INFORMATION RETRIEVAL

Project Part I

Text-based searching application with Lucene

Oğuz Birdal

Alp Tunçay

Homa Hassannia

30.11.2018

Overview: Lucene

Index construction

Index process is one of the core functionalities provided by Lucene.

Some of the index classes are: indexwriter, directory, analyzer, document, fields.

- Indexwriter is one of the most important classes that creates or updates indexes.
- Directory represent the location of indexes.
- Task of the Analyzer is as the same of its name. It analyzes a document and get the word from the text that is to be indexed.
- Document is the unit of search and index. In fact, we add Documents to IndexWriter, and then retrieve them by IndexSearcher[6].
- Fields: it is a simply a name-value pair which the name or the key is used to identify the value to be indexed.

Inverted indexes

Lucene uses inverted indexes as an index structure to find the document which contains the term in the query instead of searching the text directly. It is similar to the index in the end of the book. Lucene has three types of inverted index strategies which are batch-based, b-tree based and segment based strategies. Each segment index maintains following fields:

- Segment info
- Field names
- Stored Field values
- Term dictionary
- Term frequency data
- Normalization factors
- Term Vectors
- Per-document values
- Deleted documents

Positional indexes

Inverted indexes for retrieving content is not effective, so we should create and store Term Vectors with positions and offset information. Fields store Term Vectors. Lucene's IndexWriter class creates an index file in ".prx" format which contains the list of the positions of each term in the dataset.

Index Compression

Lucene also uses index compression while indexing. Sorted words mostly have same prefixes. In index files, Lucene keeps one of the same prefixes for all and keeps differences sorted after them. It decreases the time of loading a document from disk.

Vector Space Model

It represents text as vectors of term, each number in a vector is the weight of the term. This weight is calculated by term TFIDF (term frequency/inverse document frequency). For scoring Lucene uses Vector Space Model and the Boolean model to find document relevancy to a query. VSM[7] scores the document according to number of appearances of the query in the document. Boolean query being used for determining the documents which will be scored and after VSM is being used for final scoring. The output of that will be a single score that shows how much the document and the query are matched.

Probabilistic Scoring

Lucene also has the support for probabilistic scoring, Okapi BM25. Okapi BM25 is a ranking function used for rank matching the documents based on their relevance to a given search query. Additionally, Lucene has a difference from BM25's IDF. Regular BM25's IDF has a chance to give negative scores but Lucene simply adds one before taking the logarithm and eliminates the chance of getting negative value after taking the logarithm.

In the old versions, Lucene was using a modified version of TF*IDF[4] for probabilistic scoring. For a query which contains a specific word, the document that contains that word is important as how many times the word mentioned in the document. If a word mentioned 10 times in one document and mentioned 5 times in another document, first document is absolutely more important than second one but not twice important. So using inversed document frequency decrease value of popular words for scoring. The modified version of the formula is

*IDF score * TF score * fieldNorms=log(numDocs / (docFreq + 1)) * sqrt(tf) * (1/sqrt(length))*

Query Expansion

LucQE[5] is the query expansion module of the Lucene. The aim of the query expansion is improving the recall and precision. Similar terms are taken into query to do this. The most common module for query expansion in Lucene is Rocchio Query Expansion method.

Spell Checker

Lucene has support for spell checker too and it comes with the Lucene release located in contrib/spellchecker. It's working principle is building a second index according to an existing field from main index file. In our application, we do the spell checking for one word queries but the mechanism does not work with the phrase queries.

Dataset

We decided to use "Enron E-mail Dataset" in this project. It contains approximately half million messages from 150 users. Dataset published by Federal Energy Regulatory Commission. Messages in the dataset does not include attachments and some of them removed due to privacy issues. We arbitrarily discarded a part of dataset due to its size. Right now, the dataset we use contains approximately 100,000 files.

What We Have Done

For building this application we had 2-two choices for languages: Java and Python. Developing using Python required extra installations. Since Lucene is a Java based library, we built our application using Java. Our application provides basic search capabilities and returns documents that the query appears on. It also has the functionality for suggesting alternative query to the user. For this suggestion system, we make use of a dictionary of 10000 English words taken from [8]. If the searched term does not appear in this dictionary, then the system suggests an alternative for that. In this section, we will be discussing the components of the system in detail.

The main components in our application are: Indexer, Searcher and Suggestion. These three classes are the main actors behind the functionality of our application. Indexer is the class that is responsible for indexing the data to be searched. Searcher is the class that reads the index files saved to the index directory and conducts the search on these files based on the query given by the user. Suggestion is the part that works on the spell checks and suggestions. In this component, we are also indexing the dictionary file that we have. Besides indexing, it has the functions that are responsible for checking whether the query term is in the dictionary and finding out what to suggest to the user. Let us dive into these classes and discuss them in more detail.

Indexer

In this class we make use of five fundamental classes that Lucene provides us: Document, Field, IndexWriter, IndexWriterConfig and StandardAnalyzer. These are the classes that we mainly use for the indexing purposes. As it is explained on tutorial from [1], the way the indexing works is as follows:

1. We have to specify the directory to store the indexes. For that purpose, we use FSDirectory class in Lucene which stores the files in the file system.
2. Then we create an instance of IndexWriterConfig class which takes as input which analyzer would be used while indexing. In our application we chose to use StandardAnalyzer since it can recognize emails. After creating the IndexWriterConfig, we feed this instance into IndexWriter.
3. In Lucene each Document consists of Fields. So at first, we have to convert the data we have into Lucene Document. For that purpose, we read the content of the file, file name and the file path and converts these into Field and then add each field to Document object we have created.
4. After generating Document objects from the files, the remaining part of the indexing is adding each of these Documents to index by using the addDocument() method from the IndexWriter class.

Searcher

In this class, we used the classes provided by Lucene such as IndexSearcher, QueryParser, IndexReader and Query. The explanation of this searching process can be found on [1] and it is as follows:

1. Initially, we specify the folder which the indexes are stored. We again use FSDirectory here.
2. The IndexReader is initialized using its own open() function. The Directory that we just specified is given as input to this open() function.
3. We create the instance of IndexSearcher class using the previously created reader as the input to the IndexSearcher's constructor.
4. Following these, we have to create an instance of the QueryParser class. The input that we read from the user is of type String and the QueryParser gives us the necessary functions for interpreting this String into the Lucene's Query class.
5. For search operations, first we parse the String to Query class and then this instance of Query is fed to the search() function that is inside IndexSearcher class. From this operation we obtain an object which is of type TopDocs class. In this TopDocs object we have the top hits that are returned for the query.

Suggestion

This is the class that we use for the suggestions that are given to user. Here we make use of SpellChecker class from Lucene. In this class, we basically do indexing and finding the similar words with respect to the user's input. For finding the similar words, we use the Levenshtein Distance which is also referred to as edit distance. The workflow is as follows.

1. First we start by specifying the directory for the indexes of the dictionary we have. Again the directory here is FSDirectory.
2. We continue with the initialization of the SpellChecker class. The input to its constructor is the directory that we created.
3. Then, we create the IndexWriterConfig using the StandardAnalyzer as in Indexer class.
4. By using the indexDictionary() function from the SpellChecker class, we index the dictionary. The inputs of the indexDictionary() function are the IndexWriterConfig, PlainTextDictionary and a Boolean. The PlainTextDictionary is basically a dictionary which is represented by a text file. In our case, the text file is the "words.txt" file.

Sample Queries

In this example, since it was our first query system had to index the files first. After indexing has ended, system yielded the files that the search query appeared. In total 98192 files were indexed and it took approximately 10 minutes to index them. The search itself took 62 ms and in that time 19701 documents were found but only the top 10 of those files were presented to the user. The ordering of those files are based on the scores of the files. The console output can be seen in the picture.

```
best regards
Started indexing...
98192 File indexed, time taken: 599569 ms
19701 documents found. Time :62
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\beck-s\all_documents\211
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\beck-s\congratulations\19
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\beck-s\discussion_threads\188
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\beck-s\resumes\5
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\beck-s\all_documents\1297
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\beck-s\all_documents\1313
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\beck-s\all_documents\1669
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\beck-s\all_documents\214
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\beck-s\congratulations\16
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\beck-s\discussion_threads\1473
```

In this screenshot, we mistyped the word “hello” intentionally just to see if the application can suggest us an alternative option. For this suggestion, firstly our app searches the query word in a dictionary which includes 10000 words, if dictionary does not contain the query word then it uses the Levenshtein distance to find a suggestion and returns the word that is closest to the query we typed.

```
Enter search query:
hwllo
Did you mean: hello if yes type Y and hit enter, other wise hit enter.
Y
1592 documents found. Time :153
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\causholli-m\sent_items\189 Query: hello
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\brawner-s\all_documents\131 Query: hello
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\brawner-s\discussion_threads\107 Query: hello
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\brawner-s\sent\75 Query: hello
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\brawner-s\_sent_mail\94 Query: hello
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\bass-e\all_documents\1678 Query: hello
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\bass-e\sent\998 Query: hello
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\bass-e\_sent_mail\1096 Query: hello
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\baughman-d\deleted_items\388 Query: hello
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\dasovich-j\all_documents\4179 Query: hello
```

In this query example, we tried to use a long query including 8 words and typed some of the words wrong on purpose. Our suggestion algorithm can not handle long queries so it does not suggest anything and search the query exactly in this kind of situations.

```
Enter search query:
this email is automated notification of the availability
14743 documents found. Time :185
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\allen-p\all_documents\3 Query: this email is automated notification of the availability
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\allen-p\discussion_threads\231 Query: this email is automated notification of the availability
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\allen-p\notes_inbox\3 Query: this email is automated notification of the availability
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\allen-p\deleted_items\111 Query: this email is automated notification of the availability
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\allen-p\deleted_items\27 Query: this email is automated notification of the availability
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\allen-p\deleted_items\6 Query: this email is automated notification of the availability
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\allen-p\inbox\11 Query: this email is automated notification of the availability
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\allen-p\inbox\17 Query: this email is automated notification of the availability
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\allen-p\inbox\18 Query: this email is automated notification of the availability
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\allen-p\inbox\43 Query: this email is automated notification of the availability
```

We also made some sample searches using wildcard queries. Lucene has support for wildcards so in this example we searched “presen*” as our wildcard query and it returned documents which contains the words initiates with “presen” like present, presents, presentation etc. Firstly, we hit yes for the suggestion.

```
Enter search query:
presen*
Did you mean: [present]. If yes type Y and hit enter, other wise just hit enter.
y
2673 documents found. Time :19
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\dasovich-j\all_documents\4673 Query: present
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\dasovich-j\notes_inbox\2521 Query: present
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\campbell-l\all_documents\171 Query: present
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\campbell-l\discussion_threads\138 Query: present
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\campbell-l\sent\68 Query: present
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\arnold-j\deleted_items\523 Query: present
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\buy-r\all_documents\203 Query: present
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\buy-r\sent\132 Query: present
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\buy-r\_sent_mail\114 Query: present
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\carson-m\all_documents\81 Query: present
```

After we did not accept the suggestion and searched the query itself.

```
Enter search query:
presen*
Did you mean: [present]. If yes type Y and hit enter, other wise just hit enter.

7835 documents found. Time :23
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\beck-s\inbox\392 Query: presen*
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\beck-s\inbox\4 Query: presen*
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\beck-s\inbox\400 Query: presen*
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\beck-s\inbox\420 Query: presen*
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\beck-s\inbox\422 Query: presen*
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\beck-s\inbox\426 Query: presen*
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\beck-s\inbox\428 Query: presen*
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\beck-s\inbox\441 Query: presen*
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\beck-s\inbox\447 Query: presen*
File: C:\Users\Alp\Documents\Workspace(Spring)\IR_Project1\.\data\beck-s\inbox\448 Query: presen*
```

As it can be seen, when we used the wildcard term for querying, the number of documents that was returned increased. With the wildcard “*”, Lucene searches for terms that contain the prefix “presen” and does not care whatever comes following the prefix. This increases the number of documents that were found.

References

[1] https://www.tutorialspoint.com/lucene/lucene_first_application.htm

[2] https://lucene.apache.org/core/3_6_2/api/all/index.html

[3] http://www.lucenetutorial.com/lucene-in-5-minutes.html

[4] https://opensourceconnections.com/blog/2015/10/16/bm25-the-next-generation-of-lucene-relevation/

[5] http://lucene-qe.sourceforge.net/

[6] https://lucidworks.com/2009/05/26/accessing-words-around-a-positional-match-in-lucene/

[7] https://lucene.apache.org/core/2_9_4/scoring.html#Scoring

[8] https://github.com/first20hours/google-10000-english