A Local and Intelligent Web Information Retrieval system

by

Bijin Benny

Supervisor:
Mr. Leonard Rogers

Master of Science in Internetworking

Department of Electrical and Computer Engineering
University of Alberta

Abstract

The advent of the internet and the World Wide Web has made vast amounts of information accessible at our fingertips. It is said that an average person living in the modern society is exposed to as much information in a day as a person who lived 100 years ago would have seen in a whole year. But how do we search and find what we want from this giant heap of information dumps? Search engines have solved this problem for us. Numerous search engines such as Yahoo, AltaVista, MSN and Google were created over the years to access the billions of information on the internet effectively. Some failed to keep up, while the others have grown exponentially in terms of its collection of data and effectiveness in providing the best answers to our questions.

Despite its usefulness in filtering through the internet, search engines pose a threat to its users, who are mostly unaware of it. Popular search engines receive over a billion search queries every single day. These search queries can reveal a lot of private information about an individual or a group of individuals at scale. Hence, modern search engines contain sufficient data obtained through invading the user's privacy that could, in the worst case, be used for manipulative purposes.

This project aims to understand the working of modern search engines and to propose a local alternative with the help of various machine learning and natural language processing techniques. Through the process, different machine learning approaches will be studied, implemented and analyzed to provide the best solution to the information retrieval problem.

Table of Contents

List of Tables

List of Figures

Chapter 1

Search Engine - Architecture and Concepts

1.1 Introduction

Search engines are the most common examples of information retrieval systems that we interact with daily and they aid us with information on any topic under the sun and beyond. But, this seemingly simple system is comprised of a number of complex components and techniques that make it possible to gather and store colossal amounts of information and provide the most relevant answers to the queries posted against it in a matter of milliseconds time. In this chapter, we look at the necessary components that make up a typical search engine and also a few fundamental techniques that are used in all search engines.

1.2 The Search Engine Architecture - Overview

The components that comprise a typical search engine can be broadly classified into three categories based on their individual functions - Crawling and data gathering, Content repository or database and a Query engine and results page.

- **Crawling and data gathering** - A crawler discovers and adds the content of the Web to the search engine's data repository. Most crawlers find information by beginning at one page and then follow the outgoing URL links on that page.

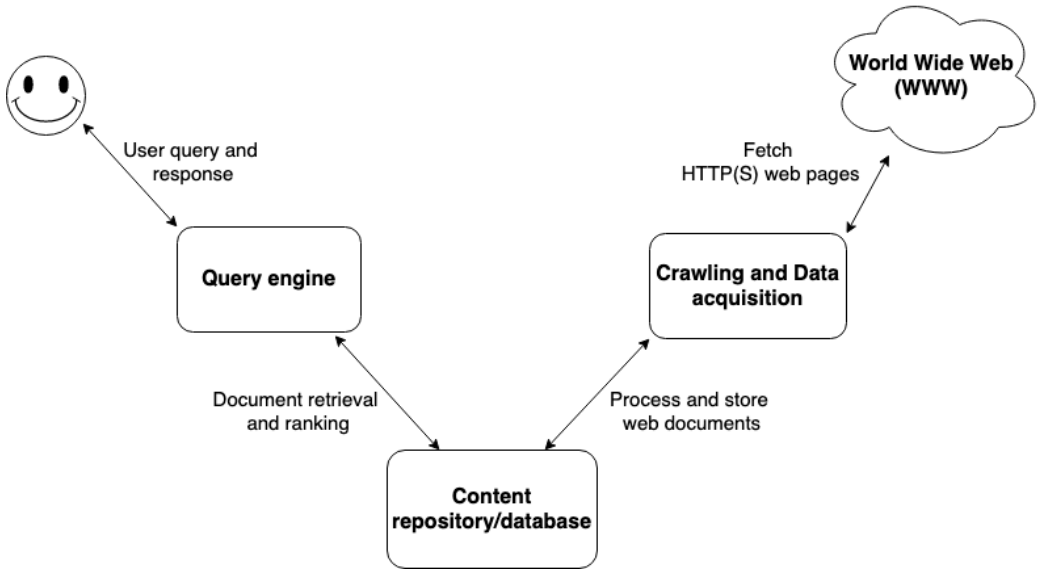

Figure 1.1: Basic search engine architecture.

Therefore, if a page is not linked to, from another page, this page may never be found by a search engine. In some cases, a search engine may acquire content through private data feeds from other companies. All of the information that a web crawler retrieves is stored in a data repository, which provides a foundation to build an online index search-able by users[1].

- **Content repository or database** - The information scraped from a myriad of web pages is pre-processed and stored in a database in a form that makes it easy to search through and retrieve the contents when a query is posted. Indexing is a technique used to efficiently store and manage documents in order to facilitate fast searching and retrieval. A more profound understanding of the pre-processing and indexing techniques is described in the following sections.

- **Query engine and results page** - The query engine module, usually a web service, takes user inputs, processes them and returns the most relevant results on the results page that match the query[1]. Multiple approaches have been employed to find the most matching documents for a particular query and this document aims to understand and compare the performance of a few of them.

The user's interaction with the results may be logged by the search engine to record the behaviour and utilize it to improve the quality of results[1].

1.3 Crawling

Web crawling is the process of collecting and extracting information from the web with the help of programs called crawlers or spiders. The primary function of a web crawler is simple - select a URL from a set of candidate URLs known as the URL frontier, download the associated web pages, extract the hyperlinks contained in the web page, and add those URLs that have not been encountered before to the candidate URLs list[2]. The objective here is to scan through as many web documents as possible quickly and efficiently and return the scraped contents for storage[3].

1.3.1 Crawler Characteristics

The following key characteristics are important in any web crawler:

- **Robustness:** Some domains may have certain 'traps' in their web pages that lead to crawlers or spiders getting stuck trying to fetch an infinite number of web pages in the domain[3]. One typical example of such a trap is infinitely looping over a calendar web page. This causes the crawler to get stuck loading useless information endlessly. Also, there could arise situations where the web page may take a long time to respond or even fail to load. A well-written crawler must be able to circumvent these problems and gather useful information while discarding the others[1].

- **Politeness:** The main objective of a crawler is to access and fetch as much information as possible from all kinds of web pages on the internet. However, in doing so, the crawler may try to use up a significant chunk of the computing and bandwidth resources of the webserver, thereby preventing the human users from accessing the website[1]. There are policies in place that restrict the crawlers

from making multiple requests to the same server at the same time or even within short time periods. Hence, the crawlers will have to be designed in such a way that it attains its peak throughput while abiding by the 'politeness' policies.

1.3.2 Types of Crawler architectures

- **Parallel Crawler**. Crawling through millions of web pages on the internet with a single instance of the crawler is a slow and inefficient task. Hence, the parallel crawler architecture was introduced that runs multiple instances of the web crawler simultaneously, scraping and downloading information from the web in parallel. Each of these crawler instances, known as a C-proc or Crawler Process, performs identical tasks - It retrieves pages from the world wide web, stores those pages locally, filters the URLs of the pages retrieved, and passes through these URLs[4]. Based on the location of these different parallel crawlers, they can be classified into two categories - Intra-site parallel crawlers (processes running on the same local network) or Intersite/Distributed parallel crawlers (processing running on different networks communicating with each other through the internet)[5]. Figure 1.2 shows the architecture of a parallel web crawler.

- **Incremental Crawler**. Incremental crawlers are designed to visit and access frequently updated web pages. Incremental crawlers update the stored content of websites by visiting them often and storing the updated version of those pages[4]. The objective of designing an incremental web crawler is to main fresh and updated versions of the web pages in its local collection. Incremental crawlers are composed of three sub-modules as shown in Figure 1.3 - the Ranking Module, the Update Module, and the Crawl Module.

  The crawler makes use of the data structure known as priority queues for choos-

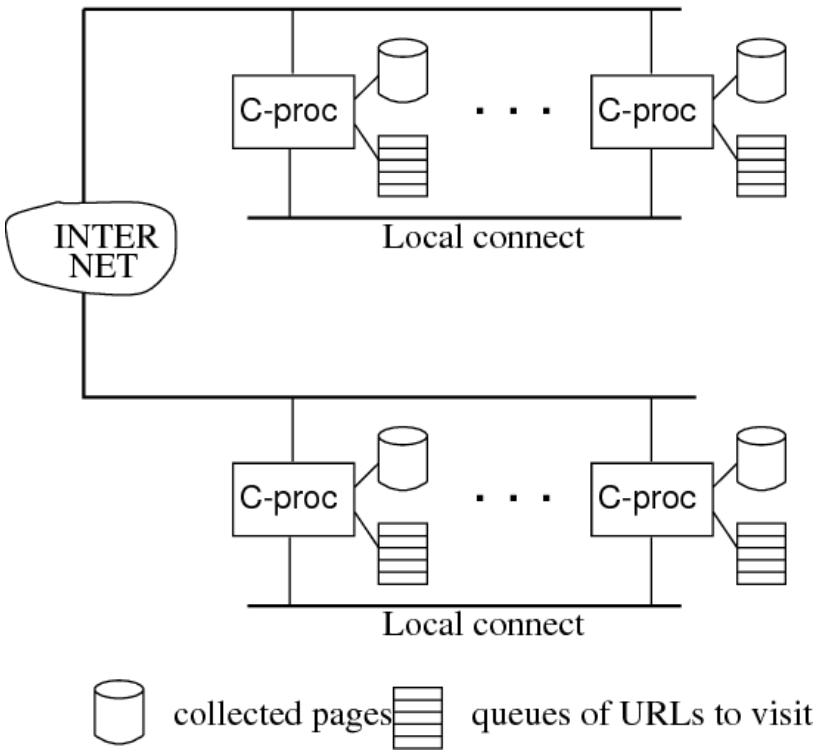

Figure 1.2: Parallel Web Crawler architecture [6]

ing the URLs that need to be revisited and updated. The AllURLs queue contains all the URLs discovered by the crawler and the CollURLs queue contains the URLS that will or have already be visited. It is the Ranking modules job to rank and move URLs from the AllURLs queue to the CollURLs queue. If it finds a URL that is not in CollURLs queue, it replaces the URL with the least priority in the CollURLS queue with the new URL and assigns the highest priority so that it will be crawled immediately.

Update Module's primary work is to check whether or not the web page content is updated. Update Module calculates the estimated frequency of change page using a clustering algorithm[7] as the ruling function. The URL is placed in the queue based on these calculations. The URL which is present closer to the head of the queue will be visited frequently and hence the incremental crawlers provide updated pages to the users[4].

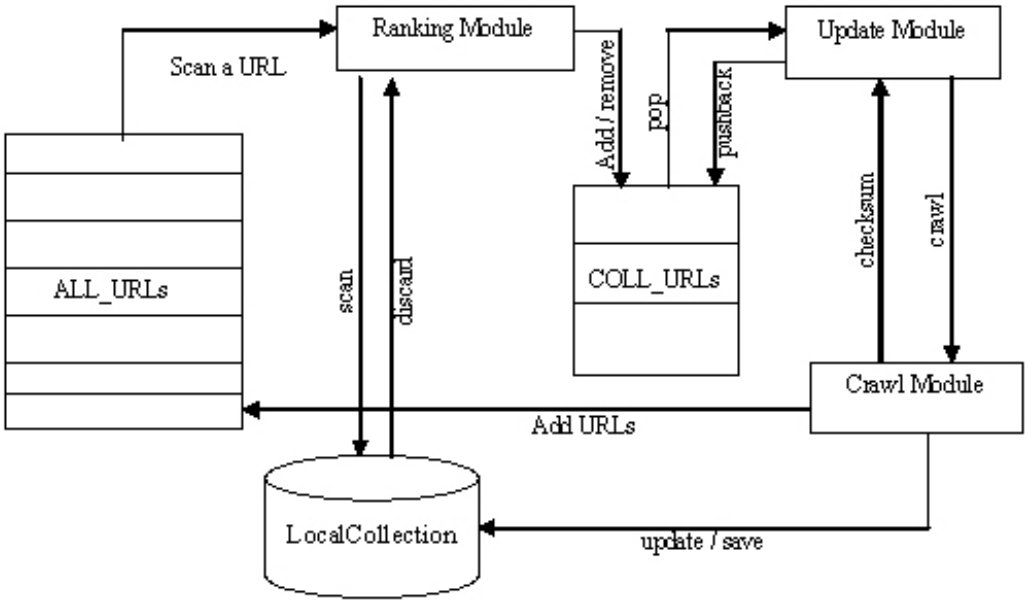

Figure 1.3: Incremental Web Crawler architecture [8]

- **Focused/Topic-driven Crawler**. Focused or topic-driven crawlers are designed to be used for specialized searches that provide deeper results from a particular domain. The crawler maintains the relevance of the web pages by introducing a validator module as shown in Figure 1.4. The validator processes the downloaded web pages and calculates the relevance to the topic using different machine learning algorithms like support vector machines and bayesian networks that are out of scope for this project[4].

- **Hidden Crawler**. The world web is divided into "surface web" and "hidden web" parts. Web content that can easily be accessed by general-purpose crawlers is referred to as the surface web. Huge web content is hidden behind search forms that are not accessible by any standard search engine, these hidden pages are referred to as the hidden web[4]. Universities and government institutions are examples of such private access content and maintain a private database that cannot be publicly accessed over the internet, i.e., restricts access members or subscribers only. Hidden web crawlers are looking for these search forms on each web page visited and this form is automatically filled by a Label Value

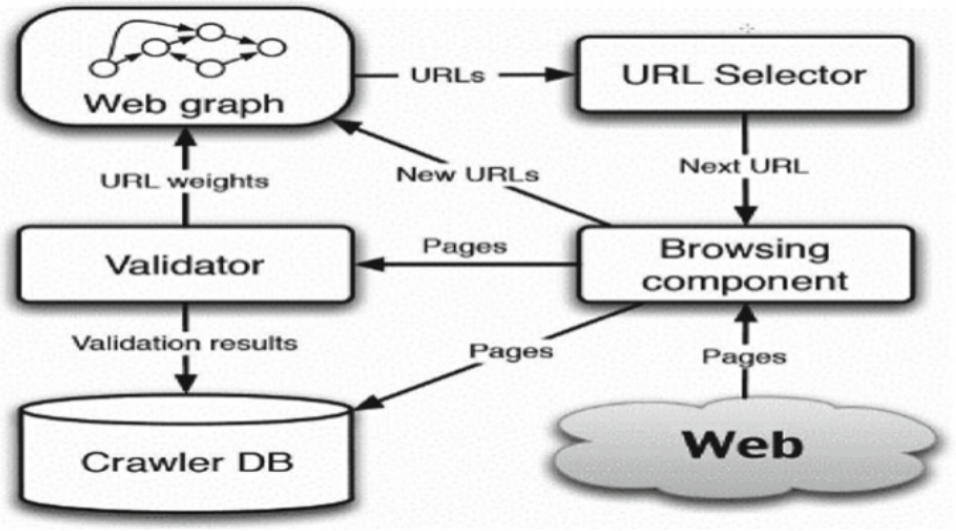

Figure 1.4: Focused Web Crawler architecture [4]

Set(LVS) manager in the hidden web crawlers and submitted to the web[4]. The form response is then parsed and the contents are stored in the database.

1.4 Content repository or database

The information scraped from the vast number of web pages is largely unstructured. It contains words that have the same meaning in different forms, punctuation and even 'stop' words that do not add any particular meaning at all. If we were to search through such unstructured data directly, it would use up a lot of time and resources rendering the process highly inefficient. Hence we need to parse through this raw information, process it into a defined structure in order to ease the searching operation. The next few sections will describe some of the techniques used in the processing of the raw data.

1.4.1 Tokenization

For a given input sequence, tokenization is the process of breaking down the sequence into its constituent words or tokens while getting rid of certain characters like punctuation[3]. These tokens are useful for indexing which will be described in the later

section. The following is an example of tokenizing an input sequence:

Input: The quick brown fox jumps right over the lazy dog.
Output: [The] [quick] [brown] [fox] [jumps] [over] [the] [lazy] [dog]

Tokenization is fairly straightforward with the English language but it is not the case with languages such as Arabic or Chinese where words are not clearly separated by white spaces. In such cases, we use powerful natural language models to identify words in a sentence[1].

1.4.2 Stop words

Stop words are those words in a sentence that do not add significant meaning towards the sentence. Words like 'the','a','for' etc are a few stop words that do not contribute any meaning to the topic in the document. These words should be removed before moving on to indexing as indexing is expensive on storage and the goal is to identify only the significant terms for indexing. However, it is incorrect to always remove these 'stop' words because some of these words do provide some context to the sentence in certain situations. For example, WHO (World Health Organization) even though it looks like a stop word, adds meaning and explicitly removing it from the documents may affect the query results.

1.4.3 Stemming and Lemmatization

Words that impart the same meaning appear in different forms in the English language due to the rules of grammar. The word 'catch', 'catches' and 'catching' are all different forms of the verb 'catch'. But it is not efficient to store all these words while indexing as they provide the same context to a sentence even though they appear as different words. Hence, we use what is called stemming to retrieve the original form of a word that imparts the same meaning as the word in consideration.

One of the most common stemming algorithms is Porter's stemmer[9] algorithm. Porter's algorithm consists of five phases of word reductions, applied sequentially. Within each phase, there are various conventions to select rules, such as selecting the rule from each rule group that applies to the longest suffix[3]. The following is an example of Porter's stemming algorithms:

Input sequence: When glass breaks the cracks move faster than 3 000 miles per hour Stemmed sequence: When glass break the crack move faster than 3 000 mile per hour

While stemming is just a crude processing on the sentences by removing the trailing characters of words, lemmatization is a more advanced technique. The concept of lemmatization is detecting semantically equivalent surface words written in different syntactic forms and relates them to their canonical base representation or lemma[10]. Lemmatization can relate words that do not look the same syntactically, but have the same semantic context. For example, the words 'see' and 'saw' look different but can have the same meaning. Stemming would just produce 's' in the case of the word 'saw' while lemmatization would return 'see' or 'saw' depending on the condition whether the word was a noun or a verb[3].

1.4.4 Indexing

Once we have the document tokenized, stemmed and lemmatized, we can now index them in the database. Indexing is simply a way of storing and organizing information that enables fast responses while searching. There are different types of indexing techniques, but we focus on the reverse index which is the most common type of indexing in information retrieval.

A reverse index is formed by pairing each of the terms or words with the list of documents where the word occurs. For example, if a document D_1 contains terms {a,b,a} and document D_2 contains terms {b,c,e} then the reverse index on the collection of these documents would look like[1]:

$a \longrightarrow \{D_1\}, b \longrightarrow \{D_1, D_2\}, c \longrightarrow \{D_2\}, e \longrightarrow \{D_2\}$

Additionally, the index could store more information about the term in the document such as its frequency and position. For the same example above, such an index would be as follows[1]:

$a \longrightarrow \{D_1, 2, (1, 3)\}, b \longrightarrow \{D_1, 1, (2), D_2, 1, (1)\}, c \longrightarrow \{D_2, 1, (2)\}, e \longrightarrow \{D_2, 1, 3\}$

In the above representation, the term **'a'** occurs twice in document D_1 at positions 1 and 3. The benefit of using a reverse index can be seen while searching for documents that contain particular terms as it occurs in a given search query. Each document can be scored based on the number and position of occurrences of the search terms in those documents.

1.5 Query engine and Ranking

The last and final stage of a search engine, the stage that interacts with the user to provide responses to the queries is the query engine and the ranking algorithm. A lot of research is going on in this area especially on techniques that employ Natural Language processing[11] in order to precisely extract the meaning of the query and provide the most matching result. We can classify the functions of this module into three categories - query processing, document retrieval and result ranking[1].

1.5.1 Query processing

Similar to the methods mentioned in Section 1.4, the query from the user needs to be processed in order to extract the real intent of the query. Search engines may re-write the query and collect other information such as the user's location in order to provide the most relevant results[1]. Query re-writing includes and is not limited to removing stop words as described in Section 1.4.2, case normalization, spell check etc.

1.5.2 Document retrieval

The index data of a search engine is typically partitioned and distributed in many machines at one or possibly multiple data centers. There are two ways to distribute data to multiple machines: term-based or document-based[1].

- **Document-based** partitioning divides the index data among machines based on document identifications. Each machine contains the inverted index only related to documents hosted in this machine. During query processing, a search query is directed to all machines that host subsets of the data[1].

- **Term-based** partitioning divides the index based on terms and assigns postings of selected terms to each machine. This approach fits well for single-word queries, but it requires inter-machine result intersection for multi-word queries[1].

1.5.3 Result ranking

Once the documents that have the best match for the given query have been retrieved from the index, it needs to be ranked in the order of decreasing relevancy before it can be displayed to the user. Most users would not go beyond the first page of the results page and hence it crucial to provide the best results first. There are a variety of parameters involved in the calculation of a score for each document based on which it is ranked. At the high level, they can be classified into two groups - Query-dependent parameters and Query independent parameters[1].

- **Query dependent ranking.** As the name suggests the score for each document is calculated based on factors that are directly or indirectly related to the query in consideration. The matches for the terms in the query could be found in different parts of a document. The hit would be of different types - It could be the document title, anchor text(text in the hyperlinks to the page), URL etc[12]. Each of these hits would be treated differently and assigned a different weight.

For example, the term “download” http://www.microsoft.com/enus/download/ is an important keyword for matching query “Microsoft download” and hence the URL hit would have a higher weight and influence while ranking the pages[1].

- **Query independent ranking.** Query independent ranking is based on the document itself without any relation to the query. Such a ranking mechanism helps to classify and differentiate popular and legitimate web pages from the less popular sources. Link popularity is one such factor used in most ranking algorithms including Google’s PageRank[13]. The score is determined by the number of links pointing to the web page and the number of links pointing out of it.

Chapter 2

Challenges of Web Search Engines

2.1 Introduction

Searching for information on the internet through web search engines has become the most common online activity during the last decade. The impact of search engines on the daily lives of the common internet user is massive and ever-growing with more than 90% of all internet traffic coming through search engines. The success of these information retrieval systems lies in providing the most accurate and personalized results to each and every individual and such levels of accuracy and relevance can only be achieved with the knowledge of the search history and trends of its users. The following chapter discusses the privacy issues concerning modern search engines and some of the approaches taken towards protecting user privacy.

2.2 Privacy Concerns

Continuous innovations in search engine functionality have led to new ways of answering users' queries, such as entity answers, currency conversions, and calculators. These modern search engines combine information about users and their queries to infer the intent of the query. Also, in addition to the 'organic' results from the queries, the search engines also display advertisements that target the users' queries and behaviour. The sequence of users' queries gives away a lot more information than we think[14]. The search and query history can reveal a user's preferences, interests,

location and also sensitive data like bank details and other embarrassing content.

In 2006, AOL released a set of user search logs and it showed that re-identification of identities is possible by just using these search logs. Simple classifiers could accurately identify the gender and age of the user from the queries. Moreover, these logs could also reveal the demographics information at scale from these query logs[15]. With the advancement in machine learning and artificial intelligence and integration of modern search engines with email, personal assistants and cloud storage, there is an increasing concern about the privacy and security of the web user.

2.3 Approaches to protecting privacy

2.3.1 Private Information Retrieval

The problem of submitting a query to a web search engine while preserving the users' privacy can be seen as a Private Information Retrieval (PIR) problem. In a PIR protocol, a user can retrieve a specific value from a database while the server, which holds the database, gets no knowledge about the data requested by the user[16].

The trivial single database PIR solution is to fetch all the data from the database server such that the server does not get any insight into the information that the client intends to use. However, this naive solution is not computationally feasible in the case of search engines that store peta bytes of information in their databases.

Another improved technique suggested in [16] is to shuffle queries among different users before submitting them to the search engines in order to confuse the server and present a distorted profile. There exists a central node that groups a collection of users who want to submit a query and their corresponding queries. The node re-assigns the queries to different individuals after cryptographically hiding the source of the query in order to ensure the privacy of the user. As a result, each user submits an entirely different query than what he/she intended to submit and thereby avoiding profiling by the search engine[16].

However, there are certain assumptions that may not hold true in all cases. Each of the users must not be able to decrypt and thereby link a query back to the original user. Also, the central node must not store and profile any of the users based on their query history. And finally, while shuffling queries among random users may provide some privacy, it could lead to extreme profile distortion and affect the quality of the search results for the queries.

2.3.2 Anonymous web browsing

Another way of ensuring the privacy of search queries is through anonymous web browsing. The key idea of anonymous web browsing is to avoid direct communication with the web server by relaying communication through intermediate points on the internet.

- **Web Proxies.** A web proxy is an intermediate node on the internet through which, when configured by the user, all the web requests pass through. A browser configured to use a proxy sends all of its URL requests to the proxy instead of sending them directly to the target Web server and the proxy relays the requests to the target server. The proxies can handle both HTTP and SSL/TLS encrypted HTTPS traffic[17].

  Figure 2.1 shows the flow of traffic through the internet in cases where a proxy is configured on the client and otherwise. The traffic from clients where the proxy is not configured (green) is routed directly to the target web server through the internet routers. Whereas, in the case of proxy configured users (red) the traffic is routed to the web proxy which then forwards the requests to the target web server and the response travels back to the user through the proxy. This method avoids direct communication between the client and the web server, which in our case, is the web search engine.

  Although this may seem like a good alternative, it does not entirely eradicate

the problems related to privacy but rather move the threat of privacy from the search engines to the proxies themselves[16]. The proxies will have control over the traffic and can monitor activities of the user while forwarding the requests to the original target. The proxies are also in a position where it is possible to perform active Man-in-the-Middle (MITM) activities such as modifying the response from a web server in a way to retrieve information from the client.

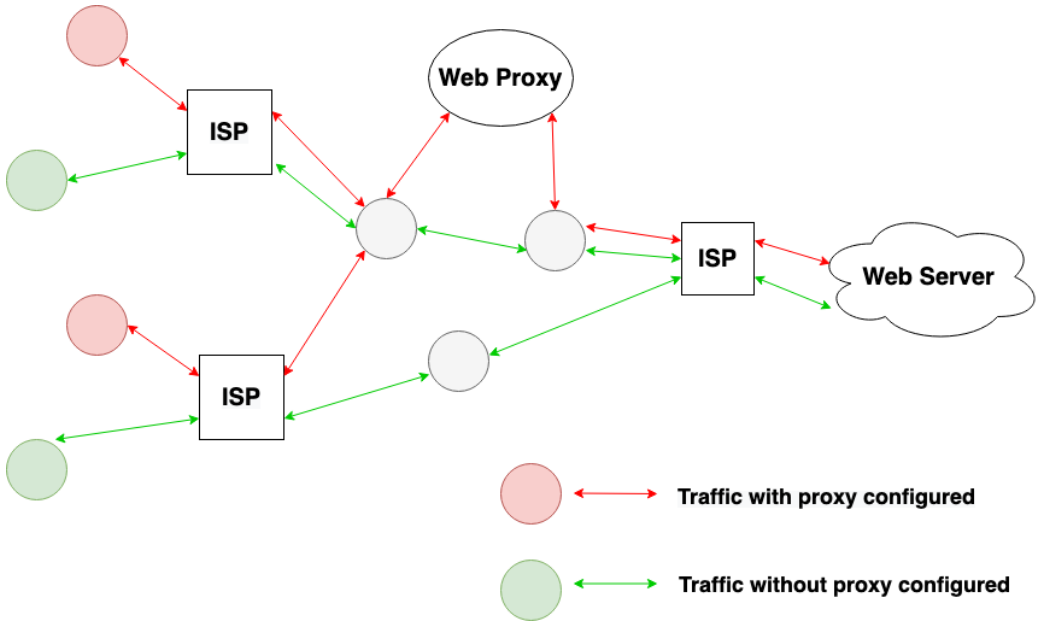

Figure 2.1: The flow of internet traffic with proxy configured(red) and without a proxy (green).

- **The Onion Routing (Tor).** Onion routing is an infrastructure that enables anonymous communication between two entities over a public network like the internet. The onion routing network or the Tor network is a collection of devices on the internet voluntarily participating in the anonymous routing mechanism. On the regular internet, IP packets have a source IP address and a destination IP address that can be seen by any router to which the packet arrives. In fact, the router needs to see the IP address in order to route the packet to the destination. But this is not the case with onion routing where the key idea is to route traffic from a source to a destination through a series of intermediary

routers while preserving the identity of the source and the destination nodes.

In order to achieve the level of anonymity, the sequence of onion routers in a route is strictly defined at connection setup and each onion router can only identify the previous and next hop along a route. Data passed along the anonymous connection appear different at each onion router and therefore data cannot be tracked en route[18].

Figure 2.2 shows the Tor infrastructure and the flow of encrypted data across the onion routers that facilitate anonymous communication. In order to communicate anonymously to Bob, Alice sends the packets to an agent router that resides at the edge of a Tor network. This router acts like a proxy server in charge of setting up the routers through which data will pass through. The agent router adds layers of encryption to the original packet with different encryption keys such that an intermediate router accessing the packet can online decrypt the outermost layer revealing its nearest neighbours. Each onion router transmitting the package only knows the addresses of its former and latter onion routers thereby preserving anonymity at the individual router level[19].

While using Tor may seem like a fairly secure alternative to protect user privacy from search engines, it has its own problems. Configuring Tor on a client is not straightforward and wrong configurations can pose security risks that may later lead to attacks[16]. Secondly, the exit nodes or nodes from which the packet leaves the Tor network on to the public internet can see the original packet and the final destination. This leaves a risk of traffic being monitored and in some cases the exit nodes are run by government agencies for surveillance purposes. Lastly, HTTP traffic through Tor introduces significant delays due to the initial connection phase and cryptographic overhead at each of its relay points [16].

- **Virtual Private Networks (VPNs).** Virtual private networks are originally used to provide secure and private internet access over an insecure public net-

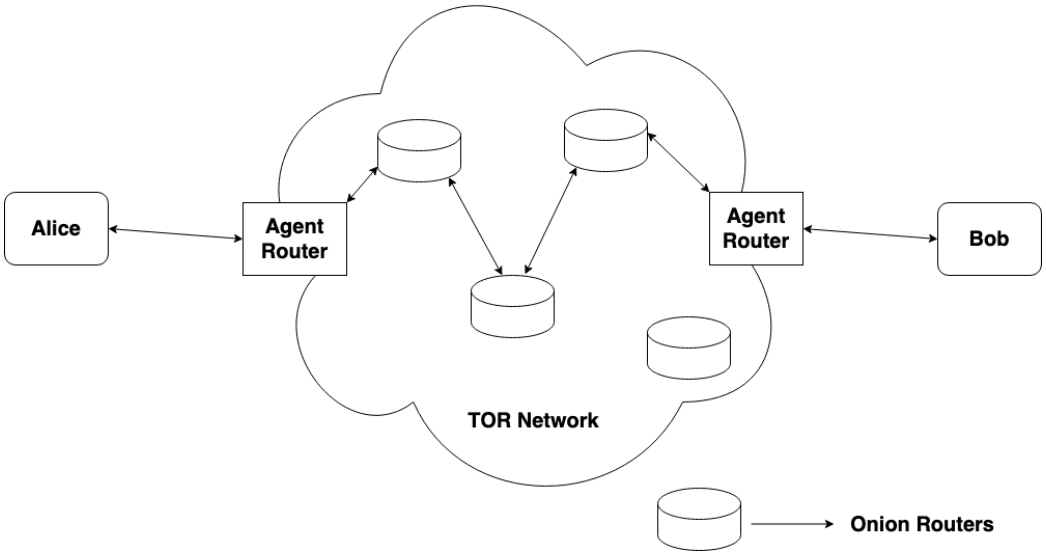

Figure 2.2: The onion routing network

work. A VPN provides a secure and encrypted tunnel from the client to the target private network. Figure 2.3 depicts the role of VPN in ensuring privacy while browsing the internet.

The connection between the client and the VPN terminator is encrypted and is impossible to eavesdrop on it and reveal the contents. The packets are securely dropped on to the 'trusted' private network and then routed to the target over the public internet. The traffic may be relayed through more than one VPN networks before it reaches its target. The web server or the search engine can track the request back to the most recent VPN network but it is difficult to trace the request back to the original client who initiated the request. Hence the identity of the user is protected behind the VPN network.

However, VPNs behave similarly to Web Proxies by acting as an intermediary between the user and the server. This leads to the problem of privacy at the VPN terminator. The VPN provider has access to the traffic, can monitor activities and analyze the browsing trends of the user. Further, some internet services put a blockade on VPNs thereby preventing users on VPNs from

accessing their content. Finally, VPNs tend to be slower due to the encryption/decryption overhead during the transit of packets.

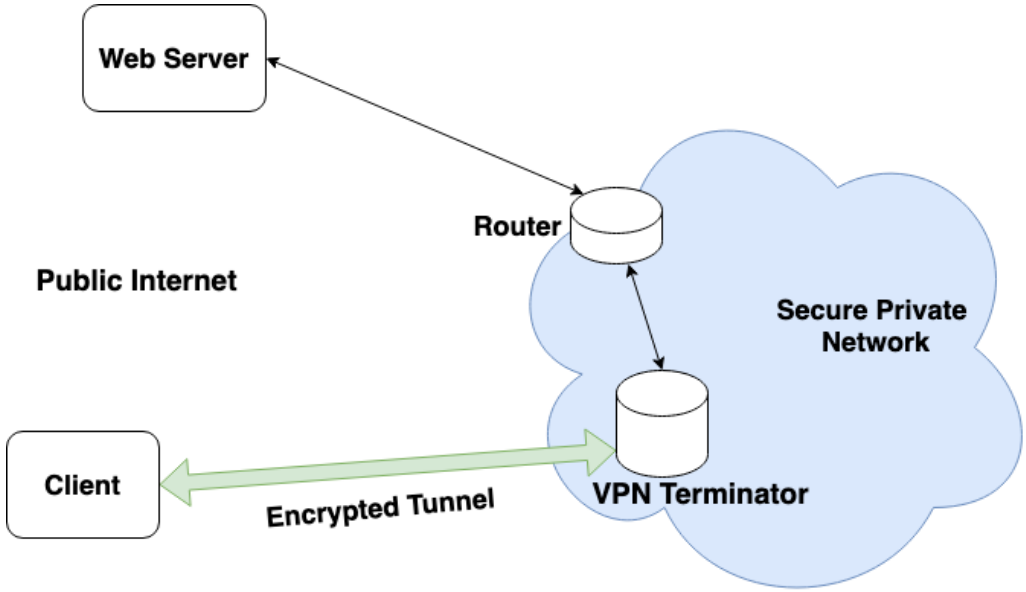

Figure 2.3: Virtual Private Network

Chapter 3

Machine learning techniques in Web Search

3.1 Introduction

Machine learning techniques have created a significant impact on a variety of domains over the last two decades. From basic linear regression problems to advanced deep neural networks, there has been tremendous innovation and technological advancement that has helped improve and optimize today's services.

The Information Retrieval realm too has benefited from the development of machine learning and artificial intelligence. Crawlers use machine learning to classify relevant pages and spam ones, predict how often pages are updated and optimize their crawl rate. Query processing modules use language models to predict the context of the query and the ranking module performs document matching and ranking based on various deep learning techniques. This chapter provides an introduction to a few key deep learning algorithms that were considered for the project such as Recurrent Neural Networks and the Transformer network.

3.2 Neural Networks - Architecture and Learning

Neural networks are the basic architecture in most deep learning algorithms and are inspired by the working of the human brain that is composed of billions of layers of neurons firing and interacting with each other. The idea of a neural network is to

make a decision based on different input parameters by processing these inputs over a number of layers of nodes called artificial neurons. These individual nodes perform a specific function on the input fed to it and provide the corresponding output.

The earliest model of an artificial neuron is a perceptron, developed in the 1950s and 1960s by the scientist Frank Rosenblatt[20]. Figure 3.1 shows the working of a perceptron node. The node takes multiple binary inputs x_1,x_2,x_3,...x_n and produces a single output. Each input x_i has a corresponding weight w_i associated with it that determines the influence of that input on the final output. The final output is either a binary 0 or 1 based on a selected threshold value. If the weighted sum of all the input to the perceptron node is greater than the threshold, the output is 1 and 0 if otherwise. The same logic can be algebraically represented as[20]:

$$output = \begin{cases} 0, & \text{if } \sum_{i=1}^{n} x_i.w_i \leq \text{threshold} \\ 1, & \text{if } \sum_{i=1}^{n} x_i.w_i > \text{threshold} \end{cases}$$

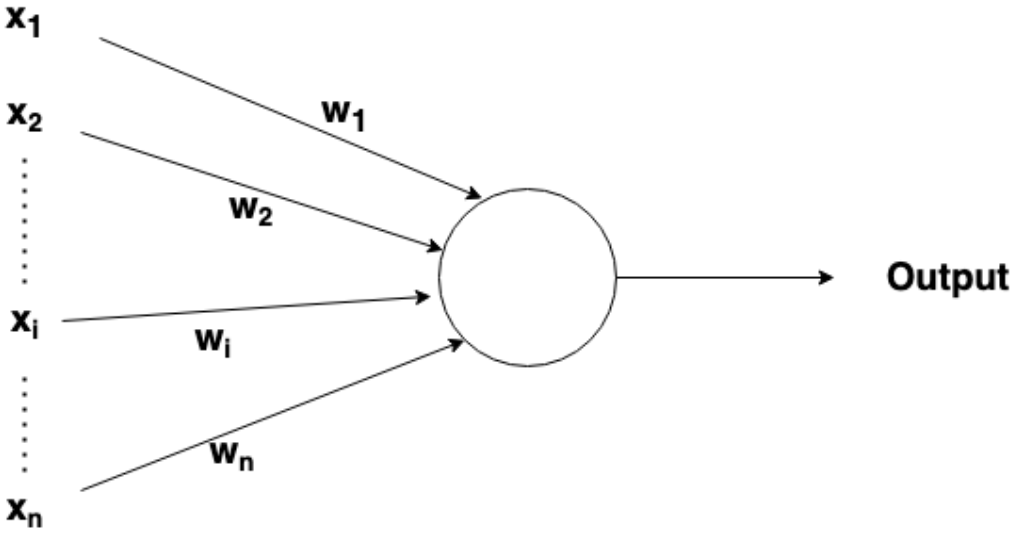

Figure 3.1: Individual perceptron node

However, there is an inherent problem with the above algebraic equation. Since

the output values are discrete, i.e either a binary 0 or 1, a small change in the input may cause the output to flip the output entirely. This change may cascade over the next layers resulting in a completely wrong output. In order to solve this problem, a new type of neuron called sigmoid neuron is introduced. Sigmoid neurons do not produce discrete values as output but rather a range of values between 0 and 1[20] as shown in Figure 3.2. The output of a sigmoid neuron can be expressed as:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

where,

$$z = \sum_{i=1}^{n} x_i.w_i + b$$

Here b = –threshold is called the bias which is a measure of ease of producing an output value of 1 by the neuron.

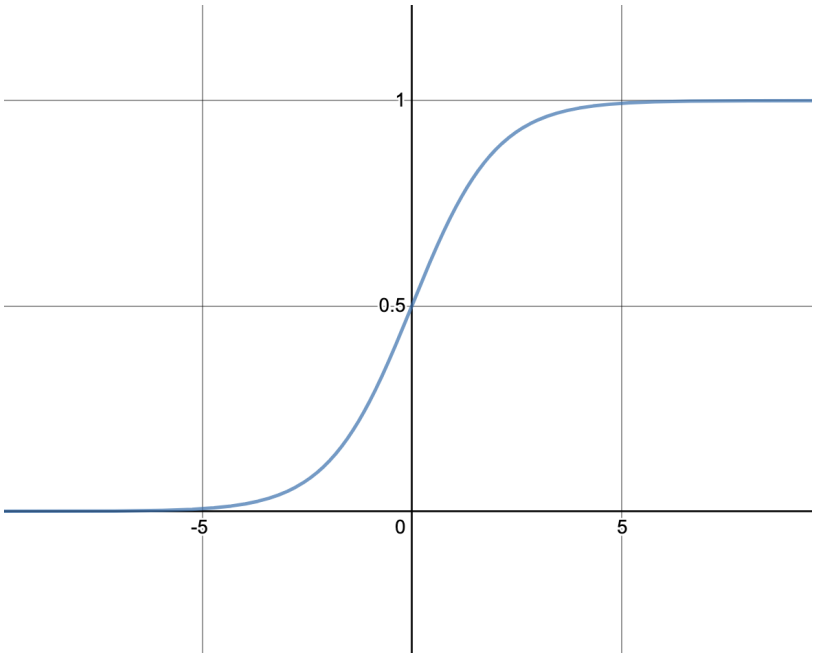

Figure 3.2: Sigmoid transfer function

A neural network is formed by combining multiple nodes like the perceptrons or sigmoid neurons over different layers in order to provide an output based on multiple input parameters as shown in Figure 3.3. The first layer is the set of input values

known as the input layer and the final layer is the output called the output layer. The middle layers are called hidden layers and the number of such hidden layers determines the depth of a neural network.

Figure 3.3 is a classic example of a neural network in which the signal flows in just one direction i.e from the input layer to the output layer through the hidden layers. Such a neural network in which there is only one direction of propagation of the signal is called a feed-forward neural network. Such networks can only determine the relationship between the input values and the output without any historical information. However, there is another variation of the neural network where the signal flows forward as well as backward in a temporal fashion. Such neural networks are called Recurrent Neural Networks and will be discussed in detail in Section 3.3.

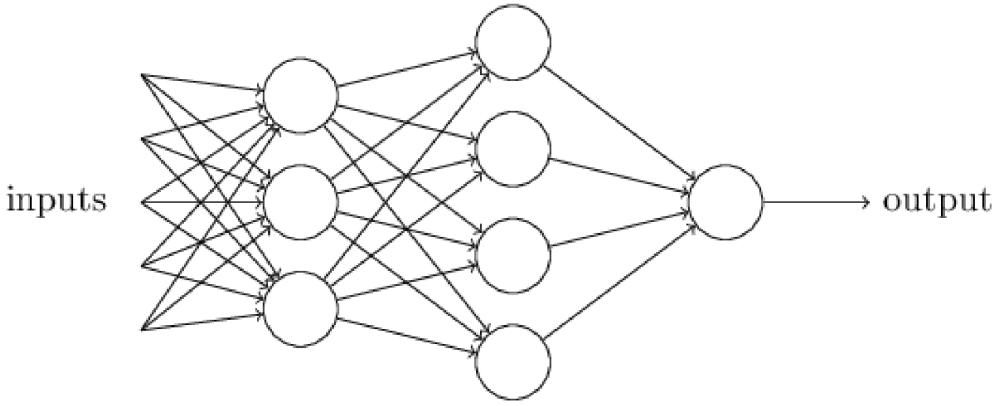

Figure 3.3: A neural network with two hidden layers[20]

A neural network does not initially perform the tasks given to it with perfect accuracy, rather it needs to be trained on the related dataset in order to 'learn' and understand the problem. The network is fed with training examples whose real outputs are known before hand and the final output from the network is used to compute the difference in these two output values, also known as the loss function. The goal of the training stage is to run the neural network over different example data in order to minimize the loss function. The learning stage can be classified into two different stages[21]:

- **Forward phase.** The neural network is initialized with random values of weights and biases and the inputs for a training instance are fed into the neural network. The neural network computes the values over the different layers and produces the final outputs. The loss function is calculated from the difference in the outputs and the derivative of the loss function with respect to the outputs is computed. The next stage is to adjust the neural network parameters in order to minimize the loss[21].

- **Backward phase.** In the backward phase, the gradient of the loss function with respect to the weights in each layer starting from the layer close to the output layer is computed. The gradient of a function gives an idea of the direction of the steepest slope of the function. Hence, using this gradient we can find and adjust the weights in order to find the minimum value of the loss function. Since, the gradient calculation and updating of weights occurs in the backward direction of the network, it is called the backward phase or backpropagation[21].

3.3 Recurrent Neural Networks

Recurrent neural networks are a modification to the classical neural networks in a way that the output of the previous time sequence is used as input to the hidden layers of the network. This modification enables the RNNs to have 'memory' of the previous outputs and is best suited for applications with sequential data like text sentences, time-series, and other discrete sequences[21].

The architecture of Recurrent Neural Networks can be explained using Figure 3.4. The output (y^t) at a given time instance t, is calculated from the input x^t and the activation of the previous time instance $a^{\langle t-1 \rangle}$. The output of the hidden state at time t-1, $a^{\langle t-1 \rangle}$ is stored and used to calculate the activation a^t at time t[22].

The blue box in Figure 3.4 is the individual artificial neuron in the RNN and has a similar function as explained in Section 3.2. The working of the individual node in

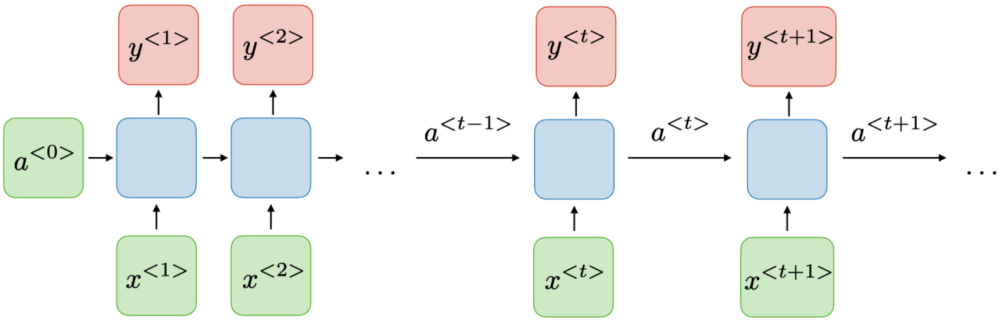

Figure 3.4: Recurrent Neural Network Architecture[22]

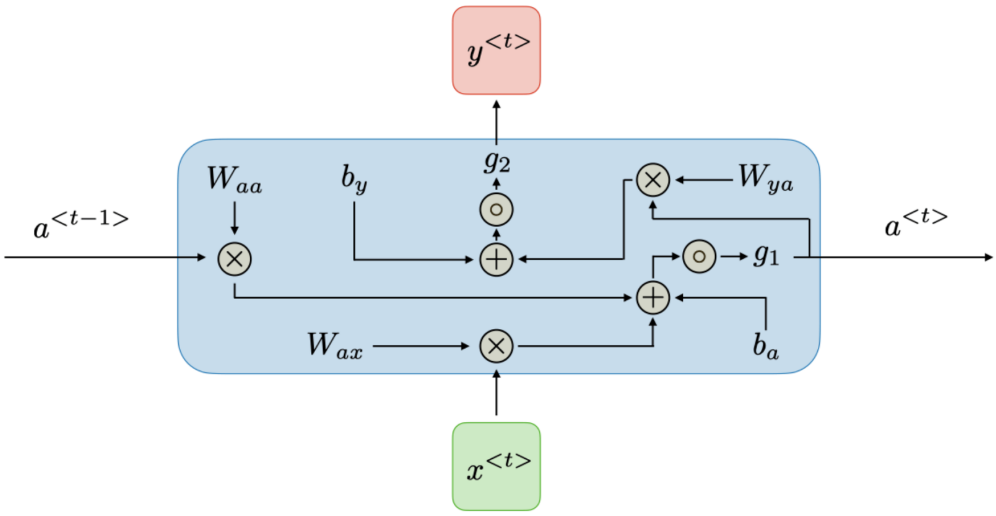

Figure 3.5: Inside the RNN neuron[22]

the RNN can be mathematically described as below:

$$a^t = g_1(W_{aa}a^{\langle t-1\rangle} + W_{ax}x^t + b_a)$$

$$y^t = g_2(W_{ay}a^t + b_y$$

where W_{aa},$W_{ax}W_{ay}$ are the weights, b_a,b_y are the biases and g_1 ,g_2 are the activation functions.

The learning process in Recurrent Neural Networks can be a little trickier than classical Neural Networks because of the temporal aspect of RNNs. The usual back-propagation method does not work in this case because it assumes there are no loops in the network and RNNs have multiple loops in their network. Hence, a modified

version of the backpropagation algorithm known as the Back Propagation through Time (BPTT) is introduced to help train the RNN[23]. The key idea of BPTT is to unfold the looped network in time so as to create multiple instances of a feed-forward network and then use back propagation algorithm on these instances to adjust the weights. Figure 3.6 shows the unfolding of an RNN into a loop-free feed forward network.

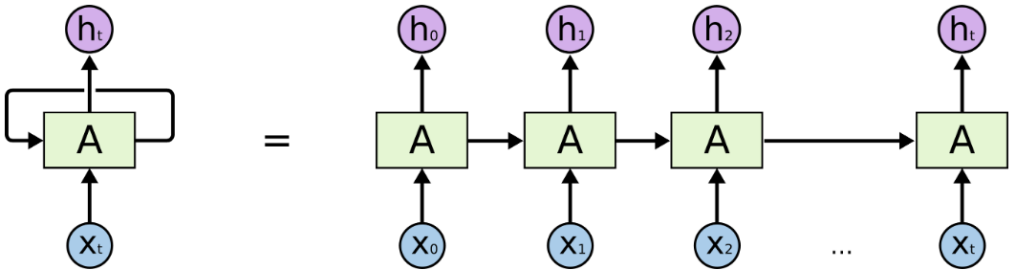

Figure 3.6: Unfolded Recurrent Neural Network[24]

Recurrent Neural Networks can perform better than static networks but the training process of RNNs is complex due to the difficulty of learning long-term dependencies. Back-propagating through the many layers may cause the gradients to shrink exponentially if the weights are small but on the other hand grow towards infinity if the weights are too large. Standard feed forward networks can deal with these exponential effects because they only have a few hidden layers, but for a recurrent network trained on a long sequence, the gradients can easily explode or vanish.[23]. An improvement to the traditional RNNs that preserves dependencies over long sequences is called the Long Short Term Memory or LSTM networks.

3.4 Long Short Term Memory networks

Long Short Term Memory networks are an improvement over Recurrent Neural Networks by mitigating the problem of vanishing or exploding gradients discussed in the previous section. In a long sequence of words, the influence of a word at the beginning on a word towards the end of the sentence will be low in the case of Recurrent

Neural Networks. For example, consider the sentence 'Ulreich lived in Germany for five years. He can speak fluent German'. In order to predict that German is the language that the person is fluent in, we need to take the first sentence into account. This provides context to the sentence based on which the prediction can be made.

LSTM network nodes contain what is called a 'cell state' that is carried through the network so that correlations between words far apart in a long sequence can be made possible. The cell state contains part of the information from the previous words in a sequence that is retained over a number of cycles of 'retaining' and 'forgetting' information as it passes through different layers of the network[21]. Figure 3.7 shows the working of a single node in an LSTM network. Unlike the RNN node, the LSTM node has four neural gates inside it - three sigmoid gates and a tanh activation function. Each of these gates plays a role in retaining, forgetting and adding new information into the cell state.

The first gate is a sigmoid activation that combines the previous hidden state h_{t-1} and current input x_t. The output of the first sigmoid ($f_{t)}$ is between 0 and 1, with a value of 0 indicating that the cell state C_{t-1} be completely forgotten, and 1 meaning retaining the entire cell state information from the previous levels. The second and

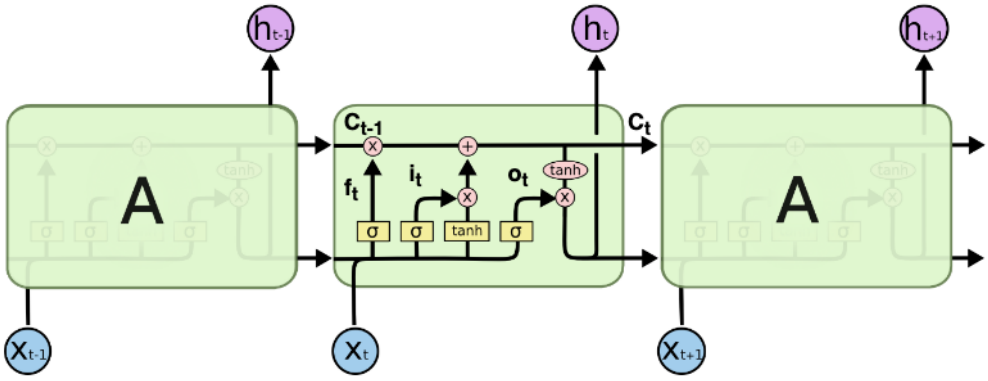

Figure 3.7: LSTM network[24]

third gates determine the amount of new information from the input to let into the cell state highway. The second layer combines the input and the previous hidden

state over a sigmoid function (i_t) that decides the amount of new input information to be allowed into the cell state while the third layer is a new activation function in the form of a hyperbolic tangent function. A tanh function maps inputs into a value between -1 and 1 as shown in Figure 3.8. The output of the tanh activation is the new candidate cell state value $C_t^{\sim}$ and is added to the previous cell state C_{t-1} after being conditioned on the output of the second layer i_t.

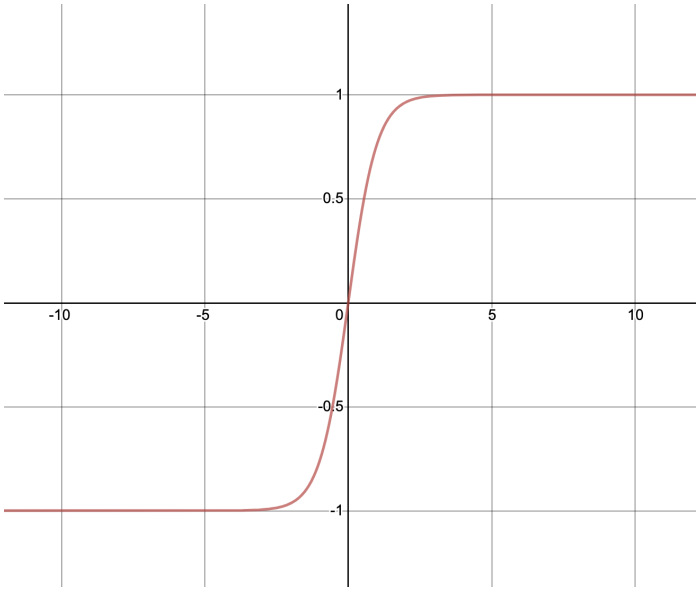

Figure 3.8: Hyperbolic tangent activation

Finally, the last sigmoid activation is called the output gate (o_t) that determines the amount of new cell state information C_t to be leaked into the hidden state of the next level h_{t+1}. Conceptually, the output vectors of each layer, f_t, i_t and o_t are known as forget, input and output gates each with the role of forgetting the previous cell state, adding new information to the cell state and leaking the new cell state information to the next hidden state respectively. The cell state equation can be mathematically expressed as[21]:

$$\vec{c}_t = \vec{f}.\vec{c}_{t-1} + \vec{i}.\vec{c}$$

From the above equation, the cell state information is continuously updated by forgetting previous cell state information and adding new information based on the newer

inputs with the help of the gate vectors. This allows more control to find correlations between words in a sentence in order to understand the context accurately.

However, even with the new architecture of LSTM networks, RNNs and LSTMs are highly sequential in nature when it comes to language modelling and understanding sentence context. It takes one word at a time into account while trying to connect different words in a sentence together to extract its true intent. The sequential nature comes with problems of high and inefficient computation that is time-consuming and one that prevents parallelization[25]. A new and state-of-the-art technique based on an attention model is discussed in Section 3.5 that overcomes the problems of sequential RNNs and is widely researched on especially for language modelling.

3.5 Transformer Networks

The transformer network model was introduced with the goal of reducing computation time in sequence modelling and translation as compared to RNNs and LSTMs. The transformer model consists of different layers of encoder-decoder blocks that can be run in parallel in order to speed up computation and reduce time. Figure 3.9 shows the architecture of a single encoder-decoder block with its sub-layers as proposed in[25].

The overall network consists of six layers of encoders and decoders, each with identical architecture but varying inputs. The initial word input is converted into a vector form known as the input embedding and is passed on to the first encoder block along with positional encoding i.e the information about the positions of each word in the sequence. The outputs from each encoder blocks are fed to the subsequent encoder block and the final output from the last encoder is fed to all of the decoder blocks

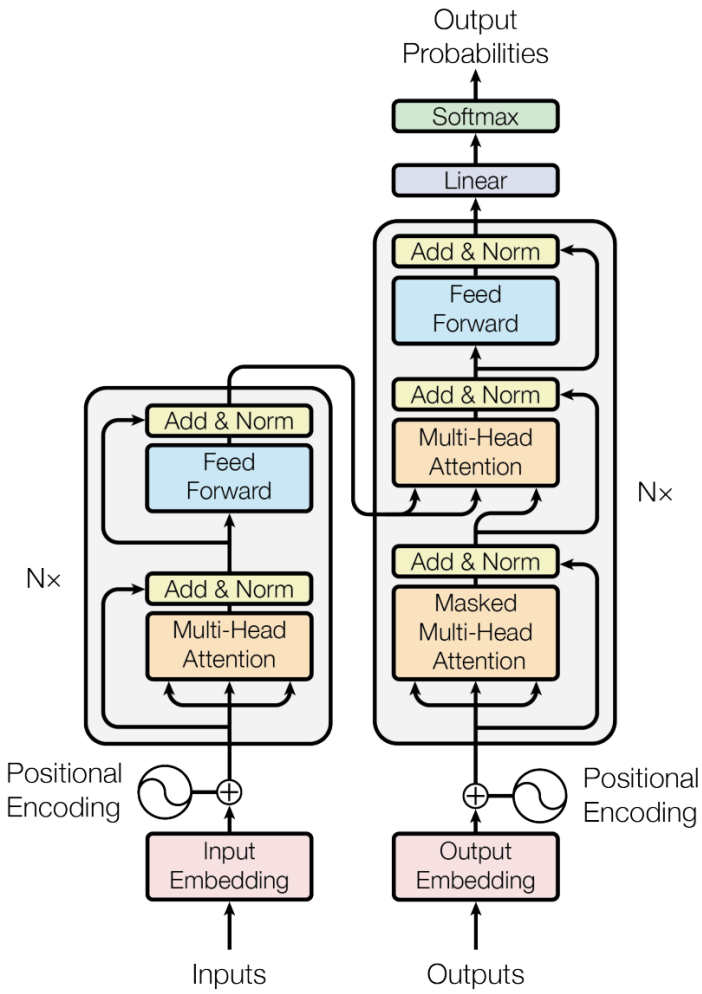

Figure 3.9: Transformer network architecture[25]

3.5.1 Encoder and Decoder blocks

- **Encoder.** The encoder block is composed of two sub-layers namely the Multi-Head Attention layer and the Feed Forward neural network layer. The functions of the Multi-Head Attention layer and the Feed Forward network is discussed in the next sections. The main idea of the encoder blocks is to transform the input sequence into a continuous sequence of vectors that contain information on how each word in the sequence relates to the other words in the same sequence. There is also a residual connection after each sub-layer in the encoder block that adds and normalizes the input with the output of the sub-layer[25].

- **Decoder.** The decoder stack works similar to the encoder with a Multi-Head Attention layer and a Feed Forward network layer with residual connections and layer normalisation functions. However, there is an additional sub-layer known Masked Multi-Head Attention layer. This layer is introduced with the purpose of preventing the decoder from gaining information on future words in the sequence while decoding a particular word. The output from this sub-layer passes on as input to the second attention layer, which also takes input from the outputs of the encoder block.

  Further, there is a Feed Forward layer and a Linear layer that processes the encoded vectors before it is fed to a Classifier. The Classifier layer generates the probabilities for each of the different sets of words and the word with the highest probability is chosen as the predicted word. Finally, the output generated from the previous steps of the decoder is fed as the input to the first decoder block[25].

3.5.2 Self Attention

The attention function in the encoder and decoder blocks is used to draw relations between the word in consideration to other words in the sequence. For example, in the sentence 'John went to bed early because he was tired', the words 'John','he' and 'tired' have a high correlation among them while words like 'bed' and 'tired' have little relation between them. The attention function uses three vectors, the Query vector (Q), the Key vector (K) and the Value vector (V) and all three vectors are derived from the input embedding of the words in the sequence. The attention layer produces an output that is a weighted sum of the value vectors, where the weight is a function of the correlation between the Query and Key vectors. The attention function here is a dot product attention represented as below[25]:

$$Attention(Q, K, V) = Softmax(\frac{Q.K^T}{\sqrt{d_k}})V$$

where d_k is the dimension of the queries and keys.

The attention function is performed on different values of Q,K and V matrices so that it provides a different representational sequence and the final score is the concatenation of these different representations. This would provide the model with the ability to focus on different positions in the sequence and prevent the word representation of a particular word from dominating on itself[25]. Figure 3.10 is a representation of the individual self attention layer and the combined multi-head attention layer.

Self-attention based models are a notable improvement to sequential models such as RNNs, LSTMs and GRUs in their ability to parallelize certain functions, thereby reducing computational time. Since, each of the self attention instances in the multi-head attention layer is independent of each other, they can be executed in parallel. Further, learning long-range dependencies is a key challenge in many sequence transduction models. The main factor affecting the ability of these models to learn such dependencies is the length of the paths forward and backward signals have to traverse in the network. Transformer networks have shorter paths between any combination of positions in the input and output sequences and hence it is easier to learn long-range dependencies[25].

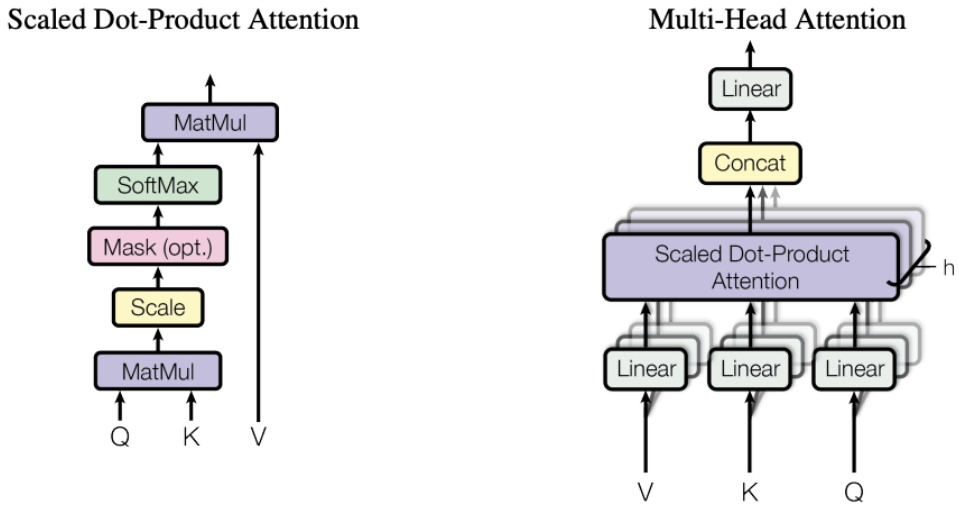

Figure 3.10: Attention layer in a Transformer network[25]

Chapter 4

Experimentation and Results

4.1 Introduction

In the last few chapters, we discussed the significance of web search engines in today's internet and the role it plays in our day-to-day activities. The architecture and behaviour of search engines were explored to gain an understanding of the core working of all common search engines. Further, the major concerns around modern web search engines with regard to user privacy were examined and approaches to mitigate the problems were identified.

However, these approaches could not completely solve the privacy problem and hence a local alternative to the web search engine is proposed. In order to create a local search engine nearly as powerful as online search engines, we dived into various machine learning techniques that could substantially improve the performance and quality of search results. This chapter describes the approach to creating a local search engine with varying search techniques, running on a single Linux virtual machine.

4.2 Methods and Procedure

The prototype search engine consists of three basic modules - a python-based web crawler, a non-relational database and an interactive web app capable of collecting user queries and displaying the search results. The engine runs on a virtual machine with 2 CPU cores, 4GB of memory, 40 GB of disk space and Ubuntu 20.04 LTS (Focal

Fossa) operating system over a VMware hypervisor.

Due to resource constraints in the VM, the prototype search engine was limited to gathering and searching information on a specific domain and not the entire web. However, the same model can be scaled with sufficient hardware resources to widen the scope of the search engine. A collection of cooking recipes was chosen as the domain for the prototype with the python-based web crawler scraping over 36,000 recipes from the web. These recipes were processed and stored in a non-relational database called Elasticsearch. Elasticsearch is an open-sourced database based upon the famous search library Apache Lucene[26]. Finally, the query and search result module is a Python web app running on the Flask web app framework on the same virtual machine. Figure 4.1 describes the components and behaviour of the prototype search engine.

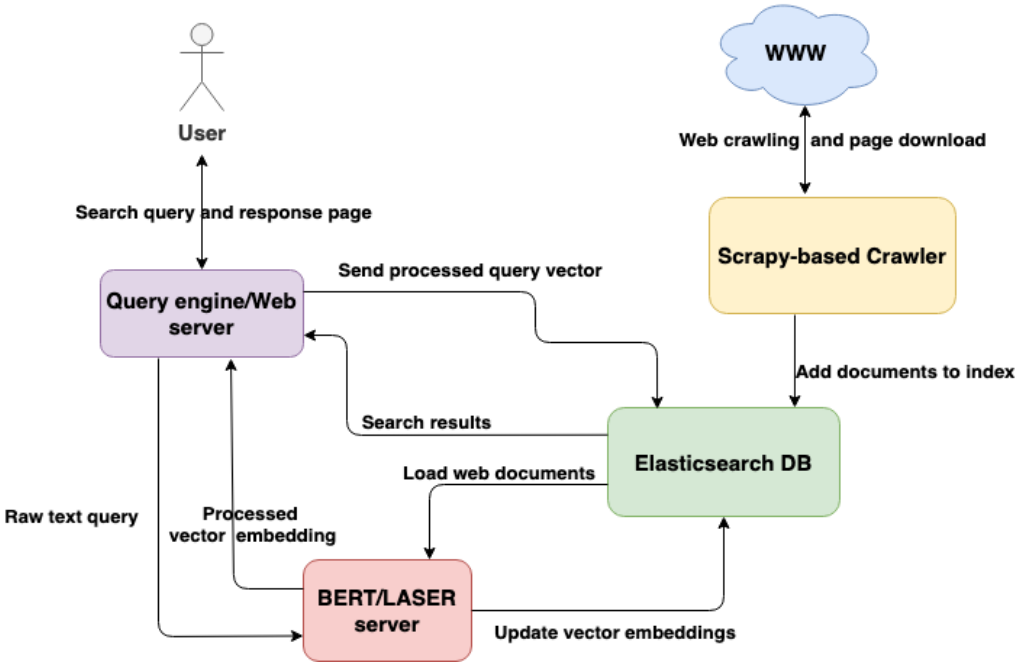

Figure 4.1: Composition of the local search engine

To understand the impact of machine learning techniques in web information retrieval systems, an experiment is conducted to compare the performance of the prototype search engine in terms of relevant search results in different scenarios. Different

techniques are employed to help the search engine retrieve relevant search results namely - TF-IDF, Laser and BERT against a standard set of queries and the search output quality is compared.

The traditional term search technique, Term-frequency Inverse document frequency (TF-IDF) is an old but useful technique used for document retrieval against a query. The method calculates the number of occurrences of each indexed term in each document known as the term frequency of that term in that document. Also, we compute a parameter that gives an indication of how rare a term is in the entire collection of documents known as document frequency. Now, each term in the query is used to compute the TF-IDF score for every document as per the following equation:

$$TF - IDF = \frac{f_{t,d}}{log(\frac{N}{n_t})}$$

where $f_{t,d}$ is the number of occurrences of term t in document d, N is the total number of documents in the collection and n_t is the number of documents containing term t. Those documents with the highest scores for the given query are picked and returned to the user.

LASER or Language Agnostic SEntence Representation is a sentence representation model based on Bi-directional LSTM encoders described in Section 3.4. Bi-directional encoders capture a better understanding of the sequence as they can generate correlations between words that appear before and after the word under consideration. The LASER model produces an output vector of dimension 1024[27].

BERT or Bi-directional Encoder Representations from Transformers is a language model developed by Google and is based on encoders from the transformer network described in Section 3.5. A pre-trained BERT model composed of 12 layers is used and it produced an output vector of dimension 768.

All documents are pre-processed and the vector representations are stored in the database for each document. Cosine similarity function is used to calculate the similarity between the document and the query. Cosine similarity function aims to calcu-

late how similar two vectors are based on the cosine of the angle formed by those two vectors. Each web document is also scored based on query independent parameters and is assigned a weight based on the content in its title, description and body.

4.3 Results and Discussion

In order to compare the different document matching techniques, the relevance of the search results is evaluated with the help of Precision and Recall metrics. Precision is the ratio of the number of relevant documents retrieved to the total number of the documents retrieved][3]. Precision can be expressed as:

$$Precision, P = \frac{\#\ Relevant\ documents\ retrieved}{\#\ Documents\ retrieved}$$

While precision gives an idea of how many relevant results are fetched, recall provides information on the number of relevant documents missed. Recall is the ratio of the number of relevant documents retrieved to the total number of relevant documents in the entire document collection[3].

$$Recall, R = \frac{\#\ Relevant\ documents\ retrieved}{\#\ Relevant\ documents\ in\ collection}$$

However, there is a trade-off problem associated with Precision and Recall. One can achieve a recall value of 1 by fetching all documents from the collection irrespective of the query. But this would mean a poor precision value. On the other hand, precision can be improved by reducing the number of documents fetched but may result in missing out on some relevant documents, thereby decreasing recall. In order to mitigate this problem and to arrive at a balanced trade-off, the F measure is used to evaluate search performance. The F measure is a single measure that trades off precision versus recall, and is the weighted harmonic mean of precision and recall[3]:

$$F = \frac{1}{\alpha\frac{1}{P} + (1-\alpha)\frac{1}{R}} = \frac{(\beta^2+1)PR}{\beta^2 P + R}\ \text{, where } \beta^2 = \frac{1-\alpha}{\alpha}$$

The test was conducted by running a set of 50 standard queries on the search engine in three different cases, each with one of the above-mentioned techniques for

document matching. A higher value of F meant that the search results were more relevant, and a lower value indicated less relevant search results. Table 4.1 outlines the results of the test for the three different techniques with $\alpha = 0.5$ or $\beta = 1$. This value of $\beta = 1$ provides a balanced emphasis on precision and recall. A value of $\beta < 1$ prioritizes precision over recall and vice-versa.

Table 4.1: Precision, Recall and F values for different document matching techniques

Method	**Precision**	**Recall**	**F measure**
TF-IDF	0.252	0.370	0.150
LASER	0.374	0.441	0.201
BERT	0.410	0.554	0.237

From Table 4.1, it is inferred that the TF-IDF technique provides the least quality search output with regard to relevancy. The TF-IDF technique does not account for the semantic context of the query but rather retrieves those documents with an exact word match. Documents that may contain relevant information on the query may be missed out because it does not contain the search terms used in the query. However, LASER and BERT in this case, perform better in providing relevant search results because of these models' ability to extract the context of the query rather than merely matching words. There is a slight improvement in the performance of the BERT model over LASER due to the fact that some of the queries were too long for the LSTM model to predict the context accurately.

The vector representations of words and sentences that convey the same meaning are closely related. Hence, their cosine similarity will be higher than those words or sentences with a different meaning. Thus, mapping the text sequences onto the vector space and correlating these vectors in order to identify similar information is proved to improve the performance of information retrieval systems.

Bibliography

[1] T. Yang and A. Gerasoulis, *Web search engines: Practice and experience*, 2014. DOI: 10.1201/b16812.

[2] M. Najork, "Web crawler architecture," in *Encyclopedia of Database Systems*, Encyclopedia of Database Systems. Springer Verlag, Sep. 2009. [Online]. Available: https://www.microsoft.com/en-us/research/publication/web-crawler-architecture/.

[3] C. D. Manning, P. Raghavan, and H. Schutze, *Introduction to Information Retrieval*. 2008. DOI: 10.1017/cbo9780511809071.

[4] P. G. Chaitra, V. Deepthi, K. P. Vidyashree, and S. Rajini, "A study on different types of web crawlers," in *Intelligent Communication, Control and Devices*, S. Choudhury, R. Mishra, R. G. Mishra, and A. Kumar, Eds., Singapore: Springer Singapore, 2020, pp. 781–789.

[5] S. Sharma and P. Gupta, "The anatomy of web crawlers," in *International Conference on Computing, Communication Automation*, 2015, pp. 849–853. DOI: 10.1109/CCAA.2015.7148493.

[6] J. Cho and H. Garcia-Molina, "Parallel crawlers," in *Proceedings of the 11th International Conference on World Wide Web, WWW '02*, 2002. DOI: 10.1145/511446.511464.

[7] Q. Tan and P. Mitra, "Clustering-based incremental web crawling," *ACM Transactions on Information Systems*, vol. 28, no. 4, Nov. 2010, ISSN: 1046-8188. DOI: 10.1145/1852102.1852103. [Online]. Available: https://doi-org.login.ezproxy.library.ualberta.ca/10.1145/1852102.1852103.

[8] D. N. Singhal, A. Dixit, and A. Sharma, "Design of a priority based frequency regulated incremental crawler," *International Journal of Computer Applications*, vol. 1, Feb. 2010. DOI: 10.5120/23-131.

[9] M. F. Porter, "An algorithm for suffix stripping," *Program*, vol. 14, 3 1980, ISSN: 00330337. DOI: 10.1108/eb046814.

[10] I. Zeroual and A. Lakhouaja, "Arabic information retrieval: Stemming or lemmatization?" In *2017 Intelligent Systems and Computer Vision (ISCV)*, 2017, pp. 1–6. DOI: 10.1109/ISACV.2017.8054932.

[11] S. Adindla and U. Kruschwitz, "Combining the best of two worlds: Nlp and ir for intranet search," in *Proceedings of the 2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology - Volume 01*, ser. WI-IAT '11, USA: IEEE Computer Society, 2011, pp. 483–488, ISBN: 9780769545134. DOI: 10.1109/WI-IAT.2011.187. [Online]. Available: https://doi-org.login.ezproxy.library.ualberta.ca/10.1109/WI-IAT.2011.187.

[12] L. Page and S. Brin, "The anatomy of a large-scale hypertextual web search engine," *Computer Networks*, vol. 30, 1-7 1998, ISSN: 13891286. DOI: 10.1016/s0169-7552(98)00110-x.

[13] M. Bianchini, M. Gori, and F. Scarselli, "Inside pagerank," *ACM Transactions on Internet Technology*, vol. 5, no. 1, pp. 92–128, Feb. 2005, ISSN: 1533-5399. DOI: 10.1145/1052934.1052938. [Online]. Available: https://doi.org/10.1145/1052934.1052938.

[14] S. Preibusch, "The value of web search privacy," *IEEE Security and Privacy*, vol. 13, 5 2015, ISSN: 15584046. DOI: 10.1109/MSP.2015.109.

[15] A. Korolova, K. Kenthapadi, N. Mishra, and A. Ntoulas, "Releasing search queries and clicks privately," in *WWW'09 - Proceedings of the 18th International World Wide Web Conference*, 2009. DOI: 10.1145/1526709.1526733.

[16] J. Castellà-Roca, A. Viejo, and J. Herrera-Joancomartí, "Preserving user's privacy in web search engines," *Computer Communications*, vol. 32, 13-14 2009, ISSN: 01403664. DOI: 10.1016/j.comcom.2009.05.009.

[17] G. V. Jourdan, "Centralized web proxy services: Security and privacy considerations," *IEEE Internet Computing*, vol. 11, 6 2007, ISSN: 10897801. DOI: 10.1109/MIC.2007.122.

[18] M. G. Reed, P. F. Syverson, and D. M. Goldschlag, "Anonymous connections and onion routing," *IEEE Journal on Selected Areas in Communications*, vol. 16, 4 1998, ISSN: 07338716. DOI: 10.1109/49.668972.

[19] Q. Fang, S. Liu, and R. Zhou, "The application of onion routing in anonymous communication," in *2010 International Conference on MultiMedia and Information Technology, MMIT 2010*, vol. 1, 2010. DOI: 10.1109/MMIT.2010.13.

[20] S. Theodoridis, *Neural networks and deep learning*, 2015. DOI: 10.1016/b978-0-12-801522-3.00018-5.

[21] C. C. Aggarwal, *Neural Networks and Deep Learning, A Textbook*. Springer, 2018, p. 497, ISBN: 978-3-319-94462-3. DOI: 10.1007/978-3-319-94463-0.

[22] A. Amidi and S. Amidi, *Recurrent neural networks cheatsheet star*. [Online]. Available: https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks.

[23] J. Franke, W. K. Härdle, and C. M. Hafner, "Neural networks and deep learning," in *Statistics of Financial Markets: An Introduction*. Springer International Publishing, 2019, pp. 459–495, ISBN: 978-3-030-13751-9. DOI: 10.1007/978-3-030-13751-9_19. [Online]. Available: https://doi.org/10.1007/978-3-030-13751-9_19.

[24] C. Olah, *Understanding lstm networks*, Aug. 2015. [Online]. Available: http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[25] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17, Long Beach, California, USA: Curran Associates Inc., 2017, pp. 6000–6010, ISBN: 9781510860964.

[26] *Welcome to apache lucene*. [Online]. Available: https://lucene.apache.org/.

[27] M. Artetxe and H. Schwenk, "Massively multilingual sentence embeddings for zero-shot cross-lingual transfer and beyond," *CoRR*, vol. abs/1812.10464, 2018. arXiv: 1812.10464. [Online]. Available: http://arxiv.org/abs/1812.10464.

[28] B. Benny, *Local-web-search*, https://github.com/bijinbenny/local-web-search.git, 2020.

[29] B. Benny, *Local-web-search-gui*, https://github.com/bijinbenny/local-web-search-gui.git, 2020.

[30] A. Sigogne, *Web-search-engine*, https://github.com/AnthonySigogne/web-search-engine, 2017.

[31] A. Sigogne, *Web-search-engine-ui*, https://github.com/AnthonySigogne/web-search-engine-ui, 2017.

Appendix A: Code Listing

A.1 New and modified code

This section describes the code snippets that were used to conduct the experiment in Section 4.2. The code is written in Python 3.0 and is either written from scratch or borrowed and modified from the respective cited sources. Modified code is labelled with the author of the modification and reason for modification. The source code is also published on Github in repositories [28] and [29].

Listing A.1: Vector mapping of text documents - vectorize.py

```
from elasticsearch import Elasticsearch
from bert_serving.client import BertClient
import json
from laserembeddings import Laser
import sys


__author__ = "Bijin Benny"
__email__ = "bijin@ualberta.ca"
__license__ = "MIT"
__version__ = "1.0"


LASER = 'laser_vector'
BERT  = 'bert_vector'

laser = Laser()

#Elasticsearch DB client
es = Elasticsearch(hosts="http://bijin:Samsung1!@localhost:9200/")

#Client connection to local BERT server
bc = BertClient(ip='localhost',output_fmt='list')

"""
doVectorize() pulls entries from the database and maps the text sequences into the
vector space using either one of LASER or BERT based on the input parameter.
BERT produces 768 dimensional vector while LASER outputs a 1024 dimensional vector
Argument : vector_type (String) --> bert_vector or laser_vector
"""
def doVectorize(vector_type):
  """
  Elastic search has a max search result limit of 10000 documents
  Hence, loop through until all documents are fetched
  """
```

```
  while(True):

    #Search query to fetch all documents with empty vector field
    response = es.search(index="web-en",size=10000,body={
      "query": {
        "bool": {
          "must_not": {
            "exists": {
              "field": vector_type } } }  }
    })
    print(response)
    hits = response['hits']['hits']

    #If length of hits list is 0, then all documents were fetched
    if(len(hits) == 0):
      break

    data = []
    for hit in hits:
      doc_map = {}
      doc_id = hit['_id']
      text = hit['_source']['title']
      if(text == '' or len(text)>200):
        text = 'N/A'
      if(vector_type is LASER):
        text_vector = laser.embed_sentences([text],lang='en')[0]
      else:
        text_vector = bc.encode([text])[0]

      doc_map[doc_id] = text_vector
      data.append(doc_map)
      source_to_update = {"doc" : { vector_type : text_vector } }

      #Update the document in the database with the new vector values
      r = es.update(index="web-en",id=doc_id,body=source_to_update)
      print(str(doc_id)+" "+ str(r))

#Main function
def main():
  error_msg = "**********************************\nInvalid script usage!\n \
  Usage : python vectorize.py <vector_type\nVector type : 'LASER' or 'BERT'\n \
  **********************************"
  if(len(sys.argv) < 2):
    sys.exit(error_msg)
  vector = str(sys.argv[1]).lower()
  if(not(vector == 'laser' or vector == 'bert')):
    sys.exit(error_msg)
  vector = "".join([vector,"_vector"])
  doVectorize(vector)

if __name__== "__main__":
  main()
```

Listing A.2: Code to run tests and generate results - run_tests.py

```
from pprint import pprint
from elasticsearch import Elasticsearch
from bert_serving.client import BertClient
import pandas as pd
from laserembeddings import Laser
import sys



__author__ = "Bijin Benny"
__email__ = "bijin@ualberta.ca"
__license__ = "MIT"
```

```
__version__ = "1.0"

#Client connection to local BERT server
bc = BertClient(ip='localhost', output_fmt='list')

#Instance of the LASER language model
laser = Laser()

#Elasticsearch DB client
client = Elasticsearch(hosts="http://bijin:Samsung1!@localhost:9200/")

"""
createScript function creates custom database queries based on the search type.
The search type includes basic TF-IDF term search, LASER vector and
BERT vector similarity searches. The function returns a unique query
for each of the scenarios.
Arguments :
query        : Text form of the query
search type  : Type of search, i.e term, laser or bert
query_vector : Vector form of the query for cosine similarity
"""
def createScript(query,search_type,query_vector):
    if(search_type == 'term'):
        return {  "simple_query_string" : {
        "query": query,
         "fields": ["title"],
         "default_operator": "and"
            }
        }
    elif(search_type == 'bert'):
        return {
        "script_score": {
         "query": {
             "multi_match": {
             "query": query,
                "type": "best_fields",
                "fields": [ "title" ]
                }
            },
        "script": {
           "source": "cosineSimilarity(params.query_vector,'bert_vector') + 1.0",
           "params": {"query_vector": query_vector}
       }
       }}
    else:
        return {
        "script_score": {
         "query": {
             "multi_match": {
             "query": query,
                "type": "best_fields",
                "fields": [ "title" ]
                }
            },
        "script": {
           "source": "cosineSimilarity(params.query_vector,'laser_vector') + 1.0",
           "params": {"query_vector": query_vector}
        }
        }  }


"""
doRunTest function performs the tests based on the 50 standard queries.
The queries are loaded from the 'Recipes.csv' file and run against the
search engine. The output is a csv file 'results_<type>.csv' containing
the top 10 results for each query item
Arguments :
search_type : Type of search technique to run
```

```
"""
def doRunTest(search_type):
    #Load the input test queries
    df = pd.read_csv ('Recipes.csv')

    resultMatrix = [['N/A' for i in range(50)] for j in range(10)]

    #Loop through each query and collect the results
    for i in range(len(df)):
        recipe = df.loc[i,"Recipe"]
        query_vector = ''

        #Generate the vector embedding of the query in case of laser or bert
        if(search_type == 'bert'):
            query_vector = bc.encode([recipe])[0]
        elif(search_type == 'laser'):
            query_vector = laser.embed_sentences([recipe],lang='en')[0]

        #Generate the database query based on the search type
        q = createScript(recipe,search_type,query_vector)

        #Database query
        response = client.search(
         index="web-en",
         body={
            "size": 10,
            "query": q,
            "_source": {"includes": ["title"]}
            }
        )

        #Aggregate results and save to output csv file
        raw_results = response['hits']['hits']
        for j in range(len(raw_results)):
            resultMatrix[j][i] = raw_results[j]['_source']['title']

    for i in range(10):
        df['Result'+str(i+1)] = resultMatrix[i]

    df.to_csv('results_'+search_type+'.csv',encoding='utf-8')


#Main function
def main():
    error_msg = "*********************************\nInvalid script usage!\n \
    Usage : python interactive_query.py <Type> \nType : 'TERM', 'LASER' or  \
    'BERT'\n*********************************"
    if(len(sys.argv) < 2):
        sys.exit(error_msg)
    param = str(sys.argv[1]).lower()
    if(not(param == 'laser' or param == 'bert' or param == 'term')):
        sys.exit(error_msg)
    doRunTest(param)


if __name__ == "__main__":
    main()
```

Listing A.3: Server code for search engine - index.py (Server)[30]

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
The web server to handle requests from the clients to search queries
and handle requests for crawling new web pages. The server runs on the
python web-server framework called Flask. Incoming client requests are
handled and tasks are added to Redis task queues for processing.
```

```
"""

__author__ = "Anthony Sigogne"
__copyright__ = "Copyright 2017, Byprog"
__email__ = "anthony@byprog.com"
__license__ = "MIT"
__version__ = "1.0"

import re
import os
import url
import crawler
import requests
import json
import query
from flask import Flask, request, jsonify
from language import languages
from redis import Redis
from rq import Queue
from multiprocessing import Process
from multiprocessing import Queue as Q
from twisted.internet import reactor
from rq.decorators import job
from scrapy.crawler import CrawlerRunner
from urllib.parse import urlparse
from datetime import datetime
from elasticsearch import Elasticsearch
from flask_rq2 import RQ
import logging
from twisted.internet import reactor


#Initialize the flask application
app = Flask(__name__)
with app.app_context():
    from helper import *

"""
__author__      : Bijin Benny
__email__       : bijin@ualberta.ca
__license__     : MIT
__version__     : 1.0
Modification    : The native Redis library used in the original reference is
                  outdated and is modified to use the new redis library specific
                  to Flask apps

Configure and intialize Redis task queue
"""
app.config['RQ_REDIS_URL']='redis://localhost:6379/0'
redis_conn = RQ(app)

"""
__author__      : Bijin Benny
__email__       : bijin@ualberta.ca
__license__     : MIT
__version__     : 1.0
Modification    : The deprecated elasticsearch library elasticsearch_dsl is
                  removed and replaced with the new elasticsearch library for
                  ES clients

Load environment variables and create elastic search DB client
"""
host = os.getenv("HOST")
user = os.getenv("USERNAME")
pwd  = os.getenv("PASSWORD")
port = os.getenv("PORT")
es = Elasticsearch(hosts="http://"+user+":"+pwd+"@"+host+":"+port+"/")
```

```
"""
__author__      : Bijin Benny
__email__       : bijin@ualberta.ca
__license__     : MIT
__version__     : 1.0
Modification    : Logging framework is added to the code to enable better debugging
                  through logs

Set logging information
"""
logging.basicConfig(filename=datetime.now().strftime('server_%d_%m_%Y.log'),
level=logging.DEBUG, format='%(asctime)s %(levelname)-8s %(message)s')


logging.info(es.info())

"""
__author__      : Bijin Benny
__email__       : bijin@ualberta.ca
__license__     : MIT
__version__     : 1.0
Modification    : The DB mapping/schema used in the orginal code is specific to
                  the application the code was used for and needs to be modified
                  to store information specific to the project experiment

Database schema used to create the DB index if the server is running for
the first time. Ignores the schema if the index already exists.
"""
settings = {
    "settings": {
        "number_of_shards": 1,
        "number_of_replicas": 0
    },
    "mappings": {
            "properties": {
                "url": {
                    "type": "keyword"
                },
                "domain":{
                    "type": "keyword"
                },
                "title":{
                    "type": "text",
                    "analyzer": "english"
                },
                "description":{
                    "type": "text",
                    "analyzer": "english"
                },
                "body":{
                    "type": "text",
                    "analyzer": "english"
                },
                "weight":{
                    "type": "long"
                },
                "bert_vector":{
                    "type": "dense_vector",
                    "dims": 768
                },
                "laser_vector":{
                    "type": "dense_vector",
                    "dims": 1024
                }
            }
    }

}
```

```
es.indices.create(index='web-en',ignore=400,body=settings)


"""
Server endpoint for crawl requests. Crawl requests with list of urls
to crawl is handled by this handler
URL : /explore
Method : HTTP POST
POST Data : url - list of urls to explore
Returns success or error message depending on the task being processed
in the Redis queue.
"""
@app.route("/explore", methods=['POST'])
def explore():

    data = dict((key, request.form.get(key)) for key in request.form.keys())
    if "url" not in data :
        raise InvalidUsage('No url specified in POST data')

    logging.info("launch exploration job")
    job = explore_job.queue(data["url"])
    job.perform()

    return "Exploration started"

@redis_conn.job('low')
def explore_job(link) :
    """
    Explore a website and index all urls (redis-rq process).
    """
    logging.info("explore website at : %s"%link)

    try :
        link = url.crawl(link).url
    except :
        return 0

    def f(q):
        try:
            """
            __author__      : Bijin Benny
            __email__       : bijin@ualberta.ca
            __license__     : MIT
            __version__     : 1.0
            Modification    : The original code used CrawlerProcess class from
            scrapy library to crawl web pages. However, CrawlerProcess class could
            not run parallely in Redis tasks threads. CrawlerProcess was replaced by
            CrawlerRunner class that could run parallely in multiple Redis tasks
            """
            runner = CrawlerRunner({
                'USER_AGENT': "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 \
                (KHTML, like Gecko) Chrome/55.0.2883.75 Safari/537.36",
                'DOWNLOAD_TIMEOUT':100,
                'DOWNLOAD_DELAY':0.25,
                'ROBOTSTXT_OBEY':True,
                'HTTPCACHE_ENABLED':False,
                'REDIRECT_ENABLED':False,
                'SPIDER_MIDDLEWARES' : {
                'scrapy.downloadermiddlewares.robotstxt.RobotsTxtMiddleware':True,
                'scrapy.spidermiddlewares.httperror.HttpErrorMiddleware':True,
                'scrapy.downloadermiddlewares.httpcache.HttpCacheMiddleware':True,
                'scrapy.extensions.closespider.CloseSpider':True
                },
                'CLOSESPIDER_PAGECOUNT':500 #only for debug
            })
            runner.crawl(crawler.Crawler, allowed_domains=[urlparse(link).netloc],
            start_urls = [link,], es_client=es, redis_conn=redis_conn)
            d = runner.join()
```

```
            d.addBoth(lambda _: reactor.stop())
            reactor.run()
            q.put(None)
        except Exception as e:
            q.put(e)
    q = Q()
    p = Process(target=f, args=(q,))
    p.start()
    result = q.get()
    p.join()

    if result is not None:
        raise result
    return 1


"""
Server endpoint to handle search queries from the web-client.
Forwards the query to the Elasticsearch DB and return the top
relevant results.
URL : /search
Method : HTTP POST
POST Data : query - The search query
            hits  - The number of results to be returned
            start - Start number for the hits (for pagination purpose)
"""
@app.route("/search", methods=['POST'])
def search():
    def format_result(hit, highlight) :
        #Highlight title and description
        title = hit["title"]
        description = hit["description"]
        if highlight :
            if "description" in highlight :
                description = highlight["description"][0]+"..."
            elif "body" in highlight :
                description = highlight["body"][0]+"..."


        #Create false title and description for better user experience
        if not title :
            title = hit["domain"]
        if not description :
            description = url.create_description(hit["body"])+"..."

        return {
            "title":title,
            "description":description,
            "url":hit["url"],
            "thumbnail":hit.get("thumbnail", None)
        }

     #Analyze and validate the user query
    data = dict((key, request.form.get(key)) for key in request.form.keys())
    logging.info("[search request data : "+ str(data))
    if "query" not in data :
        raise InvalidUsage('No query specified in POST data')
    start = int(data.get("start", "0"))
    hits = int(data.get("hits", "10"))
    if start < 0 or hits < 0 :
        raise InvalidUsage('Start or hits cannot be negative numbers')
    groups = re.search("(site:(?P<domain>[^ ]+))?( ?(?P<query>.*))?",
        data["query"]).groupdict()
    logging.info("Expression query : " + str(groups["query"]))

    """
             __author__      : Bijin Benny
            __email__        : bijin@ualberta.ca
```

```
            __license__      : MIT
            __version__      : 1.0
            Modification     : The referenced code included searching web pages
    based on their domains as well as search queries. Domain search was irrelevant
    to the experiment use case and the code is modified to perform only query search
    Send search request to Elastic search DB with the user query
    """
    response = es.search(index="web-en",body=query.expression_query(groups["query"]))
    logging.info("Raw response" + str(response))
    results = []

    #Process, sort and return the results back to the user
    for domain_bucket in response['aggregations']['per_domain']['buckets']:
        for hit in domain_bucket["top_results"]["hits"]["hits"] :
            results.append((format_result(hit["_source"],
            hit.get("highlight", None)),hit["_score"]))

    logging.info("Before Sort Results :" + str(results))
    results = [result[0] for result in -
    sorted(results, key=lambda result: result[1], reverse=True)]
    logging.info("After Sort Results :" + str(results))

    total = len(results)
    results = results[start:start+hits]
    logging.info("Total results : "+ str(total))

    return jsonify(total=total, results=results)
```

Listing A.4: Client side code for search engine - index.py (Client)[31]

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
Client facing flask application that receives search requests from the user and
 forwards the requests to the back-end server for processing. The results from
 the back-end are received and displayed on the browser.
"""


__author__ = "Anthony Sigogne"
__copyright__ = "Copyright 2017, Byprog"
__email__ = "anthony@byprog.com"
__license__ = "MIT"
__version__ = "1.0"

import os
import requests
from urllib import parse
from flask import Flask, request, jsonify, render_template

#Initialize flask app and load environment variables
app = Flask(__name__)
host = os.getenv("HOST")
port = os.getenv("PORT")


"""
End point for search requests. Receives search queries and forwards it to
back-end.
Method          : HTTP GET
Request Parameters : query - The search query
                     hits  - The number of results to be returned
                     start - Start number for the hits (for pagination purpose)
"""
@app.route("/", methods=['GET'])
def search():
```

```
    #Parse and analyze HTTP GET request.
    query = request.args.get("query", None)
    start = request.args.get("start", 0, type=int)
    hits = request.args.get("hits", 10, type=int)
    if start < 0 or hits < 0 :
        return "Error, start or hits cannot be negative numbers"

    #If valid query exists, create a request and forward to back-end server
    if query :
        try :
            r = requests.post('http://%s:%s/search'%(host, port), data = {
                'query':query,
                'hits':hits,
                'start':start
            })
        except :
            return "Error, check your installation"

        #Get response data and compute range of results pages
        data = r.json()
        i = int(start/hits)
        maxi = 1+int(data["total"]/hits)
        range_pages = range(i-5,i+5 if i+5 < maxi else maxi) if i >= 6
        else range(0,maxi if maxi < 10 else 10)

        #Display the list of relevant results
        return render_template('spatial/index.html', query=query,
            response_time=r.elapsed.total_seconds(),
            total=data["total"],
            hits=hits,
            start=start,
            range_pages=range_pages,
            results=data["results"],
            page=i,
            maxpage=maxi-1)

    #Return to homepage (no query)
    return render_template('spatial/index.html')



#Jinja Custom filters for presentation#

@app.template_filter('truncate_title')
def truncate_title(title):
    """
    Truncate title to fit in result format.
    """
    return title if len(title) <= 70 else title[:70]+"..."

@app.template_filter('truncate_description')
def truncate_description(description):
    """
    Truncate description to fit in result format.
    """
    if len(description) <= 160 :
        return description

    cut_desc = ""
    character_counter = 0
    for i, letter in enumerate(description) :
        character_counter += 1
        if character_counter > 160 :
            if letter == ' ' :
                return cut_desc+"..."
            else :
                return cut_desc.rsplit(' ',1)[0]+"..."
        cut_desc += description[i]
```

```
    return cut_desc

@app.template_filter('truncate_url')
def truncate_url(url):
    """
    Truncate url to fit in result format.
    """
    url = parse.unquote(url)
    if len(url) <= 60 :
        return url
    url = url[:-1] if url.endswith("/") else url
    url = url.split("//",1)[1].split("/")
    url = "%s/.../%s"%(url[0],url[-1])
    return url[:60]+"..." if len(url) > 60 else url
```

A.2 Omitted code

The following listings represent code snippets present in the actual reference ([30], [31]) that were removed from the code base for the experiment because they were either deprecated or irrelevant to the use case.

Listing A.5: Omissions from index.py (Client)[31]

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

__author__ = "Anthony Sigogne"
__copyright__ = "Copyright 2017, Byprog"
__email__ = "anthony@byprog.com"
__license__ = "MIT"
__version__ = "1.0"


"""
The following function of referencing a website using the url and author email is
irrelevant to the search engine use case and is ommited from the working code.
URL : /reference
    Request the referencing of a website.
    Method : POST
    Form data :
        - url : url to website
        - email : contact email
"""
@app.route("/reference", methods=['POST'])
def reference():

    # POST data
    data = dict((key, request.form.get(key)) for key in request.form.keys())
    if not data.get("url", False) or not data.get("email", False) :
        return "Vous n'avez pas renseigne l'URL ou votre email."

    # query search engine
    try :
        r = requests.post('http://%s:%s/reference'%(host, port), data = {
            'url':data["url"],
            'email':data["email"]
        })
    except :
        return "Une erreur s'est produite, veuillez reessayer ulterieurement"
```

```
    return "Votre demande a bien ete prise en compte et \
    sera traitee dans les meilleurs delais."
```

Listing A.6: Omissions from index.py (Server)[30]

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

__author__ = "Anthony Sigogne"
__copyright__ = "Copyright 2017, Byprog"
__email__ = "anthony@byprog.com"
__license__ = "MIT"
__version__ = "1.0"

import re
import os
import url
import crawler
import requests
import json
import query
from flask import Flask, request, jsonify
from elasticsearch_dsl.connections import connections
from elasticsearch_dsl import Index, Search, Mapping
from language import languages
from redis import Redis
from rq import Queue
from rq.decorators import job
from scrapy.crawler import CrawlerProcess
from urllib.parse import urlparse
from datetime import datetime

"""
connections module from the elastic_dsl.connections is a deprecated library used
to create client connections to the Elasticsearch database. It is replaced with
the new elasticsearch library
"""
client = connections.create_connection(hosts=hosts, http_auth=http_auth, port=port)

"""
The native library to create Redis task queues is old and replaced with the newer
library that is suited for flask applications called flask_rq2
"""
redis_conn = Redis(os.getenv("REDIS_HOST", "redis"), os.getenv("REDIS_PORT", 6379))


"""
The following snippet creates multiple indices in the elasticsearch DB for different
languages. This use case not relevant to the experiment that uses text in English
language only and hence is removed
"""
for lang in ["fr"] : #languages :
    # index named "web-<language code>"
    index = Index('web-%s'%lang)
    if not index.exists() :
        index.create()

"""
The following index mapping or database schema is applicable only to the original
reference. The experiment requires different types of data to be stored in the DB
and hence the original mapping is replaced with a custom mapping that is specific
to the experiment
"""
  m = Mapping('page')
    m.field('url', 'keyword')
    m.field('domain', 'keyword')
    m.field('title', 'text', analyzer=languages[lang])
```

```
    m.field('description', 'text', analyzer=languages[lang])
    m.field('body', 'text', analyzer=languages[lang])
    m.field('weight', 'long')
    #m.field('thumbnail', 'binary')
    #m.field('keywords', 'completion') # -- TEST -- #
    m.save('web-%s'%lang)

# index for misc mappings
index = Index('web')
if not index.exists() :
    index.create()

# mapping of domain
m = Mapping('domain')
m.field('homepage', 'keyword')
m.field('domain', 'keyword')
m.field('email', 'keyword')
m.field('last_crawl', 'date')
#m.field('keywords', 'text', analyzer=languages[lang])
m.save('web')



"""
The index function is used in the original reference code to add a particular
web page  into an index one at a time. Since, the experiment explores multiple
web pages at the same time parallely using the /explore API, this API is useless
and is removed.
    URL : /index
    Index a new URL in search engine.
    Method : POST
    Form data :
        - url : the url to index [string, required]
    Return a success message.

"""
@app.route("/index", methods=['POST'])
def index():

    # get POST data
    data = dict((key, request.form.get(key)) for key in request.form.keys())
    if "url" not in data :
        raise InvalidUsage('No url specified in POST data')

    # launch exploration job
    index_job.delay(data["url"])

    return "Indexing started"

@job('default', connection=redis_conn)
def index_job(link) :
    print("index page : %s"%link)

    try :
        link = url.crawl(link).url
    except :
        return 0


    process = CrawlerProcess({
        'USER_AGENT': "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 \
        (KHTML, like Gecko) Chrome/55.0.2883.75 Safari/537.36",
        'DOWNLOAD_TIMEOUT':100,
        'REDIRECT_ENABLED':False,
        'SPIDER_MIDDLEWARES' : {
            'scrapy.spidermiddlewares.httperror.HttpErrorMiddleware':True
        }
    })
```

```
    process.crawl(crawler.SingleSpider, start_urls=[link,], es_client=client,
     redis_conn=redis_conn)
    process.start() # block until finished


"""
The following function of referencing a website using the url and author email is
irrelevant to the search engine use case and is ommited from the working code.
 URL : /reference
    Request the referencing of a website.
    Method : POST
    Form data :
        - url : url to website
        - email : contact email
"""
@app.route("/reference", methods=['POST'])
def reference():
    # get POST data
    data = dict((key, request.form.get(key)) for key in request.form.keys())
    if "url" not in data or "email" not in data :
        raise InvalidUsage('No url or email specified in POST data')

    # launch reference job
    reference_job.delay(data["url"], data["email"])

    return "Referencing started"

@job('default', connection=redis_conn)
def reference_job(link, email) :
    print("referencing page %s with email %s"%(link,email))

    try :
        link = url.crawl(link).url
    except :
        return 0

    # create or update domain data
    domain = url.domain(link)
    res = client.index(index="web", doc_type='domain', id=domain, body={
        "homepage":link,
        "domain":domain,
        "email":email
    })

    return 1
```