Lunr.js - A JavaScript-Based Text Processing Toolkit

CS 410 Technology Review

Xiaohan Tian (xtian13@illinois.edu)

1. Introduction

Lunr.js is a JavaScript-based text processing toolkit. The codebase is very small but provides rich features. It doesn't have any 3rd party dependencies, and it's ideal for embedding into any lightweight applications.

Lunr.js offers inverted indexing, stemming, tokenization, and complex querying, and it even allows users to use customized weights (weight boosting) in runtime when scoring documents. Additionally, the entire pre-processing pipeline can be customized.

But as a JavaScript-based lightweight toolkit, it also has some limitations. By default, the index is entirely in-memory, although in modern days, the memory is large enough for indexing all personal data such as bookmarks; it's not suitable for indexing a huge amount of data such as a general-purpose webpage search engine. It also doesn't have a built-in data-persistence solution.

This article will discuss both features and limitations of Lunr.js through code analysis and a series of performance tests.

2. Code Implementation Analysis

2.1 Architecture

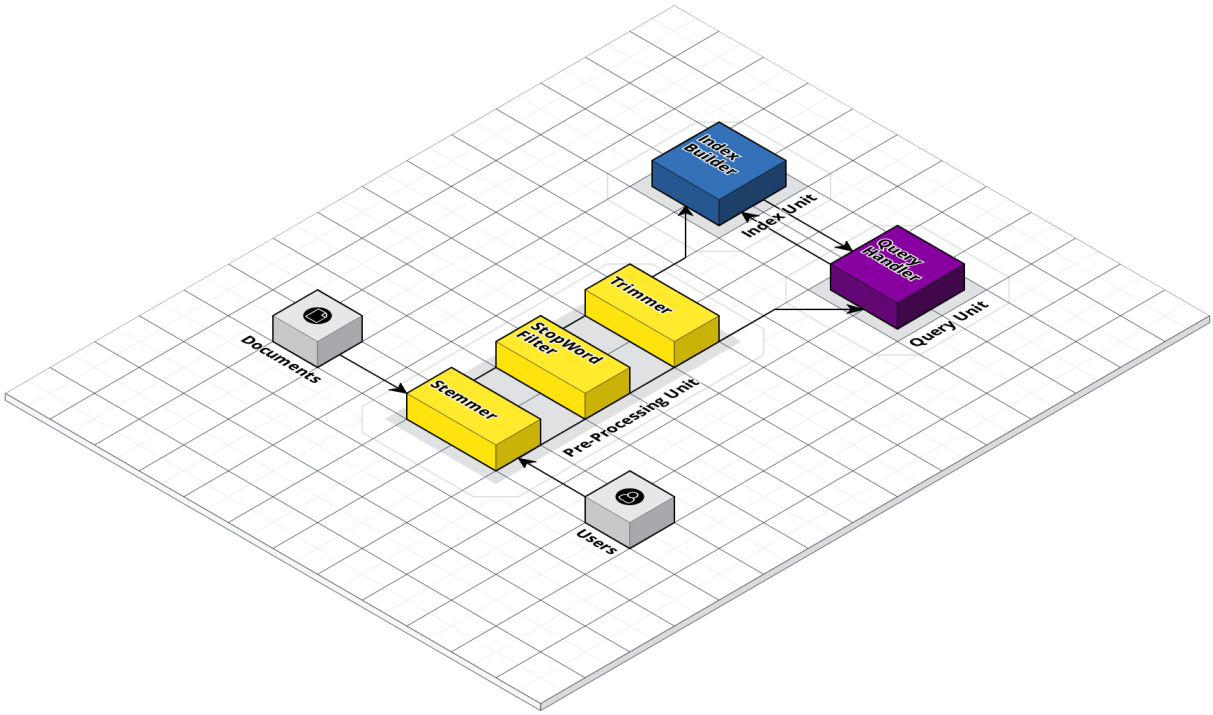
By analyzing the source code, it can be found that Lunr.js can be roughly divided into three different major components: pre-processing unit, index unit, and query unit. Also, it contains a few utility components, such as the vector computation module.

The pre-processing unit provides a pipeline that can be used to convert raw text strings into integer-based tokens. By default, the pipeline contains three modules: stemmer, stop word filter, and trimmer. The stemmer will process words and convert them into stem form, which can be used to reduce the total number of terms. In lunr.js, it uses a JavaScript implementation of PorterStemmer (please see reference for the detailed algorithm) initially invented by Martin

Porter. The stop word filter is a simple function that can remove words coming from a stop word list from the original content. Lunr.js has a built-in stop-word list that contains 118 common words. The trimmer is a module that removes starting and ending non-word characters from terms; this step is required before sending tokens into the indexing unit. Lunr.js uses regular expressions to perform this operation.

The Indexing unit contains the logic to index documents into inverted index form, and it also provides an interface to accept queries. When building the index, Lunr.js also pre-computes different values such as TF, IDF, averaged doc length, etc. As mentioned earlier, Lunr.js doesn't provide a way to persist the index data, also it always uses the in-memory index, but it provides a way to convert its index data into JSON format; this enables the user to easily extend the system to save and load an index from persistent storage. Please see section 2.2 for the detailed data structure Lunr.js is using.

The querying unit is capable of handling different types of queries; it supports multi-word queries, leading and trailing wildcards, queries within a scope, term boost, and term presence modifiers. The querying unit will read the query command and convert it into a query request, then query it from the index. The similarity comparison is using the vector space model, and the vector computation uses its own pure JavaScript-implemented algorithms. Please see section 2.2 for the detailed scoring function.



(Figure 1: Lunr.js High-Level System Architecture)

2.2 Models and Algorithms

2.2.1 Inverted Index

In Lunr.js, the main data model is the inverted index. It uses a four-level map to store the combined dictionary table and posting table:

```
invertedIndex[term][fieldName][docRef][metadataKey]
```

Compared with the raw implementation of the dictionary table and posting table, this data structure is very suitable for in-memory storage, it allows fast query and avoids record offset computation since it is a straightforward multi-level map, the query speed is very fast. Additionally, compare with the raw implementation of the dictionary table and posting table, it introduced the concept of “Field”; this allows the user to use multiple fields when indexing documents and also can support querying in scope feature.

2.2.2 Similarity Comparison

By reverse engineering the code, it can be observed that the scoring function Lunr.js is used for single terms:

$$\text{score}_w = \frac{(k1+1)\ \text{IDF}\,(w)\ \text{TF}\,(w)}{k1\left(1-b+b\frac{|d|}{avdl.}\right)+\text{TF}\,(w)}$$

Where:

$$\text{IDF}\,(w) = log\left(1+\left|\frac{M-k+0.5}{k+0.5}\right|\right)$$

And the final score for the query:

$$\text{score} = \sum_{w\in V}\left(\text{score}_w \cdot \text{boost}_w\right)^{c(w,q)}$$

Note:

- M is the total number of documents
- k is the number of documents with the given term
- $boost_w$ is the boosting weight of the given term w

3. Performance Evaluation and Comparison

3.1 Environment and Methodologies

For performance evaluation, I'm using a standard `t2.small` AWS EC2 instance. The `t2.small` AWS EC2 instances provide:

- 1 vCPU (Intel Xeon Haswell E5-2676)
- 2 GB RAM
- 20 GB SSD Storage (Configurable)

For a fair comparison, every test will be executed based on a cleaned installation from OS image `Canonical, Ubuntu, 22.04 LTS, amd64 jammy image build on 2022-09-12` with the following additional components:

- Oracle JDK 17.0.4.1
- NodeJS 12.22.12

Below sections are comparisons between Lunr.js and Lucene. Lucene is a Java-based text processing toolkit developed by Apache Software Foundation, and it is one of the most commonly used search engines in the industry. Please see the reference section for more details.

Each of the following tests will be executed ten times by using shell scripts, and the computed averaged value will be taken.

The test dataset is created by me, and it's based on my own bookmarked webpages. Each page has been captured by using Puppeteer and converted into plain text format. The Query judgment test set was created by using Cranfield Evaluation Methodology.

3.2 Indexing Performance

	Lunr.js	Lucene
10 documents	141 ms	1188 ms
100 documents	968 ms	2369 ms
1000 documents	4659 ms	3617 ms

It shows that the Lucene has some "warm up" time in the beginning; it becomes very fast after stabilized. But Lunr.js also can provide a relatively fast indexing speed.

3.3 Query Performance

	Lunr.js	Lucene
1 query	11 ms	208 ms
5 queries	37 ms	257 ms
10 queries	65 ms	391 ms

It shows for the query performance, Lunr.js is significantly faster than Lucene.

3.4 Result F1 Score

Lunr.js	Lucene
0.86265	0.87428

The F1 score between Lunr.js and Lucene doesn't have any major difference.

4. Potential Enhancements

Lunr.js comes with a few ways to customize the toolkit itself. The major way is to use the plugin system. Besides plugins, it also allows users to customize the pre-processing pipeline, token metadata, and tuning of two parameters $k1$ and b in the scoring function.

4.1 Lunr Plugins

The Plugin system is the major way to enhance Lunr.js. It allows users to inject their own logic into the processing when building the index, a very similar concept to the filters in the J2EE web applications.

The plugin itself doesn't have any special format requirements; it can be a simple JavaScript function. Those plugin functions will be executed during the indexing process, and the context parameters will be provided. Also, the runtime index builder instance will also be provided.

It is a very useful mechanism to enhance Lunr.js but it is limited to the indexing process, and it's hard to use the plugin system to persist the data.

4.2 Code Customizations

Lunr.js is under MIT license (see https://github.com/olivernn/lunr.js/blob/master/LICENSE). Thus, it is permitted to modify the source code for personal needs.

By code analysis, there are multiple parts that can be easily customized:

- `lunr.Pipeline` is the place to put any customized text processing logic.
- `lunr.stemmer` is coded using PorterStemmer method, but it can be easily replaced by other methods.
- `lunr.generateStopWordFilter(stopwords)` can set the stopwords, and any additional stopwords can be added to the default list.
- `Vector.similarity` is used for computing vector distance. By default, the similarity is computed as:

```
this.dot(otherVector) / this.magnitude() || 0
```

  It is possible the modify the code to use different similarity comparison methods.
- The index data can be serialized into JSON format. It is possible to save the JSON into a file on a persistent device. The JSON formatted index data can be loaded using `lunr.Index.load(s)` method.

5. Conclusion

Lunr.js is a powerful text processing toolkit. It's written in pure JavaScript, and it doesn't require any 3rd party dependencies. Overall it's very easy to use and ideal to embed into the browser for a quick in-memory indexed search engine. Although there are some limitations, it is possible to customize the original code in order to support features such as customized similarity comparison and data persistence.

6. Reference

- Lunr.js official website
  - https://lunrjs.com/
- Lunr.js customization
  - https://lunrjs.com/guides/customising.html
- Lucene official website
  - https://lucene.apache.org/
- Stemming and lemmatization
  - https://nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatization-1.html
- Porter Stemmer Algorithm
  - https://tartarus.org/martin/PorterStemmer/
- Puppeteer
  - https://pptr.dev/

- The BM25 Algorithm
    - https://www.elastic.co/blog/practical-bm25-part-2-the-bm25-algorithm-and-its-variables
- Overview of ECMA 6 Features
    - https://github.com/lukehoban/es6features