Design and Implementation of a Vancouver Restaurant Search Engine

Authors: Torres Jin
Silvery Fu[1]

Abstract

Nowadays general purpose search engine is the dominant search engine in the Internet. However, under certain circumstances, they fail to provide QoS-oriented search results to the users. This paper illustrates why and how we develope a search engine for searching restaurants in Vancouver based on tf-idf weighting mechanism and vector model. The final software shows better precision than other major search engines and those build-in search engine of a hub site. We conclude that this kind of limited-purpose search engine is easier to design and implement, lightweight but still provides satisfying service.

Keyword vector model, QoS, search restaurants, Vancouver

[1] 作者：金黎伟 3110103392、傅迪 3110102585

1 Introduction

Vancouver “YumYum” search engine is a search engine provides Vancouver restaurants search services for users. By simply type into one or multiple key words, users can get the information of restaurants in Vancouver in terms of different location, types of foods, price or services.

2 Related Work in Developing Search Engine

This project may concern following algorithms/techniques:

2.1 Inverted Index

A inverted index contains a list of references to documents for each word, additionally contains the positions of each word within a document.The latter form offers more functionality(like phrase searches), but needs more time and space to be created. One of the advantage is that we don't need a lot of space to store useless information(like 0s).

The inverted index also set up a fundament of Td-idf Weighting, and we can calculate the term frequency and document frequency easily.

2.2 Tf-idf Weighting

The document frequency dft is defined as the number of documents that t occurs in. We define the idf weight of term t as follows:

$$\text{idf}_t = \log_{10} \frac{N}{\text{df}_t}$$

where idf is a measure of the informativeness of the term.

2.3 Vector Space Model

By viewing documents and query as vectors composed by terms, it becomes simpler to compute the similarities between them.

2.4 Self-Correcting Mechanism

A simple way is that for every correct words, we computing the distance between a misspelled word and a correct word. And then choose the "correct" word that has the smallest Levenshtein distance to the misspelled word. But practically, the total number of words is large and it is impossible to calculate every distance. So we use the k-gram index to enumerate a set of candidate vocabulary terms and then compute the edit distance from query to each term in this set.

3 The Development Process

The development process of Yum search engine is comprised of five major milestones:

3.1 Implementation of the Crawler and Parser

We use a python module called urllib2 to simulate sending the request and getting the response from the server. We use another python module called Beautiful Soup to find the html tags which contain the useful information we want. What's more, we use depth first search for crawling the page. As for parsing, as the urls of the every restaurants we crawl all include the "/restaurant/", according this and regular expression, we can judge and get all links of restaurants easily.

3.2 Crawling and Parsing Pages

After knowing how to crawl a page, we start to crawl the web page, but still face a lot of challenge. Firstly, We always aren't allowed to visit the website because of too many requests and have to re-crawl the page. Then we come up with a idea that every time we crawl a page, we parse the page and store the date into the file. Then after we rerun the program, the program ignores the pages that have been crawled according to the file, which brings a lot of convenience. Apart from that, the program also doesn't need to fetch the page that has been fetch if some exceptions occur. Secondly, the process of crawling is too slow, so I optimize the program by replace the url with its ip address instead of its domain name. One of the advantage is that the DNS doesn't need to resolve the domain name every time sending the request. Finally, the restaurant name is

stored according to the alphabetical order. Then the word can be searched more fast.

3.3 Implementation of the Indexer and Scoring System

The indexer and scoring system consists of four components: a dictionary generator, which generates the dictionary; a document frequency generator, which generates the number of documents where a specific term appears(used to compute idf); a tf-idf generator which generates the tf-idf weights of a specific term with a specific document;; and lastly a matrix generator, which generates the final term-document postings weights based on the results by the tf-idf generator.

With regard to the dictionary generator, it reads in every term in the collection of documents, if it already exists in the dictionary then just simply ignore it; otherwise add it into the dictionary. Also noteworthy is that, by using python “dictionary” structure, very easily that we group the terms by their first letter, which saves many of other programs a lot of time. (It can be viewed as a B-tree with 26 nodes in its first layer) The results are saved in a .txt file.

The document frequency generator mark the number of documents where the term appears by browsing through every document each term. To faster the process, we make the program skip the document once the current term has been proved there. The results are saved in a .txt file.

Moreover, the tf-idf generator accepts term and the document number as its two parameters, then it computes the tf of this term. After that, it refers to the document frequency’s .txt file to find the term’s document frequency to compute its idf, then use the equation:

$$w_{t,d} = (1 + \log \mathrm{tf}_{t,d}) \cdot \log \frac{N}{\mathrm{df}_t}$$

To get the weight of this term. What needs to stress is, when searching the term’s dFt, we can firstly get the first letter of the term, and jump to the group directly to save some time. The results are saved in a .txt file.

Lastly, the matrix generator will synthesis all of the above .txt files to compute out the postings of each term. For a document that does not

contain the term, it is omitted in the posting lists. Likewise, the results are saved in a .txt file.

3.4 Implementation Self-correction Mechanism

According to the method mentioned above in ch.2.4, first we generate a k-gram index, and them choose the "correct" word that has the smallest Levenshtein distance to the misspelled word. Considering that it's too time-consuming that generating the k-gram index for each query from web page, so we set up a server for correcting the spelling and it returns the result every time it receives a request from web page.

3.5 Accessing a Query

Based on the .txt files, we can evaluate the query by applying the term-by-term algorithm, which goes like this: completely process the postings list of the first query term, and create an accumulator for each document ID we encounter. Then completely process the postings list of the second query term, and so forth.

So much so that we obtained a list of the document ID sorted decreasingly by their scores. The list is returned as the result, which other program will deal with locating the right document according to its ID and show the information to user.

3.6 Implementation the UI Design

Both of the programmers are familiar with html/css developing, and they had experience with Bootstrap. So much so that the UI is mostly constructed based on Flatstrap, a variant of Bootstrap.

Firstly, The structure of our UI was piled up “div” by “div”, after the “divs” were set, css was added.

Secondly, the style of the UI is so-called “flat”, as a trend nowadays. As a result, our UI is clear, brief, and still providing sufficient information to the users. Pictures of the foods are also shown on the search result.

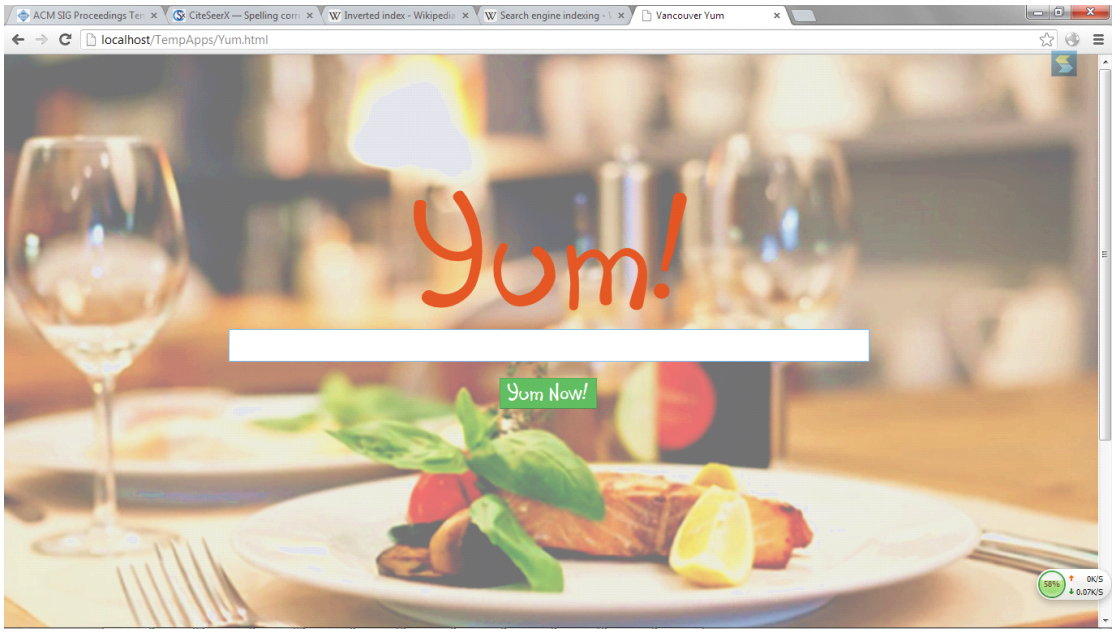
Also, javascript are used to make interactions. For example, when

user types keywords into the search bar, the hints will pop out.

Generally speaking, our UI design is consistent and quite impressive.

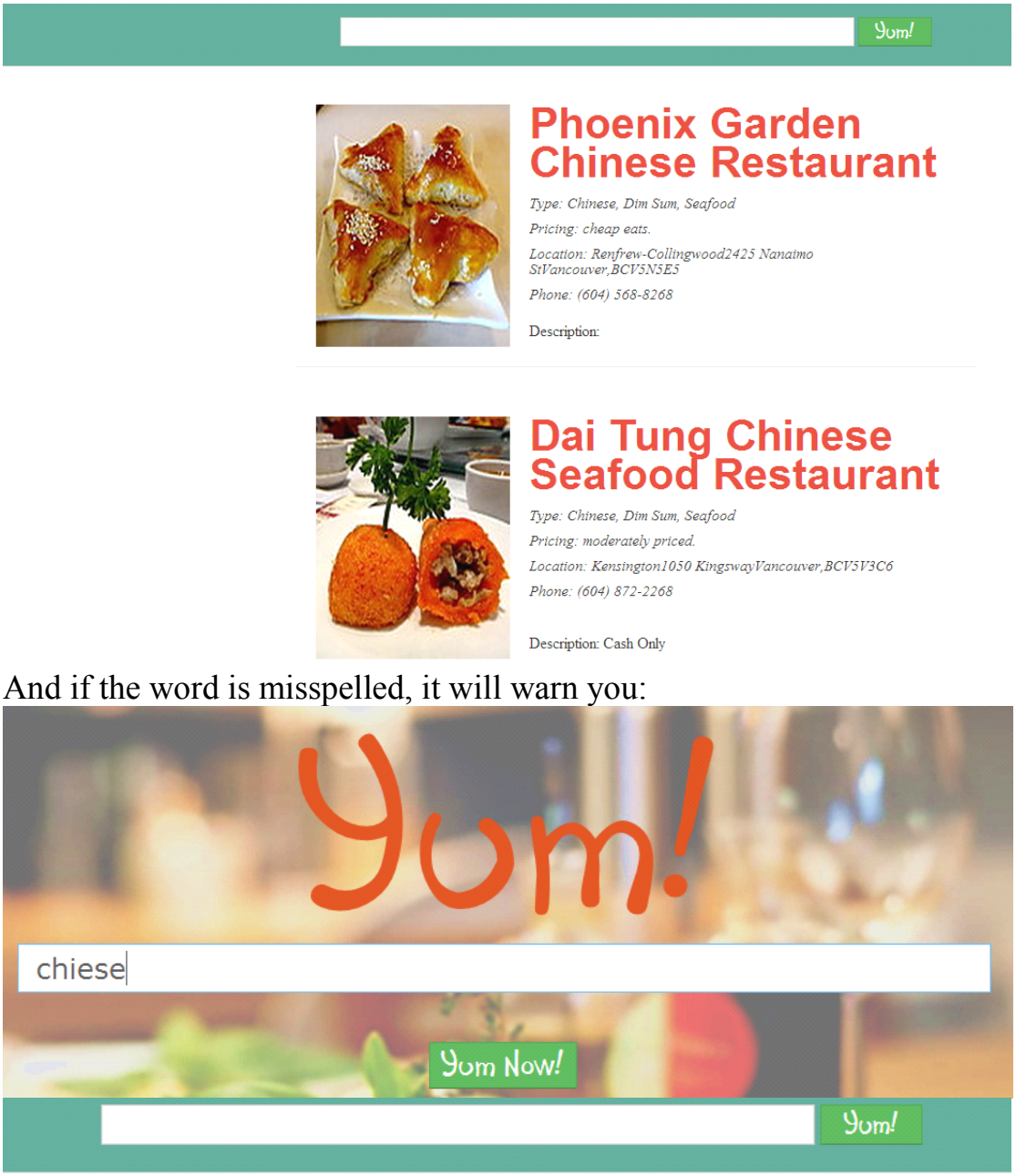
4 Results and Evaluation

User Interface:



Input box with a pop out:

After clicking, it will show the result:
The result is sorted by their rank



And if the word is misspelled, it will warn you:

5 Conclusion

By way of conclusion, we think this kind of limited-purpose search engine is easier to design and implement, lightweight but still provides satisfying service.

Besides, we found out that the keys to develope a search engine are: making compromises, parallel in as many aspects as possible, good algorithm and processing data properly.

6 Acknowledgment

With the help from professors and teaching assistant, especially the latter one, we completed this search engine in time. We have learned a lot. Tons of appreciations.

7 Reference

Anh, Vo Ngoc, and Alistair Moffat, Pruned query evaluation using pre-computed impacts, SIGIR, 2006

Garcia, Steven, Hugh E. Williams, and Adam Cannane, Access-ordered indexes, Australasian Conference on Computer Science, 2004

Amit Singhal, Chris Buckley and Mandar Mitra, Pivoted document length normalization, SIGIR, 1996