Blog Content Search Engine

Albert Folch
University of Amsterdam
Amsterdam, The Netherlands
albertfolchg@gmail.com

Xavi Moreno
University of Amsterdam
Amsterdam, The Netherlands
xaviml.93@gmail.com

Ben Wilmers
University of Amsterdam
Amsterdam, The Netherlands
benwilmers@me.com

ABSTRACT

Information retrieval systems have revolutionized the way people interact with the internet. They are tools with a wide array of uses which give rise to the need for web search engines that are specialised for specific use cases. In this paper, we describe a new blog post search engine and its architecture, starting with data acquisition and followed by data processing and storage. We combined these modules into a single web application where a user is able to query for blog posts in a standard or advanced way. Finally, we discuss the evaluation of the system with collected validation data and discuss our system performance using offline metrics.

KEYWORDS

blog, search engine, scrapy, elasticsearch, angular

1 INTRODUCTION

Since their emergence in the early 1990s, web search engines have transformed the nature of research and knowledge acquisition through their utility as research tools. The fundamental function of a search engine is to retrieve relevant information to a query from a given knowledge domain, but their applications are wide and far-reaching. From entertainment to news to shopping to academic research papers, search engines facilitate the instantaneous retrieval of many diverse kinds of knowledge. Over 1 trillion Google searches are made every year[1], with this number continuing to rise as more and more people gain access to the internet. This widespread availability of knowledge, irrespective of physical location, which has been made possible by web search engines, has revolutionized the way in which we learn and consume information.

2 PROJECT DESCRIPTION

The goal of this project was to create a blog content search engine. The steps that led to the construction of this engine are described in the following subsections. As well as detailing the system and its architecture in this paper, we documented everything and made it all publicly available on Github [1].

2.1 Data Acquisition

Data gathering is the foundation of any search engine. It should focus on the topic of the engine and gather sufficient varied and meaningful pages to be effective. We crawled seven blog and article websites with 10,000 web pages each, all diverse and with enough content (21GB) — Medium[2], Wired[3], Gizmodo[4], The Verge[5], TechCrunch[6], BuzzFeed[7] and Steemit[8]. Together with the webpages crawled, we attached a file containing the URLs crawled and their connected links.

In order to make this project possible, we used Scrapy, an open source Python framework for extracting relevant data from websites. This framework facilitates the way the data is extracted from the aforementioned blogs. Moreover, it controls concurrency and politeness by adding flags in a settings file.

2.1.1 Politeness. Crawling can be performed in many different ways and using different technologies. Regardless of the tools used, it is important to ensure the process is carried out politely. Politeness here refers to the behaviour of the crawler when gathering data. It is important to remember that when a crawler visits a certain web page, it consumes the resources of its servers, such as CPU power and bandwidth. This process, therefore, should be done in a respectful manner by obeying the robots.txt of each website. Website managers often welcome well-behaved crawlers because they increase exposure of the website, which in turn normally translates to more traffic and/or more revenue (if applicable). On the other hand, an ill-behaved crawler can be banned from crawling that site and from other sites it has not even visited yet. This can be done through automatic detection systems or the website manager himself can ban the crawlers by looking at the servers' logs, where he can find the date, time, the URL requested and the IP address of the bot. The crawler should also identify itself with a User-Agent name and provide a means of contact such as an e-mail.

Scrapy has an option (ROBOTSTXT_OBEY) that we enabled to obey the robots.txt files. After changing this setting, we could see in the logs whether each URL to be requested was forbidden or not.

The next option we added was the name of our bot's user agent and email so the web manager could contact us in case they had any issue with our bot. This setting helps to prevent the web manager from having to look at the logs for our IPs.

Introducing delays is often advised as it reduces consumption of the servers' resources. We decided to set the DOWNLOAD_DELAY to 1 second to avoid hitting the servers too frequently as well as disabling RANDOMIZE_DOWNLOAD_DELAY. This randomization policy reduces the chance of being detected and blocked by bot detection systems but is not transparent.

2.1.2 Distributed crawling. The first setting that we changed to achieve distributed crawling was the CONCURRENT_REQUESTS _PER_DOMAIN. We used the default value of 8 to define the maximum number of requests to any domain. Next, we also left as 16 the CONCURRENT_REQUESTS which defines the maximum concurrent requests that our computer will carry out when crawling

[1] https://github.com/xaviml/bloogle
[2] https://medium.com
[3] https://www.wired.com
[4] https://gizmodo.com
[5] https://www.theverge.com
[6] https://techcrunch.com
[7] https://www.buzzfeed.com/
[8] https://steemit.com

different domains simultaneously. Finally, in the settings configuration, we enabled the setting AUTOTHROTTLE_ENABLED to adjust the delays between requests based on their latency.

Furthermore, Scrapy provides a class called *CrawlerProcess* which allows you to run different spiders simultaneously. These spiders were totally independent, so each of them was tasked with crawling content from one website with no overlap.

2.1.3 Refreshing repository. One feature which we implemented in our system was refreshing of the repository. There are two aspects to consider — updating existing posts and retrieving new ones. Articles and blog posts rarely receive significant updates after the time of creation so we did not deem this necessary for our system. However, every day thousands of posts are created on sites such as Medium or Buzzfeed so we decided to implement this aspect of updating in our solution.

Every time we run the crawl process, we save a metadata file with the date that the crawler was run. This metadata is used the next time the crawler is run, so that only posts made more recently than the time of the last crawling process are downloaded. Thus, refreshing of the repository can be implemented through the creation of an automatic process that checks every 12 hours for new content. This allowed us to solve the issue of adding recent posts to the repository.

2.2 Data Processing and Storage

Carrying out the data acquisition resulted in 70,000 pages in raw HTML. These served as the input for the following step of data processing and storage. In this section, we explain how we processed the pages and created an inverted index with Elasticsearch.

Elasticsearch is an open-source search engine which is built on the Lucene library. It allows for storage, retrieval and management of data and comes with a large range of features and customizability.

2.2.1 Text Preprocessing. The first step in text processing is cleaning the raw HTML. All websites are different and are structured differently, so we used the readability-lxml library in order to extract the title and main text content from the HTML, which was then sent to Elasticsearch.

2.2.2 Extraction of Additional Information. In addition to extracting the title and main text content from each article, we also extracted various metadata, namely the author, date of publication and date of most recent update (if applicable). We achieved this through the use of the BeautifulSoup and readability-lxml libraries.

Secondly, after obtaining structured data from the HTML, we processed the text in order to create the inverted index. Elasticsearch provides a settings API where all of this is configurable. These are the followed steps in text preprocessing:

(1) *Tokenization.* When the data is submitted to Elasticsearch, the tokenization is automatically done by dividing the text into a list of words.
(2) *Possessives removal.* We configured Elasticsearch to remove possessive endings (*'s*) before applying the stop-word removal filter.
(3) *Stop-word removal.* We also configured Elasticsearch to remove stop-words using the pre-defined English stop-word list.
(4) *Stemming.* The final pipeline step is stemming. This can also be easily implemented automatically in Elasticsearch. The stemming type we chose to apply was Porter stemming.[2]

We also removed documents with content of less than 150 characters, resulting in 55,000 indexed posts from the initially crawled set of 70,000.

2.2.3 Distributed Indexing. Distributed indexing can be implemented in Elasticsearch by simply setting the number of shards for each index. A shard is defined as a subset or segment of an index. When an index is created the shards are distributed among the nodes of the cluster. In our case, we set the number of shards to 2.

2.2.4 Structured Indexing. Using the information crawled and preprocessed in the previous steps, we indexed the title, the content (body), the author, the published date and the last modified date for each post, using the URL as the primary key.

2.2.5 Updating Index. Elasticsearch provides a HTTP Post method called _refresh (see Figure 1) that is automatically called every time a new (blog) post is saved. First, Elasticsearch detects whether that post already exists by checking the primary key. If it already exists, it overwrites the previous post; otherwise it creates a new one. An issue that may arise is deleted content. To avoid showing the

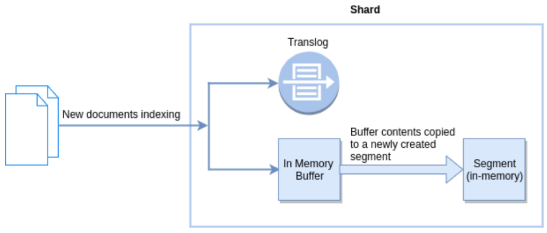

Figure 1
Refreshing index

user posts that no longer exist, we implemented a script to remove unavailable posts. Although blog posts are generally unlikely to be deleted, it is important to ensure this content is removed from the index so that users are not presented with results which no longer exist. To achieve this, the script iterates over all the posts that have been indexed and checks whether each one still exists by sending a GET request for the URL. The system checks whether the server returns an HTTP success code or redirect, which range between 200 and 399. If the server does not return a success code, the post is removed from the index. By executing this script periodically we can ensure that the index remains up-to-date.

2.3 Basic Search

Having built the index, the next step was to implement search functionality. Elasticsearch provides a domain-specific language which can be used to carry out queries over an existing Elasticsearch index. In addition, one of the advantages of Elasticsearch is that it provides an out-of-the-box search implementation.

2.3.1 Query Preprocessing. Elasticsearch allows for queries to be preprocessed in the same way as documents. Thus, tokenization was automatically implemented and possessives removal, stopword removal and English Porter stemming were enabled using the Elasticsearch settings API.

2.3.2 Term-based Search. Elastic search allows for out-of-the-box implementation of BM25. Although it is implemented by default, we still included it in the settings configuration. The formula for BM25 is as follows:

$$\text{score}(D,Q) = \sum_{i=1}^{n} \text{IDF}(q_i) \cdot \frac{f(q_i,D) \cdot (k_1+1)}{f(q_i,D) + k_1 \cdot (1 - b + b \cdot \frac{|D|}{\text{avgdl}})}$$

where k_1 and b are hyperparameters for the formula which we set as $b = 0.75$ and $k_1 = 1.2$. Parameter b considers the length of the documents, with larger documents being penalized as words are more likely to occur in those documents. Parameter k controls the weighting of the term frequencies relative to the average length of the documents.

2.3.3 Spelling Correction. We implemented spelling correction using the Elasticsearch term suggester feature. This is done by sending an additional suggest request (containing the query) as part of the main search request. In this subrequest we specified the query and the raw content of the posts so that Elasticsearch would return the suggestions based on that field without stemming. For each word, Elasticsearch returns an object containing the original word and and a ranked list of alternative words the user might be referring to. The ordering of the suggestions is based on the Levenshtein distance to the original word. When there are multiple suggestions with the lowest Levenshtein distance, the one with the highest document frequency is chosen.

2.4 Additional Search Features and Interface

Finally, we designed a Google-like interface where users are able to search for posts via full text query. We decided to use *Angular 7* to develop this interface together with *elasticsearch-browser.js* so we did not need a backend as a bridge between Angular and Elasticsearch. To provide a more "Google-like" interface, we created two pages - the landing page and the listing page. On the landing page users can type their query and then click on the search button or the "I feel lucky" button . Clicking on the search button will redirect the user to the listing page where the relevant documents (or an error message if no results are found) will be displayed. Clicking on "I feel lucky" will redirect the user to the URL of the most relevant result or to the listing page if no results are found. This page displays the results ordered by relevance with the following information for each: title, URL, author/s, date published and a short extract of the content. The number of results available and the time taken to execute the query is displayed at the top of the page and there is a paginator at the bottom in case there are too many results for a single page.

The first additional feature we included was *Time*. One of the fields saved in our index is the date published. Using this information, we can filter the results for *Any time*, *Past year*, *Past month* and *Past week*(see Fig. 2).

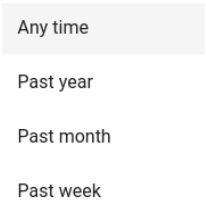

Figure 2
Time feature

Another feature we added was querying terms between quotes (") (see Fig. 3). This allows the user to query exact sentences or sequences of words. Elasticsearch comes with phrase matching capability, so all that is required is to parse the query and put the sentence on Elasticsearch. The terms in the quoted phrase still undergo the previously described preprocessing and are matched against the preprocessed documents in the index.

As well as being able to query full exact sentences, we can add or rule out individual terms by adding a prefix (+,-) to terms (see Fig. 3). Elasticsearch allows users to indicate which terms must or must not appear in the results. In the example query, we indicate that only documents which meet the following conditions should be returned:

- contains the phrase *apple*
- contains the term *fruit*
- does not contain the term *ios*.

These requirements include morphological variations such as plurals as the query terms are preprocessed and compared to preprocessed documents.

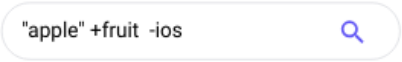

Figure 3
Query example

When the queries are sent to Elasticsearch, we search over all fields — title, content, author and date published. By default, all fields are weighted equally in Elasticsearch, so we had to indicate manually which fields are more important when we created the index. We boosted the title with scale factor 3, content with scale factor 2, and the author and date published with scale factor 1.

Finally, we implemented highlighting of all the query terms which appear in the content and/or title of a relevant document, displaying them in bold (see Fig. 4). The highlighted terms of the content are displayed in context and joined with ellipses.

When the user makes a query, Elasticsearch returns the closest spelling correction for the query (if a term is not found in the index) as described in the previous section. This correction will be shown to the user with the misspelt term highlighted. However, the displayed results are those for the misspelt query, so the user has the freedom to look at the content they queried. A link is displayed giving the option to make a new query with the corrected spelling.

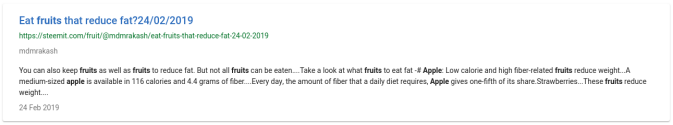

Figure 4
Result example

2.5 Offline Evaluation

In this section, we describe how the system evaluation was done.

2.5.1 Interface for Gathering Data. First, we added two new buttons to each retrieved document on the listing page (see Fig. 5) for the assessors to indicate the relevance of the documents. Once each assessor had finished their judgments, a structured JSON file (see Fig. 6) was downloaded containing a list of elements that have the query and all the documents assessed for that query. Each document consists of the URL and rank of that document on that query and its relevance. Note that there is one JSON file per assessor.

Figure 5
Feedback buttons

```
{
  "queries": [
    {
      "data": [
        {
          "relevant": true,
          "url": "www.example.com",
          "rank": 1
        },
        {
          ...
        }
      ],
      "query": "query example"
    },
      {
        ...
      }
  ]
}
```

Figure 6: JSON Format example for evaluation

2.5.2 Instructions for Assessors. Before assessing our system, we prepared a script with instructions for the assessors. This consisted of:

- A description of the system and its content.
- A brief explanation of the system features.
- A list of queries to be assessed.
- Instructions on How to select the relevance of a document.

2.5.3 Inter-assessor Agreement. In order to validate the judgements, we computed the Cohen's kappa coefficient, which is a statistical measure of inter-assessor agreement. This takes into account the number of agreed judgments between two assessors and applies the following formula:

$$\kappa \equiv \frac{p_o - p_e}{1 - p_e} = 1 - \frac{1 - p_o}{1 - p_e}$$

where

$$p_o = \frac{\#\text{ agreed}}{\#\text{ total}}$$
$$p_e = p_{rel} + p_{nonrel}$$
$$p_{rel} = \frac{\#\text{ a1 relevant}}{\#\text{ total}} \cdot \frac{\#\text{ a2 relevant}}{\#\text{ total}}$$
$$p_{nonrel} = \frac{\#\text{ a1 nonrelevant}}{\#\text{ total}} \cdot \frac{\#\text{ a2 nonrelevant}}{\#\text{ total}}$$

In order to compute this coefficient between more than 2 assessors, we took the average of all the pairwise computed kappa coefficients.

2.5.4 Offline Metrics. All evaluated data is processed automatically so the metrics are computed immediately after evaluation. These are the metrics we computed per query:

- Precision at rank k. This metric considers the rank by computing the precision of the first k documents. We compute P@2 since the viewport of the results shows 2 entire retrieved documents before scrolling (see Fig. 11). We also compute P@5 and P@10.
- Reciprocal rank. This is another ranked metric, it considers the rank of the first relevant document, with a higher score being preferable. Thus, the score is calculated by:

$$RR = \frac{1}{\text{rank of first element item}}$$

- Average precision. This metric is the average of precision at rank k, where element k is relevant. The formula is as follows:

$$AP = \frac{\sum_{d \in rel} P@k_d}{\#(\text{relevant items})}$$

- Normalized DCG. Finally, we computed the normalized discounted cumulative gain. This is the only metric based on user behaviour that we considered. The formula is as follows:

$$DCG = \sum_{k=1}^{N} \frac{2^{R_k} - 1}{\log(k+1)}$$

$$NDCG = \frac{DCG}{DCG_{ideal}}$$

It is not possible for us to compute any metric that considers recall since it is unfeasible to obtain judgments for the relevance of all documents for most queries. Another metric we did not consider was the expected reciprocal rank. This is a user-oriented evaluation that assumes users have a given probability of continuing to look for relevant results when a non-relevant document is returned. The formula is defined as follows:

$$ERR = \sum_{k=1}^{N} \frac{1}{k} \cdot \theta^{k-1} \cdot R_k \cdot \prod_{i=1}^{k-1} (1 - R_i)$$

where

$R_k \in \{0, 1\}$ since we classify documents as relevant and not relevant
θ is the probability of continuing to look

However, as we consider binary relevancy, the formula is reduced to:

$$ERR = \frac{1}{k} \cdot \theta^{k-1}$$

where k is the rank of the first relevant item. If we assume users always continue to look until they find a relevant result, we can compute ERR with $\theta = 1$ (it is normally considered to be close to 1):

$$ERR(\theta = 1) = RR = \frac{1}{k}$$

With this assumption the expected reciprocal rank will be equivalent to the reciprocal rank, a metric we had already computed.

2.5.5 Helping Two Other Teams. We evaluated two other systems developed by group 9 and 20. The former team developed an unbiased news search engine, diving it on left, neutral and right political content. We assessed 10 documents per query with a total of 20 queries on a web form that they provided physically. The second team developed a car search engine. They sent us a document by email with instructions for assessors to assess their system. As with the other team, we assessed 10 documents per query for a total of 20 queries on their actual website.

3 SYSTEM ARCHITECTURE/DESIGN

We based our architecture (see Fig. 7) on a simple model consisting of three chained modules. The modules are as follows:

- *bloogle-bot.* This module is responsible for data acquisition. It crawls posts from the aforementioned websites and saves the data on disk as plain HTML.
- *bloogle-indexer.* This module is responsible for data processing and storage. It creates an inverted index from the documents that we crawled with *bloogle-bot.*
- *bloogle-search.* This module provides a Google-like interface that acts as a bridge between the user and Elasticsearch. It uses the indexed results from the *bloogle-indexer.*

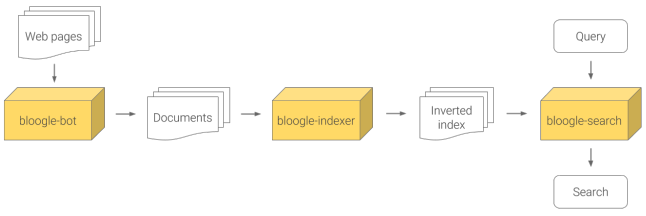

Figure 7
System Architecture

This modular architecture facilitated the development of the project since every subsection is independent. This meant that each module could be coded separately and the workload was easily spread out amongst the team members. Moreover, we chose three technologies (Scrapy, Elasticsearch and Angular) that facilitated the work on each of the modules respectively.

4 PROBLEMS ENCOUNTERED

In this section, we describe the problems we encountered during this project and our approach to dealing with them.

4.1 Data Acquisition

One of the major problems we encountered while crawling websites was implementing refreshing of the repository. There are three major tasks when refreshing a repository — Crawling new content, updating existing content, and removing non-existent content. The majority of current web pages, including the ones we crawled, are dynamic, so the *dateModified* attribute does not appear on the HTTP head response. Therefore, given the limitation of not knowing the most recent date of modification, we had to make the assumption that the crawled content will never change as blog posts are rarely altered after their initial publication. Thus, the only factors taken into account are crawling new content and removing deleted posts.

4.2 Additional Search Features

Another challenge we faced was detecting and obtaining query text within inverted commas and after - and + signs. To solve this we used the following regular expressions.

- Inverted commas: `"(.*?)"`
- Minus sign: `-(\w*)`
- Plus sign: `\+(\w*)`

4.3 Search interface

One problem we encountered with the interface was when the query terms were highlighted on the title only. In this particular case, Elasticsearch we did not have the highlights of the content and therefore we had to use the entire content of the post and add ellipses automatically using CSS when the content text overflowed the post space. This way of adding the ellipses is more elegant and does not break words.

5 REFLECTION

In this section, we explain the insights about the values we obtained from the evaluations of our system.

We picked 4 assessors (3 from group 3 and 1 from group 9). Each assessor assessed 10 documents per query for a total of 20 queries (200 documents judged). We computed the Cohen's kappa coefficient and obtained a value for the average pairwise computed coefficient of 0.496, which is relatively low. In order to be an acceptable judgment, the value must be higher than 0.68. Fig. 8 shows that the high values of the coefficient come from agreements between the members of group 3. However, the assessor from group 9 does not agree with them, lowering the average. If we drop the member from group 9 (considering him to be a poor judge) the average raises up to 0.588, which is significantly higher than the previous score.

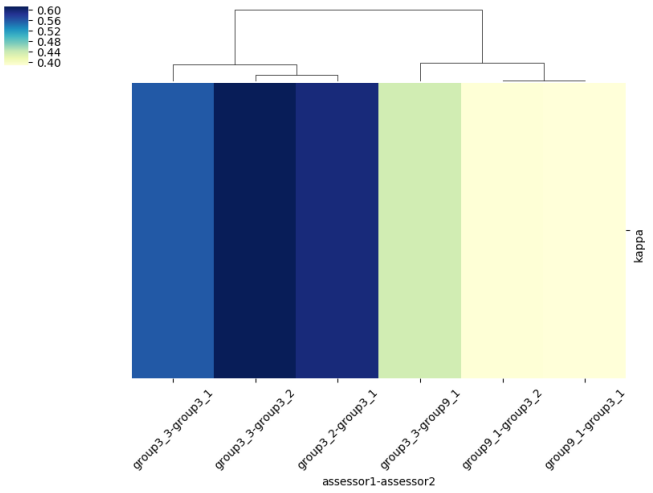

Figure 8
Cohen's kappa coefficient

Once all judgments were completed, we took the mean judgment of the 3 assessors for each documeat nnd rounded them to the closest binary relevance score: 0 (nonrelevant) or 1 (relevant). Then, we computed the aforementioned offline metrics (see Fig. 9). Since we computed 6 metrics for 20 different queries (to compare them individually) this resulted in a total of 120 computed metrics. First of all, we can see that while the system performs well for some of the queries, there are several for which the performance is poor. We observed that the low value queries are either too specific (e.g. "apple watch" vs. "samsung galaxy watch") or queries for which there exist few relevant documents within the scope of the index (e.g. "apple" +fruit -ios) as we mostly crawled technology blog posts.

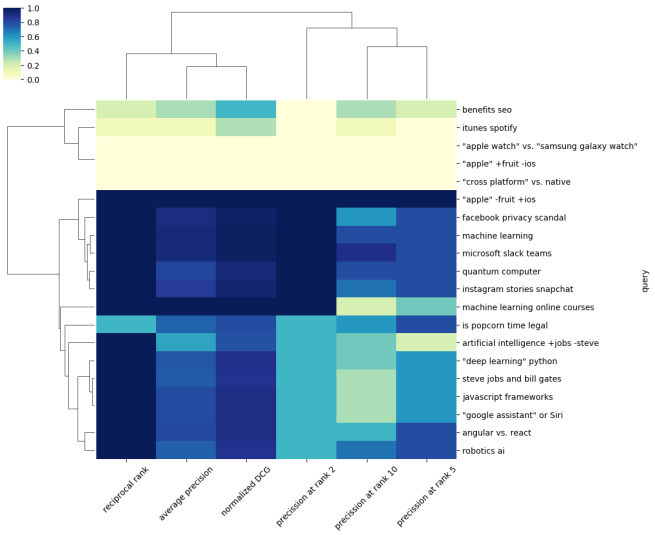

Figure 9
Offline metrics evaluations

Finally, we observe that most of the queries present a relevant document on the first rank (reciprocal rank). In addition to this, a valuable metric to observe is precision at rank 2 since the viewport shows two results before scrolling down to see more. We observe that this metric is quite high for all queries, meaning that the visible content immediately after querying is relevant. However, if we take a closer look at p@10, we notice that it is significantly lower, meaning that while the first results are relevant, the rest are mostly irrelevant. The system is quite inconsistent when scrolling down and looking for further results.

Bloogle is a good system for general purpose queries such as 'machine learning'. However, one limitation of the system is that the relevance of documents is calculated using BM25, which is a term-based method. While BM25 is considered a state of the art retrieval function, it does not take into account semantic features of queries and documents. Implementing a semantic search algorithm would help with handling issues such as synonymy and polysemy.

Another issue which our system faces is the limited scope of the content crawled. Given the vast size of the blogosphere and the limited scope and timeframe of the project, the indiscriminate crawling strategy we applied within each website meant there were many topics for which a limited number of posts were scraped and indexed. However, this problem could be easily solved with more time or by increasing the number of machines used and distributing the crawling process further.

6 SUMMARY OF WORK DONE

Fig. 7 shows an overview of the architecture we built. We started by crawling many blog content web pages in a distributed and polite manner and also enabled refreshing of the repository through the gathering of new posts.

We then pre-processed the crawled content for indexing. This included stemming and stop-word removal. We also allowed distributed indexing and updating of the index for updated and deleted blogs. Additionally, we extracted time as meta information and indexed the different parts of the blog posts separately.

The next step was implementing the basic search. Here, we created the search engine that used the same text processing pipelines so that queries could return and match real content accordingly. We also offered spelling correction so that users can see if they misspell any words.

In the additional search features and interface section, we created a Google-like UI based on Angular so that the user could feel comfortable with our engine. The UI also gives the possibility of full text search, term inclusion and exclusion as well as filtering by date.

Finally, we created an evaluation system. We added a new feature to the interface whereby assessors could judge the relevance results in an easy manner. After evaluation data was collected, we computed the inter-assessor agreement to make sure all the gathered data was consistent and calculated offline metrics to evaluate the performance of our system.

7 FUTURE WORK

Some ideas that could be added to our search engine include search features such as semantic-based search and entities. The former

would enrich the results with higher accuracy whereas the latter could enable the extraction of information about a searched author and display it using cards on the side. Another possible option could be to add language detection when indexing so results can be filtered by language.

8 DISTRIBUTION OF WORK

As mentioned previously, the modularity of our system as well as the consistent technical knowledge of all team members allowed an even distribution of work throughout the project.

REFERENCES

[1] [n. d.]. Google Zeitgeist 2012. ([n. d.]). https://archive.google.com/zeitgeist/2012/

[2] Tartarus. [n. d.]. The porter stemming algorithm. ([n. d.]). http://snowball.tartarus.org/algorithms/porter/stemmer.html

A SCREENSHOTS

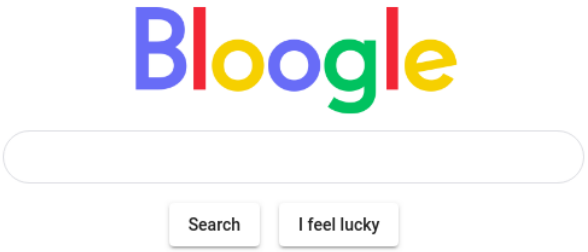

Figure 10: Bloogle's main page

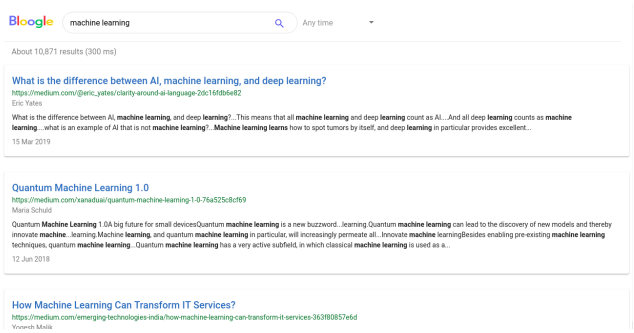

Figure 11: Viewport on 13" screen

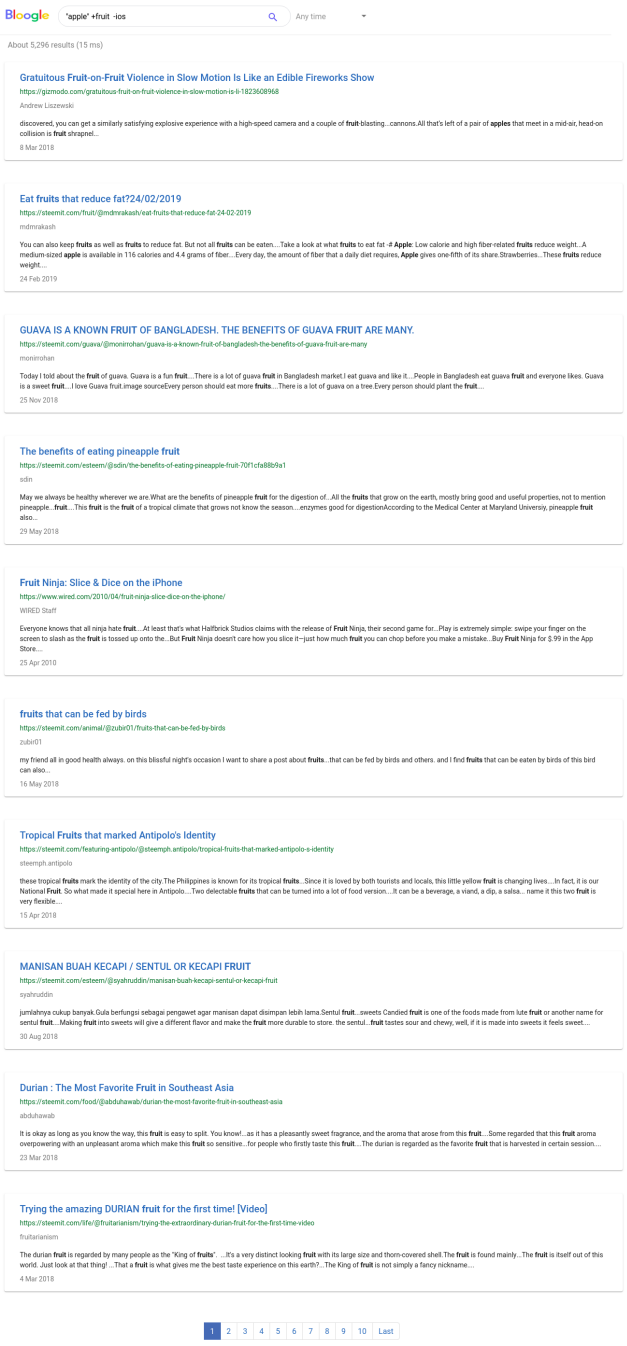

Figure 12: Query: "apple" +fruit -ios

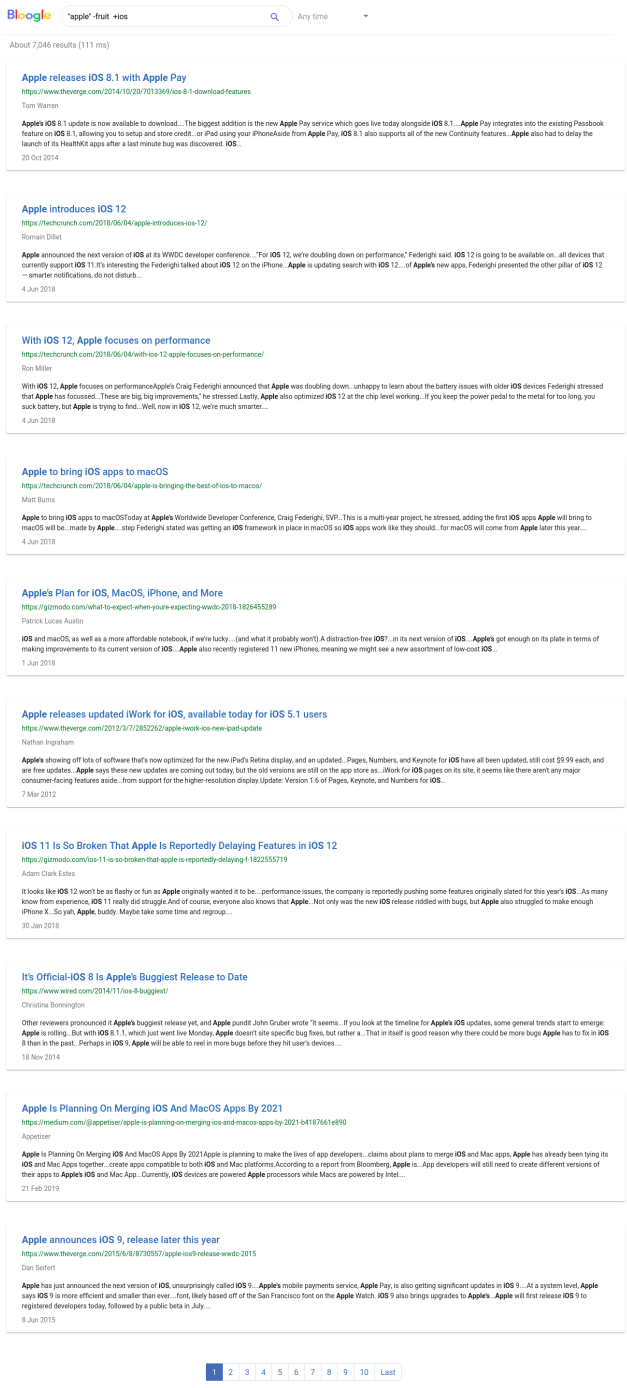

Figure 13: Query: "apple" -fruit +ios