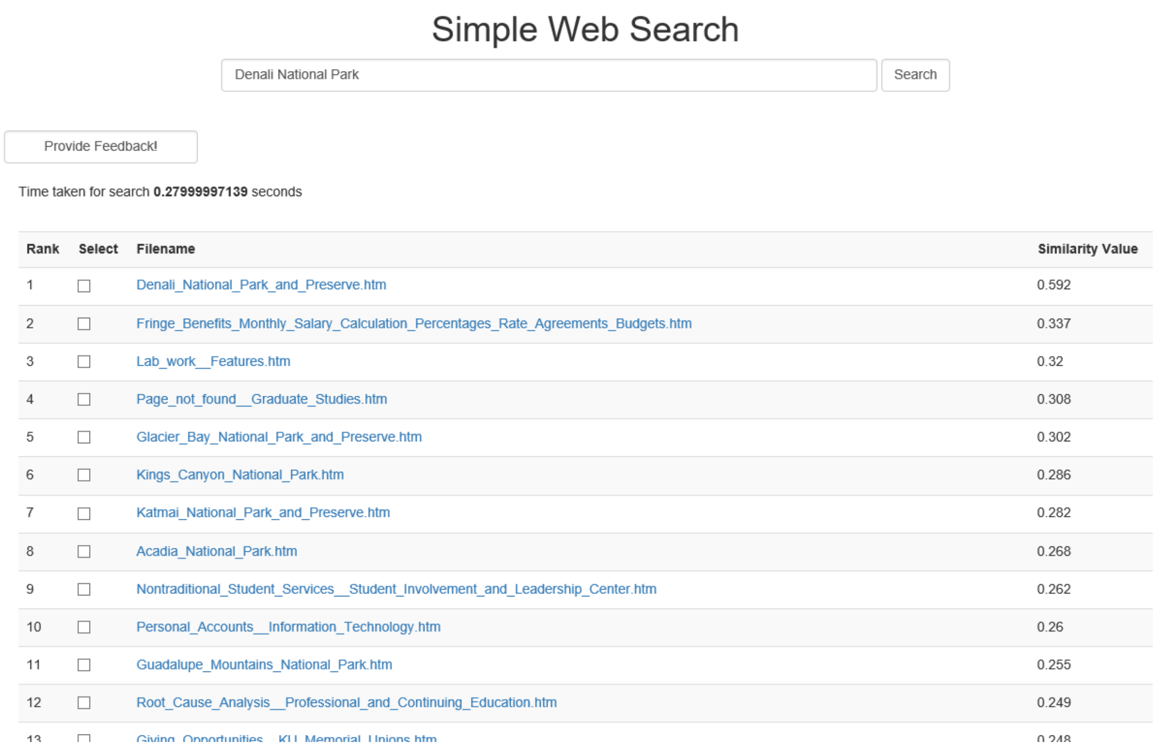
Search

Provide Feedback!

Time taken for search **1.70106697083** seconds

Rank	Select	Filename	Similarity Value
1	☐	Zion_National_Park.htm	0.633
2	☐	Bryce_Canyon_National_Park.htm	0.421
3	☐	Kenai_Fjords_National_Park.htm	0.384
4	☐	Gates_of_the_Arctic_National_Park_and_Preserve.htm	0.357
5	☐	Lake_Clark_National_Park_and_Preserve.htm	0.351
6	☐	Glacier_National_Park_(US).htm	0.349
7	☐	Arches_National_Park.htm	0.349
8	☐	Page_not_found__Parking.htm	0.32
9	☐	Denali_National_Park_and_Preserve.htm	0.315
10	☐	Petrified_Forest_National_Park.htm	0.311
11	☐	Capitol_Reef_National_Park.htm	0.311

5. Without term proximity



Simple Web Search

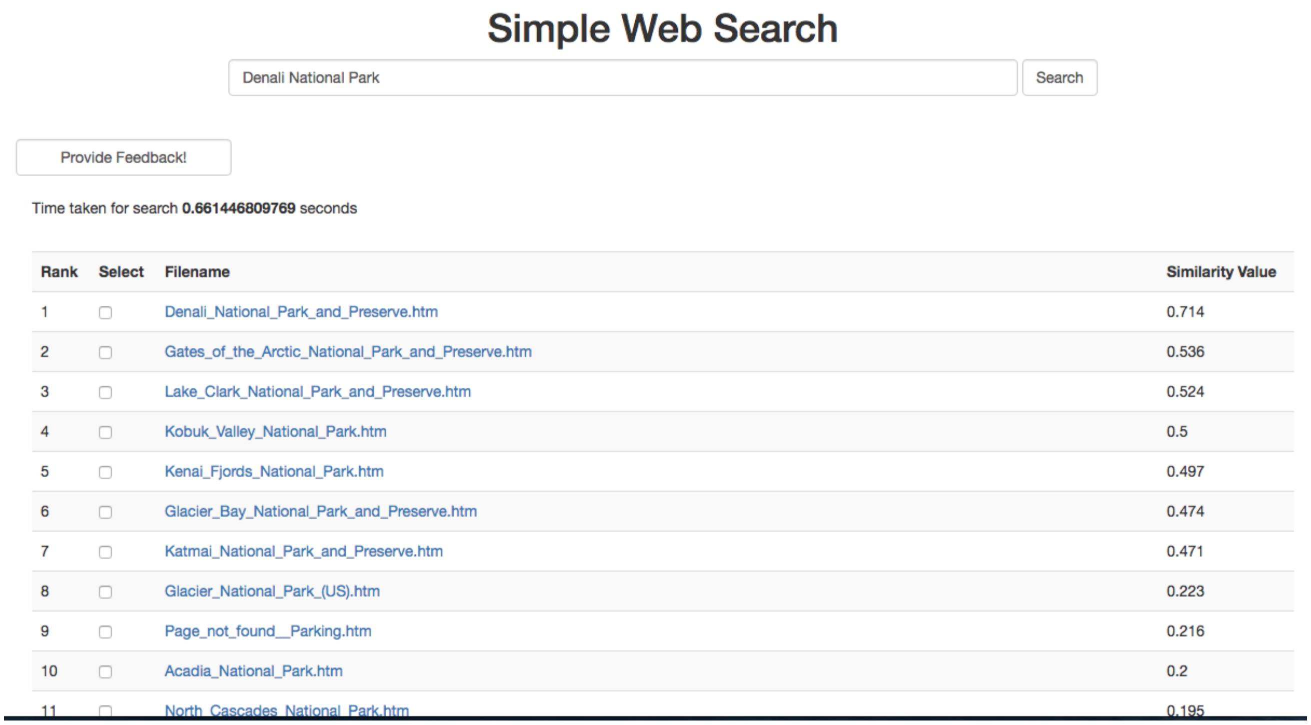
Denali National Park | Search

Provide Feedback!

Time taken for search **0.27999997139** seconds

Rank	Select	Filename	Similarity Value
1	☐	Denali_National_Park_and_Preserve.htm	0.592
2	☐	Fringe_Benefits_Monthly_Salary_Calculation_Percentages_Rate_Agreements_Budgets.htm	0.337
3	☐	Lab_work__Features.htm	0.32
4	☐	Page_not_found__Graduate_Studies.htm	0.308
5	☐	Glacier_Bay_National_Park_and_Preserve.htm	0.302
6	☐	Kings_Canyon_National_Park.htm	0.286
7	☐	Katmai_National_Park_and_Preserve.htm	0.282
8	☐	Acadia_National_Park.htm	0.268
9	☐	Nontraditional_Student_Services__Student_Involvement_and_Leadership_Center.htm	0.262
10	☐	Personal_Accounts__Information_Technology.htm	0.26
11	☐	Guadalupe_Mountains_National_Park.htm	0.255
12	☐	Root_Cause_Analysis__Professional_and_Continuing_Education.htm	0.249
13	☐	Giving_Opportunities__KU_Memorial_Unions.htm	0.248

6. With Term Proximity



Simple Web Search

Denali National Park | Search

Provide Feedback!

Time taken for search **0.661446809769** seconds

Rank	Select	Filename	Similarity Value
1	☐	Denali_National_Park_and_Preserve.htm	0.714
2	☐	Gates_of_the_Arctic_National_Park_and_Preserve.htm	0.536
3	☐	Lake_Clark_National_Park_and_Preserve.htm	0.524
4	☐	Kobuk_Valley_National_Park.htm	0.5
5	☐	Kenai_Fjords_National_Park.htm	0.497
6	☐	Glacier_Bay_National_Park_and_Preserve.htm	0.474
7	☐	Katmai_National_Park_and_Preserve.htm	0.471
8	☐	Glacier_National_Park_(US).htm	0.223
9	☐	Page_not_found__Parking.htm	0.216
10	☐	Acadia_National_Park.htm	0.2
11	☐	North_Cascades_National_Park.htm	0.195