Eindhoven University of Technology

Title goes here

Stefan Patelski

November 10, 2013

Contents

1 Introduction **3**

2 Scope and Architecture **3**
2.1 Scope . . . 3
2.2 Architecture . . . 3

3 Information Retrieval **3**
3.1 Crawler : S.R. Patra . . . 3
3.1.1 Motivation . . . 3
3.1.2 Approach . . . 3
3.1.3 Challenges . . . 3
3.1.4 Evaluation & Summary . . . 4
3.2 Lucene Search, Wildcard Queries & Autocomplete : S.R. Patra . . . 4
3.2.1 Motivation . . . 4
3.2.2 Approach . . . 4
3.2.3 Evaluation & Summary . . . 5
3.3 Tf-idf: F.N. Butnariu . . . 5
3.3.1 Motivation . . . 5
3.3.2 Approach . . . 6
3.3.3 Argumentation for choice . . . 7
3.3.4 Evaluation of obtained results . . . 7
3.3.5 Conclusions . . . 7

4 Machine Learning **7**

5 Conclusion **7**

1 Introduction

2 Scope and Architecture

2.1 Scope

2.2 Architecture

3 Information Retrieval

3.1 Crawler : S.R. Patra

3.1.1 Motivation

The very first task was to collect data for our project. We decided to implement a focused crawler for Rotten tomatoes website to collect information about different movies. Large collection of movies with credible reviews made Rotten Tomatoes a suitable candidate for us to get our data from. The goal was to start with a seed URL and extract all the yet unseen URLs on that HTML page along with the metadata about the movie such as movie name, director, producers,reviews etc. The process needs to be repeated for all the collected links.

3.1.2 Approach

For each page, the method *CollectData* in the crawler read the HTML content line by line, only collecting the relevant information and disregarding all the other content that were not relevant to us. This was done with the help of an HTML parser which used *Jsoup* library functions. Whenever relevant data were encountered, they were appended to an XML document. To find the movie URLs , Java's pattern matching functionality was being used which looks for a particular pattern of anchor texts which are specific to movie pages. The crawler architecture is shown in fig.1.

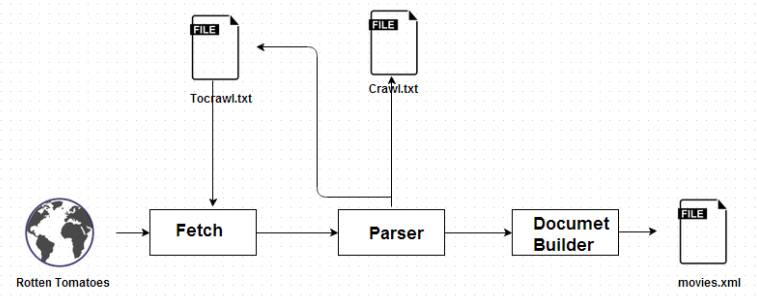

Figure 1: Crawler Architecture

3.1.3 Challenges

The main challenge was to encounter the non uniformity of the content in Rotten Tomatoes web pages. Some of the web pages contained all the information about the movies while in others, a few of the information fields were missing. The crawler was designed to scrap the HTML data in a way such that maximum amount of information can be retrieved. Another challenge was to make sure that the crawler is not blocked by the site. This was handled by

building the crawler in a flexible way so that, during an iteration, the crawler will only crawl through a specific number of pages specified to it as a parameter. After each iteration, the URLs which are already crawlled and the URLs those are yet to be crawlled are written back to *Crawl.txt* and *Tocrawl.txt* respectively which will be used in the next iterations. Fig.2 shows an entry in the *movies.xml* file.

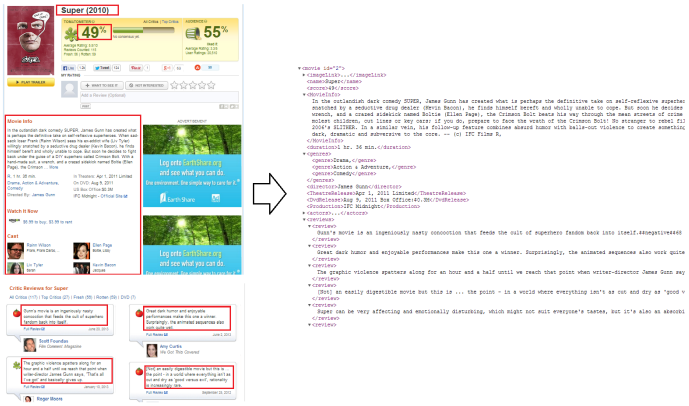

Figure 2: Collecting data from Rotten Tomatoes

3.1.4 Evaluation & Summary

After several iterations, we collected 6227 movies each containing movie metadata and the reviews. We collected only top critics reviews because we feel that those are the most representative of all the reviews. As focused crawler parse all the content in documents and filter only the relevant content for the application, crawling took more time than a normal crawler would take. But the flexibility in the design of the crawler allowed us to run it in iterations so we did not face any issue regarding the time for crawling. We took 20 random samples from the file and compared the contents with the content in the site. In all the cases, we were able to capture all the relevant data needed for the search and sentiment analysis functionality.

3.2 Lucene Search, Wildcard Queries & Autocomplete : S.R. Patra

3.2.1 Motivation

Lucene provides very rich library functions to implement indexing and search. So we implemented lucene search as a metric to evaluate our boolean and vector space model performances.

3.2.2 Approach

Lucene.net library was used for generating the index and for getting the search results from the processed query. We limited our search results to be 10 top relevant results. Index was created for different fields to allow the user to search based on any criteria they want i.e. based on actor, director, genres etc. We also implemented the *Autocomplete* feature to help the user with the query. This feature was developed in *JQuery* and was implemented as a

web service which queries the database about possible movie names after each input letter in the text box.

Wild card queries were used to provide query suggestions to the user when the user only knows parts of the terms in the query. The speech to text query assistance feature was also included using the HTML5 Webkit speech. All the features discussed above are shown in fig.3.

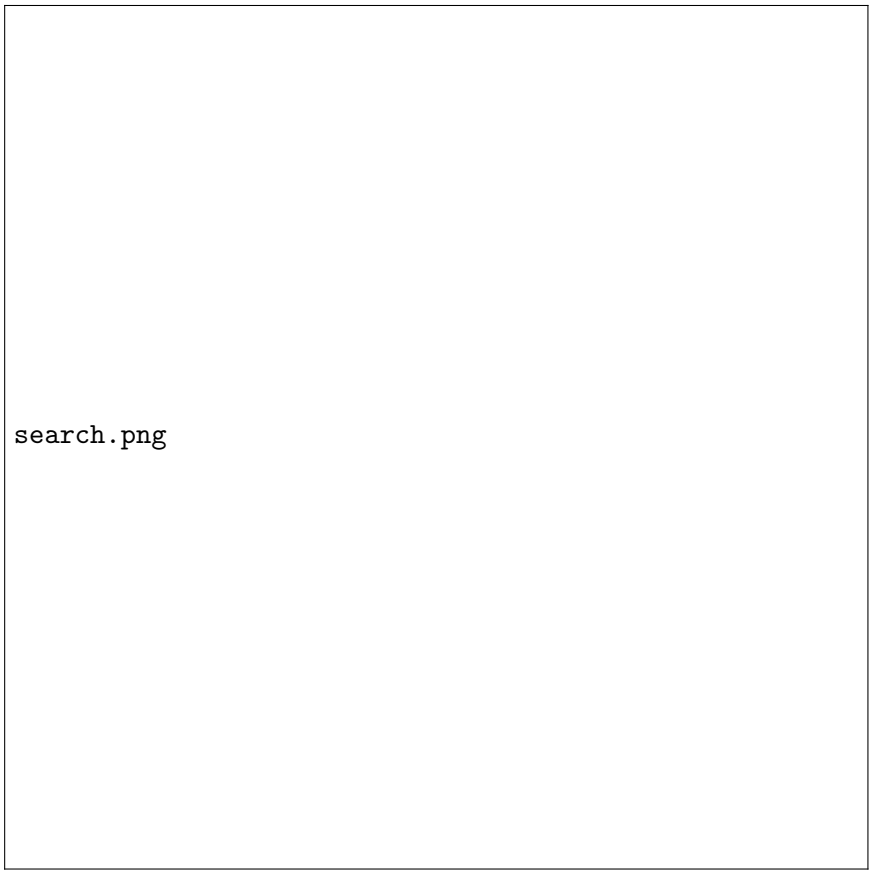

Figure 3: Searching functionalities

3.2.3 Evaluation & Summary

After trying a number of sample queries for different search criterias, we feel that the search provided quite relevant results in all the cases. The wildcard queries were also accurate in their suggestions and the users were able to retrieve relevant movie names based on parts of the search query terms.

3.3 Tf-idf: F.N. Butnariu

3.3.1 Motivation

Initially the tf-idf component was developed to primarily support the vector space model, but subsequently it became obvious that it will also play an important role in the hierarchical

agglomerative clustering task, where cosine similarity was employed to measure resemblance of movies based on information such as title, description, genre, director and cast.

Considering our dataset of movies in this task we aimed to assign a tf-idf weight to each term in each document pertaining information about a movie. The input may be viewed as a collection of documents, where a document contains metadata about a single movie such as title, description, genre, director, cast, reviews and other details. The output is a set of triples `<term, ID, weight>`, where `ID` identifies a document in our collection in which `term` has a tf-idf value given by `weight`.

3.3.2 Approach

The entire information about our set of movies resides in a single XML file, so the first challenge was to transpose this information into a document representation corresponding to each movie in our dataset. This was accomplished using a SAX parser, which basically created for each movie a String to which all the relevant information about the movie was appended. Therefore, at this stage each document pertaining information about a movie can be viewed as a large String. In the next stage by employing one of Lucene's text analyzers each String corresponding to a movie was tokenized into a bag of words. We do not take into account the order of the words within a document, but instead only consider the number of occurrences of a word inside a document. Here we use *term* and *word* interchangeably.

At the heart of the tf-idf component lies an index having the structure represented in Listing 1. The index records for each term its document frequency, i.e. the number of documents in our collection in which term occurs. In addition, for each term a list of pairs `<ID>, <term frequency>` is stored. For every document identified by `ID`, in which the term is present, there is a corresponding term frequency, i.e. the number of occurrences of the term inside the document.

Listing 1: The structure of the tf-idf index

```
<term>, <document frequency>:
        (<ID>, <term frequency>;
        <ID>, <term frequency>;
        ...
        <ID>, <term frequency>; ...)
<term>, <document frequency>:
        (<ID>, <term frequency>;
        <ID>, <term frequency>;
        ...
        <ID>, <term frequency>; ...)
```

This index was built by going over each document in our collection in one pass and examining each term within a document. Thus it also functions as a dictionary since it encompasses all the terms found in our document collection.

By making use of this index a tf-idf weight can quickly be calculated using Equation 1.

$$tfidf_{t,d} = tf_{t,d} \times log\frac{N}{df_t} \tag{1}$$

The set of triples `<term, ID, weight>` can be computed by going over the entire index in one pass.

3.3.3 Argumentation for choice

Parsing the XML file holding our dataset of movies presented us with two options, namely, either using a SAX parser or a DOM parser. We opted to employ the former approach because of its reduced memory overhead in comparison with the latter technique, which would load in memory a costly tree data structure for the entire XML file.

Following the same line of reducing memory overhead and keeping in mind that in-memory operations are much faster than disk access operations, we decided to use the index described earlier to compute tf-idf weights on the fly instead of relying on a much larger data structure, which would have hold for every possible term and for every possible document a tf-idf weight.

Although the tf-idf weighting scheme is designed to assign a low weight to terms appearing in almost all documents, we opted to leave out stop words in most of the cases.

3.3.4 Evaluation of obtained results

3.3.5 Conclusions

4 Machine Learning

5 Conclusion