Course Project

1 Introduction

The following document aims to illustrate and explain some of the details of MovieSearch, my tool. In the next sections, I explain some of the design choices, websites crawled, as well as some architectural choices.

To get a better overview of how to run the tool, please read the `README` embedded in the project folder.

Before you go any further, if not yet done, it is recommended to download the repository at `https://github.com/pennal/\tool`

2 Components

2.1 Crawling

The first step was to obtain the information required for the tool. To do so, I used Apache Nutch. The configuration was left quite standard, only for a few additions that are explained in the next sections. In general, the number of rounds was left at 20 or 30, which I found to be a rather acceptable parameter. This allowed me to gather enough information to have a working tool.

Crawling was done independently for each of the websites. This was done to parallelize the work, and therefore speed up the process. Also, this allows me to keep an eye on exactly what comes from where, and have a better overview on the size of the crawling process.

Once done, the segments for each of the websites were taken and dumped to an html file. This made it easy to extract the information needed from each of the websites. This is better explained in the next section.

2.1.1 IMDB

The structure of the website is quite simple, and easy to crawl. The only issue I encountered were the external links that are embedded in almost every page through the use of widgets. To avoid this situation I modified the regex to only include those pages which pointed to an actual movie. The following RegExes were used:

```
-(.*(?:sort).*)
-(.*(?:view).*)
+(^(?:http|https)\:\/\/(?:www\.)?imdb.com\/title\/(?:tt\d*)\/(?:(?![\w\d])|connections))
+((?<![\w\d])^(?:http|https)\:\/\/(?:www\.)?imdb.com\/(?![\w\d]))
+(^(?:http|https)\:\/\/(?:www\.)?imdb.com\/search\/title(?![\w\d]))
-.
```

2.1.2 AllMovie

This is another website which is easy to crawl. To do so minimal modification was required to the basic regexes. The only issue is with the URLs, which always contain the `/movie` word in their URLs. Therefore it was impossible to distinguish the movies form the TV Shows. This cleaning was done later on in the cleaning phase. The following RegExes were used:

```
+(^(?:http(?:s?))\:\/\/(?:w{3}\.)?allmovie\.com\/(?:genre|movie)\/)

# Refuse anything else
-.
```

2.1.3 RottenTomatoes

Due to the structure of Rotten Tomatoes, it was unfortunately impossible to parse over 500 documents. Therefore they were left out as they almost did not contribute to the final dataset.

2.2 Filtering

Instead of relying on Nutch to do the heavy lifting, I decided to write an external program that would ingest the content, filter it and extract the required content. To do so, I take the plain text dumps from the crawling phase, parse them, and then depending on the website that the content belongs to, extract the information using an HTML parser.

To do so, I implemented a strategy pattern, where each website must implement an interface that I declared. This interface contains methods that are common to all, such as one for getting the title, getting the content, etc. and delegates the actual implementation to another class, hiding the implementation from the user of the system.

Once such filtering is done, I aggregate the movies coming from different sources. To do so, I take the title of each movie, apply stopword removal and tokenization, and look for a match. If a match is found, then the details are merged together and a unique movie is left.

Once such filtering is performed, the data is dumped into a unique JSON, which contains all valid movies crawled by Nutch and filtered by my utility. For each of the movies, the following fields are present:

- Title
- Description
- Image URL
- Genres
- Release Date
- Description
- Link and rating from IMDB
- Link and rating from AllMovie

These fields are then sent to Solr, which takes care of indexing the data, and storing it. This is done using the `post` command provided by Solr (for more details, see the `README`).

2.3 Solr

To index and serve the content, I used Apache Solr. After the filtering phase, the data is sent to Solr which is used as a simple web server which server the content based on the user query. To send the data to Solr, I used the data obtained from the filtering phase which was cleaned and included all the data I needed.

2.4 Frontend

In order to present the results to the user, I created a very simple webpage with a single field on it. Once the user has inserted the query, the data is sent over so Solr which returns the documents related to the search. The results are then shown to the user in two different views, a simple table and a set of cards.

The design of the page was left quite simplistic on purpose. In the end, I decided to use what Bootstrap calls "cards" which are essentially containers.

To implement the frontend, I used Vue.js with axios to make the requests. Other than that, I used the Bootstrap framework to make the website as responsive as possible.

3 Querying

As previously explained, I used Solr to host all of the information. Luckily there is a build in query engine that allows us to obtain the information even if stored in custom fields. Therefore, I used the normal query URL, simply plugging in the information required by the user. To filter the results coming from Solr, I used the `fq` field acting on the title. Being as most people search for the title, this seemed like the correct approach. The resulting URL, for the query "Mission Impossible" is as follows:

```
http://localhost:8983/solr/nutch/select?fq=title:impossible&fq=title:mission&q=*:*
```

Solr will then return a payload as JSON, which can be easily traversed and the required information extracted.

4 Running the tool

In order to repeat the entire process, please check the `README` file.

5 Evaluation

In order to evaluate the tool, I asked a few classmates to look for movies that were both recent and not. No further instructions were given, in order to see how user friendly the tool is. All that was asked was to talk during the evaluation, and to narrate any doubt or questions.

The results are summarized in table 5. All values are in the scale from 1 to 10 and represent how satisfied (10) or unsatisfied (1) they were.

Question	Person 1	Person 2	Person 3
Search for a famous movie	8	9	9
Search for a recent movie	9	9	9
Search for an old movie	6	7	6
Quality of the card display	7	6.5	8
Quality of the table display	6	6	6
Is all the information required displayed	8	7	8
General Score	7	7	8

Table 1: Summary of the evaluation

The positive impressions were as follows:

- Simple to use UI, almost too much
- Results are coherent with the search query

- The card view was nice to look at
- The system was responsive, never hanging too much in the query phase

On the other hand, there were a few remarks:

- The description text is not expandable on the results page
  This is due to a limitation in time, and could be interesting to implement.
- Sometimes, information is missing
  This is unfortunately due to a poor structure on the websites. Sometimes information is missing without reason, and therefore the parser cannot extract it

6 Conclusion

In conclusion, I am quite happy with the tool. It performs as expected, and the results are quite accurate. If I were to do it again, I probably would spend more time in the crawling phase to really optimise this part, both in speed and accuracy.