Stock NLP Project Report

Feiyi Ouyang

1 Introduction

In the information era, the more information you have the more likely you're gonna win. It can not fit stock market more. One of the most urgent need of stock traders is to retrieve information about the whole picture of the market and how each stock performs. However, time is limited and news come out every minute. It's impossible to check out ginormous source of information manually. Facing such a chaos world, how to collect and extract useful information about stock market has always been a temptating but challenging task.

On the other hand, latest advances in information collecting and analyzing field, like NLP, machine learning offer great opportunities to solve these challenges. Late last year, a group of MicroSoft researchers successfully predicted stock market by analyzing publicly available documents like 10-K reports using natural language processing and machine learning techniques [1]. The success of the project provides thrilling prospect that analyzing the treasure of public resources provides the key to the secret of the stock market with the help of modern analyzing technologies.

This project builds a stock analyzing and summarizing tool by integrating web crawling, NLP parsing, and information retrieval model ranking techniques. The project ansers the following questions:

- What are the most popular topics of a certain day?
- What is the marketś response to a stock (positive / negative)?
- What are the most related articles for a stock or a query?

2 Overview design

We choose an authoritative Chinese finance website CNFOL.com which provides news of public companies as the information source. The website provides a good capture of activities of public companies, as well as people's response to it.

The project is divided into the crawling and querying process. For the crawling process, the crawler goes through the following stages:

- Crawl the articles of all the posts in a given time period. The crawling process is scheduled as a daemon job running at specific time everyday.
- Parse the articles and index the terms. The indexed result is stored into a database.

The querying process goes through the following stages:

- Client sends queries to server.
- Server parses the queries, calculates related statistics, and retrieves documents in response to queries
- Server sends result in json format to the client
- Client displays the data in a user friendly way

3 Investigation over available solutions

3.1 Crawling

There are many open source web crawler libraries, like Scrapy in python and Crawler4j in Java. However, considering that the data we need to crawl is not big for everyday crawling process, we can implement single thread crawler ourselves.

For our crawler, we need a html client which opens a connection, and a HTML parser which selects specific DOM elements. Jsoup provides good interface for these use cases [2].

3.2 Parsing

Unlike English where each word is a individual unit, Chinese words can be grouped by undefined number of characters. Further more, Chinese words are not separated by splitters. These two characteristics make it hard to parse chinese words.

HanLP [3], a NLP library for Chinese language is an ideal tool for our parsing. Not only does it provide robust segmenter and POS tagger, but it also provides customized dictionary, which allows users to mark words with specified taggers. This flexibility enables us to distinguish our own word groups like stock names, words that describe positive or negative stock trend, and stop word list etc.

3.3 Indexing, storing, and querying

The most popular indexing tool for search engines is ElasticSearch, which nicely bundles indexing, ranking, and querying functionalities and provides an easy user interface. It can be seen as a database, but is more powerful in supporting various query forms and full text search, which are not supported by common databases.

However, ElasticSearch is challenging for our use. Although Smart Chinese Analysis plugin helps segmenting Chinese, it doesn't not offer POS tagging and customized dictionary as HanLP does. Furthermore, some people criticize that ElasticSearch is not ready to be the main storage of data as it doesn't implement checksum in the writing process [4]. Although it outperforms databases in terms of indexing and reading, it is not optimized for maintenance and writing when we're re-importing and changing data structure.

On the other hand, the principle behind ElasticSearch is quite simple[5]. The statistics needed to compute scores for each document in response to a query is easy to calculate if data is stored in a carefully-designed way. Here, MongoDB provides a elegant solution for flexible data storage using Json without specifying schema,

along with powerful JAVA driver for it. Furthermore, there's free MongoDB instance hosted on AWS (mLab), which largely improved the statability compared to local storage and reduces the need to maintain. Thus, rather than using ElasticSearch, we build the indexer ourselves, store them to MongoDB, and calculate scores by implementing information retrieval ranking functions. Although it increases the complexity of implementation, it is a good opportunity to learn the underlying information retrieval theories and makes lighter project because of less dependencies.

3.4 User interface and communication with search engine server

A search engine without a good user interface is disappointing. Instead of a command line interface, we provide a graphic interface to send user queries and display ranked documents in a good format. In our project, we used React to make a light-weighted web application.

Web pages can only communicate with external resources over HTTP. Thus, we need another server to expose HTTP endpoint to the web client. Given that most of our search engine is written in Java, we build a Java socket to talk with the client.

3.5 Why we chose Java

There are good libraries in Python and JavaScript for each stage of the project, but Java is the one that can do all the job. For example, Python is good for the crawling stage because of the Scrapy framework, but it is not powerful when it comes to the complex parsing and indexing stage. JavaScript is good for implementing servers on NodeJS platform, but it does not have good support for HanLP library. In addition to library supports, a compiled language like Java is more efficient than a scripting language like Python and JavaScript for the indexing phase, which is computationally intensive.

4 Implementation (refer to supplementary material for the pipeline)

4.1 Customized dictionaries

We creat three customized dictionaries for HanLP parser to tag segmented Chinese words. The stock name dictionary contains all the stock names with our own tag "sn". The positive / negative words dictionary contains all the sentimental words used to describe public response to a certain stock / stock market, and these words are tagged as "pos" and "neg", respectively. The stop words are collected from several common stop word lists online and tagged "sw".

4.2 Java classes [source code: 6]

4.2.1 Crawler and MetaData

The Crawler class establishes a connection with the seed page (www.cnFOL.com), crawls a list of article posts, and constructs a queue of MetaData class, which summarizes the title, url, and time stamp of each post. As the seed page uses dynamic rendering, which pulls more data when users drag to the buttom, we can not extract all links only based on static html. Thus, we monitor the network activity in Chrome debug tools when

the page renders new contents and find the http endpoint to retrieve JSON format of article posts. We launch http requests to that endpoint specifying which page of data we want and get history data.

For each MetaData we called 'parse' method on itself. The parsing pipeline involves the following processes:

1. Extract the article text of the page associated with the url using JSON xml selecter.
2. Pass the article text through NLP segmenter and get a list of words with POS tag (including our customized tags like "sw" for stock names, "pos" for positive words, "sw" for stop words).
3. Filter words with stop words tags and punctuation.
4. Extract contexts for each the filtered term lists. Each context is centered around stock names and has a minimum length of 20 words. If contexts overlap, we designed an algorithm to merge them into one.
5. Index terms and docs with their hashes. Create term entrys, document entrys, and post entrys. The three kinds of entrys has the following JSON format:
   - term:
     `"_id":<term_hash>,"term":<term_word>`
   - document:
     `"_id":<doc_url_hash>,"url":<doc_url_string>,`
     `"title":<doc_title>,"tiem":<time_stamp>,`
     `"pos":<number_of_positive_words>,"neg":<number_of_negative_words>,`
     `"snippet":<snippets_list>,"relatedStocks":<stock_name_string>`
   - post:
     `"term":<term_hash>,"doc":<doc_hash>,`
     `freq":<term_frequency_in_doc>,"norm":<position_of_term>`
6. Write the above collections in batches to mLab hosted on the cloud using MongoDB driver in Java.

4.2.2 Query

The Query class implements a static method 'calDf' to calculate document frequency for each term in each collection (each day), by counting the number of postings containing term hash. These document frequencies (for each term) are stored into seperate collections to reduce the response time in real time query resolving, which simulates the caching mechanism. This function is scheduled to run as a daemon process everyday.

The Query class resolves three kinds of queries in response to a query string:

- Query popular words: find the top 10 terms in the document frequency collection of each day.
- Query stock: find all the documents with the 'relatedStocks' field containing the stock. Calculate the total number of positive and negative words as an index of market response. Also return all the related stocks.
- Common query: replicate step 2 and 3 in parsing pipeline to get list of terms. For each term, calculate a score for all the documents containing this term. The score is calculated as:
  score(q,d) = coord(q,d) * sum of (tf(t in d) * idf(t) * idf(t) * boost * norm(t,d)) (for t in q)
  where:
  - 'term(t,d)' is retrieved from the 'frequency' field of the match posting;

- 'idf(t)' is calculated using df(t) which is retrieved from the matched document frequency entry;
- 'coord(q,d)' is the sum of tf(t,d) of all query terms;
- 'boost' is a measure of freshness (eg, 4 = document of today; 3 = document of yesterday, etc)
- 'norm(t,d)' is a measure of term position weight (eg, 3 = if term in context; 2 = if term in title; 1 = otherwise)

Return the documents with the top scores.

4.2.3 Server

The server class opens a socket on port 9090 and keeps reading the socket input stream. It parses the arguments of input stream, calls corresponding Query functions, and returns the query results in JSON format

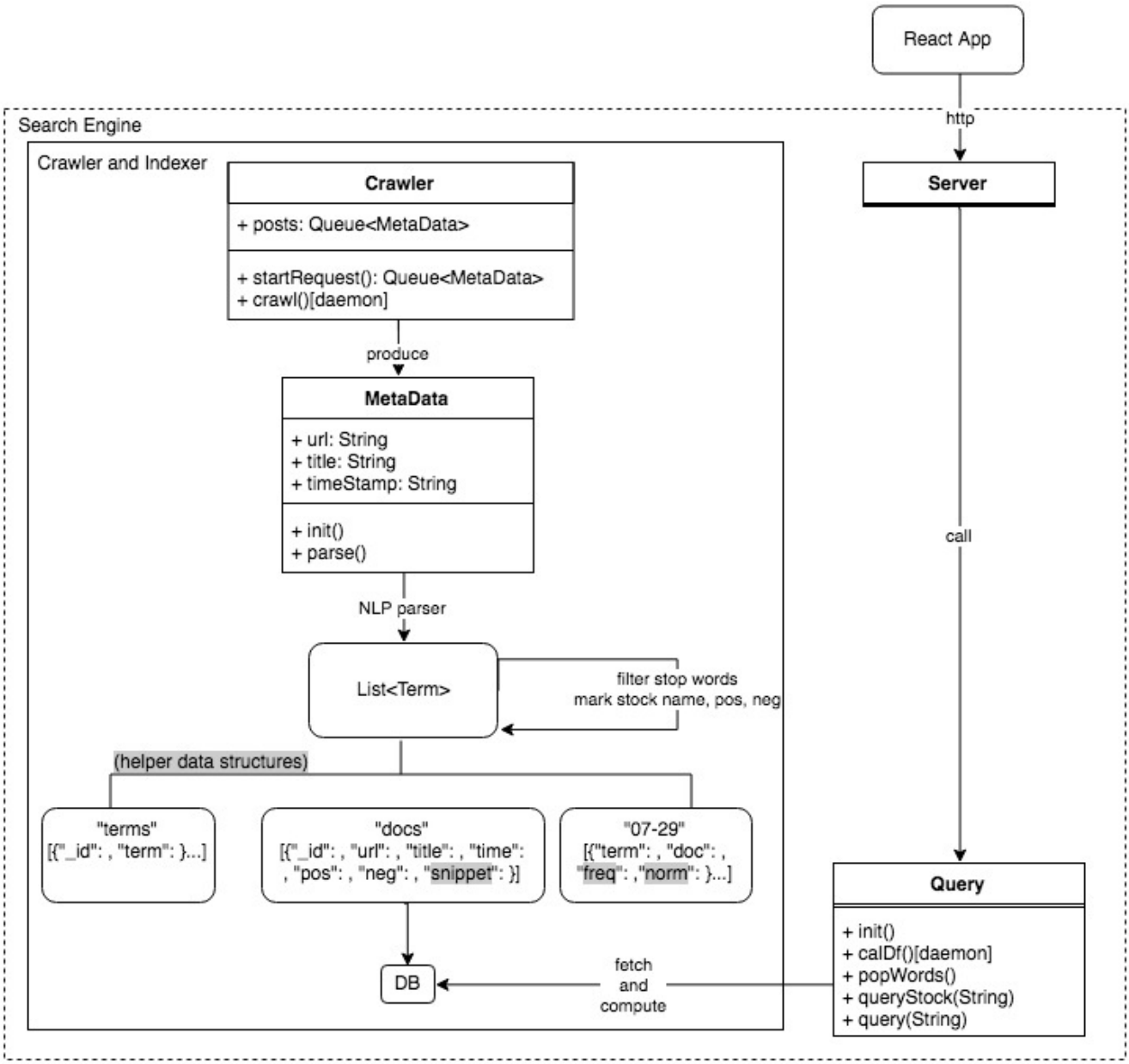
4.3 React client

The React client provides three search boxes for the three queries respectively. Getting responses from the Java server, the React client parses the data and display data in the right format, eg displays title, url, and snippets of a document in the way most search engines do.

5 Result, Evaluation and Improvements

- Popular words query: We found that the top words retrieved contain too much noise. Although we filtered out common words in the stop words process, we did not filter the common words used in the financial world, like companies, stock holders. To actually extract the hot topic words, we need to combine other indexes rather than using pure document frequency. For example, we can compare popular words between days and only select those which have high document frequecies recently.
- Stock query: First, we analyzed if the total number of sensational words (positive / negative words) captures the stock market. We found that there were many trend predicting words we didn't capture. Thus, we need to manually check the parsed articles and add more sensational words to our dictionary. Second, we evaluated if the snippets captured the essence of the article. By looking at the snippets along, it's easy to get the main idea of the article. However, the current snippets are still too long and contain too much noise. We should consider developing another filter over the extracted contexts to remove those information that are not useful. Further more, we should consider build snippets with the unfiltered terms in stead of filtered terms to make it easier to understand for users.
- Common query: We found that the returned documents contained a high percentage of query words, which is consistent with our hypothesis. Also, we found that the documents with high term frequencies for all the words in the query had higher scores than those with extremely high frequency for one of the query words, which is also consistent with our assumption. However, we should normalize documents by it's length to prevent bias towards longer documents.

6 Supplementary Material



7 References

[1] MicroSoft stock prediction project: https://www.microsoft.com/developerblog/2017/12/04/predicting-stock-performance-deep-learning

[2] Jsoup: https://jsoup.org

[3] HanLP: https://github.com/hankcs/HanLP

[4] ElasticSearch compared to MongoDB: https://www.ip-label.co.uk/performance-wire/mongodb-and-elasticsearch

[5] ElasticSearch scoring principle: https://www.ip-label.co.uk/performance-wire/mongodb-and-elasticsearch

[6] The source code is stored here: https://github.com/fayeoyaee/SearchEngine/tree/master/stockNLP