Big Data Processing

Implementation of a large-scale search engine

Group B: Adil Choudhury, Muhammad Nabil Fadhiya, Mohammed Owais Khan, Mohanad Al Sayegh

15/12/2017

Contents

Search Engines and Algorithms

Search engines are programs that index and search documents and files for specified search terms. The engine determine how relevant the documents and file are to the search terms by using ranking functions, returning the most relevant documents and files as a result. The next section will talk about some of the methods used by engines to determine how relevant a document or file is.

True or False

At first search engines dealt only with Boolean operators. Either the document contained the search terms or not. This posed a number of problems; firstly you would have to consider how to tokenise words like “Mile End”, either as “mile” and “end, or “mile-end”. Secondly, when you searched for multiple terms you would always end up with either too many results or barely any, and of the results you did get there would be nothing to tell you which pages would be the most relevant.

Page Rank

An algorithm famously used by Google to measure the importance of a page for a search term. It does this by counting the number of links to a page to calculate a value describing its importance. The assumption that page rank has is that the most important pages will have the largest quantity of links pointing to them.

Term Frequency

This method considers the number of times a search term appears in a document to determine which document is more relevant. If a document contained a search term “chicken” a total of 8 times it would be treated as more relevant than a document which contained it 7 times.

An issue with this is if the search terms contained terms like ‘the’, then the document which contained it the most would be scored as most relevant. But the term 'the' would have a high number of occurrences due to being one of the most commonly used words in the English language, which would make it the commanding factor when determining term frequency. The ranking would very quickly devolve into what documents contain the most mentions of the most common words in the language making the other search terms irrelevant.

To deal with this issue inverse document frequency is used. It is a method to weight words dependant on how many times they appear in the document. The more the word appears the lower its weight and the less it will skew the ranking function.

Vector Model

An issue of term frequency is that the longer the document is the higher the weight search terms could have. A document can contain more content than another, without being more relevant. This is solved by calculating the cosine similarity between the documents.

Spark

Our chosen platform for the search engine project was Spark. Spark has a reusable data structure called Resilient Distributed Dataset (RDD). RDD’s are an immutable collection of elements that spark can partition so each node can operate on it in parallel. The main reason why we used Apache Spark was due to the multiple transformations and operations we can apply to each RDD. Due to TF-IDF requiring multiple transformations and operation, there would be a greater overhead of code in Apache Hadoop. Apache Spark allows us to pipe transformations and operations on each RDD to compute and reduce each dataset easily in a smaller code base.

Implementation: TF-IDF

For this project our group will be implementing tf-idf, a frequency based text analysis algorithm and short for term frequency-inverse document frequency. It is a numerical value intended to reflect how important a term is to a document. Using this value to assign weight to a term in a document we can determine how relevant a document is to compute the top 10 results for a search term.

The data set being used is the Stack Overflow Data set from 2017. This can be found in QMUL's HDFS at `/data/stackOverflow2017`. The data is in the XML format which an example of can be found below:

```
<row Id="9" PostTypeId="1" AcceptedAnswerId="1404" CreationDate="2008-07-31T23:40:59.743" Score="1546" ViewCount="399006"
Body="&lt;p&gt;Given a &lt;code&gt;DateTime&lt;/code&gt; representing a person's birthday, how do I calculate their age in years? &lt;/p&gt;&#xA;"
OwnerUserId="1" LastEditorUserId="6025198" LastEditorDisplayName="Rich B" LastEditDate="2017-05-10T14:44:11.947"
LastActivityDate="2017-08-01T00:04:18.453" Title="Calculate age in C#" Tags="&lt;c#&gt;&lt;.net&gt;&lt;datetime&gt;"
AnswerCount="60" CommentCount="8" FavoriteCount="344" CommunityOwnedDate="2011-08-16T19:40:43.080" />
```

Parsing XML Data

Originally, we were given a sample Java code to parse XML into a map data structure but since we were using Scala to develop our search engine, we decided to convert it into similar Scala code. The code is provided below, note that .broadcast is used to ensure the total number of posts is cached in all partitions for efficiency. After we create all the posts we filter out any posts that do not contain the search term for lower overhead:

```
// Parse XML posts as Post Objects
var posts = sc.textFile(data_loc).map(row => new Post(row)).filter(_.getMap() != null).persist(MEMORY_AND_DISK)
val posts_count = sc.broadcast(posts.count().toDouble)
var posts_query_filtered = posts.filter(eachPost => !(eachPost.getWordsFromBody().filter(word => query_asHashSet.value.contains(word)).isEmpty) )
posts.unpersist()
```

A row from the XML file is parsed into a post object, which is added to an RDD which stores all the posts from the dataset. This is the code of the .getMap() function in Post class:

```
def getMap() : Map[String,String] = {
    return postMap
}
```

The map being returned, created by the code below, is a map where a key is the xml element and the value is the content within it i.e. (k,v) → (id,9),(OwnerUserId,1),(PostTypeId,1).

```
private def transformIntoMap() : Map[String, String] = {
    return toBeParsed.split("(=\")|(\"[\\s])|(<[\\w]*)|(/>)").map(_.trim).filter(_.nonEmpty).grouped(2).collect { case Array(k, v) => k -> v }.toMap
}
```

The next section details how we cleaned the data from the dateset.

Data Cleansing

Since we only wanted to index posts and more specifically posts with a 'Body' element we parsed the data through custom methods before we transformed a row into a map.

First we had to check whether the data contained any irrelevant lines that would not be a post to include in our index. In posts.xml the first line would be `<?xml version="1.0" encoding="utf-8"?>` while the second and last lines would be <posts> and </posts> respectively. The code created to ignore these lines is provided below:

```
private def isHeaderOrFooter() : Boolean = {
    return (toBeParsed.contains("<?xml version=\"1.0\" encoding=\"utf-8\"?>") || toBeParsed.endsWith("posts>"))
}
```

We also had to sanitise the contents of the body element as it contained a lot of html tags within it for web page rendering that would be of no use to the user. For example, if the data was not sanitised then &lt;p&gt;Given and the word given would be considered two different words but its representation on a page would both be "Given" which would be useful for a user. The code created to remove html tags is provided below:

```
private def extractWordsFromBody() : Array[String] = {
        if (getBody() == null) return null else return getBody().toLowerCase
        .replaceAll("&lt;code&gt;", "")
        .replaceAll("(&[\\S]*;)|(&lt;[\\S]*&gt;)", " ")
        .replaceAll("[\\s](a href)|(rel)[\\s]", " ")
        .replaceAll("(?!([\\w]*'[\\w]))([\\W_\\s\\d])+"," ")
        .split(" ").filter(_.nonEmpty)
}
```

We also use the same filtering for the query that is provided by the user

```
val filtered_query = query_string.toLowerCase.replaceAll("&lt;code&gt;", "")
                          .replaceAll("(&[\\S]*;)|(&lt;[\\S]*&gt;)", " ")
                          .replaceAll("[\\s](a href)|(rel)[\\s]", " ")
                          .replaceAll("(?!([\\w]*'[\\w]))([\\W_\\s\\d])+"," ").split(" ").filter(_.nonEmpty)
```

A final step was that we ensured that every post had a unique id so it was suitable to use a key in our RDDs.

Extra Utilities

To aid us in our implementation we created methods to give us easy access to desirable components of the data.

A method to retrieve the id of a post to use as a document id for indexing:

```
def getId() : Int = {
    if (getMap() == null) return -1 else return postMap.get("Id").getOrElse("-1").toInt
}
```

A method to retrieve the sanitised content of the body element which contains the terms for indexing:

```
def getWordsFromBody() : Array[String] = {
    if (wordsInBody == null) return null else return wordsInBody
}
```

A method to retrieve the length of the post:

```
def getNumberOfWordsInPost() : Int = {
    return getWordsFromBody().length
}
```

TF-IDF Components

The formula to calculate tf-idf is as follows, tf-idf = " ((Number of times term t appears in a post) / (Total number of terms in the post)) * log_e(Total number of post/ Number of post with term t in it) " (Tfidf.com, 2017). There are 4 distinct values that need to be calculated per term, per document.

Calculating Term Frequency

This is the code we created to calculate the term frequency:

```
var tf_set = posts_query_filtered.flatMap(eachPost => eachPost.getWordsFromBody()
                .filter(word => query_asHashSet.value.contains(word))
                .map(word => ((word, eachPost.getId), 1.0/eachPost.getNumberOfWordsInPost))
                .reduceByKey((a,b) => (a+b))
```

Comments accompanying the code provide an explanation but a visualisation of the code is provided below:

Posts		**Step 1**		**Step 2**
Post1 = [word1 word1 word2 word1 word3]	- - -→	(word1,Post1), 0.2	- - -→	(word1,Post1), 0.6
Post2 = [word1 word1 word2 word2]		(word1,Post1), 0.2		(word2,Post1), 0.2
		(word2,Post1), 0.2		(word1,Post2), 0.5
query_asHashset		(word1,Post1), 0.2		(word2,Post2), 0.5
[word1]				
[word2]		(word1,Post2), 0.25		
		(word1,Post2), 0.25		
		(word2,Post2), 0.25		
		(word2,Post2), 0.25		

Word 3 in this scenario is filtered out because the query of "word1 word2" does not contain it thus its tf-idf would not be helpful.

Calculating Inverse Document Frequency

This measures how important a term is by computing weights based on the occurrences of the term in the dataset.

The first component required to calculate inverse document frequency is the total number of posts. This was as simple as counting how many posts had passed our data cleansing checks, of which the value was broadcast to all partitions with the following line of code:

```
val posts_count = sc.broadcast(posts.count().toDouble)
```

This code below is how we calculated the number of times term t appeared in a post, comments accompanying the code provide an explanation:

```
var wordTuple = posts_query_filtered.flatMap(_.getWordsFromBody()
                .filter(word => query_asHashSet.value.contains(word)).distinct)
                .map(word => (word,1)).reduceByKey((a,b) => (a+b))
```

We then calculated the inverse document frequency using the following code; comments accompanying the code provide an explanation:

```
var idf_set = wordTuple.map(eachWordTuple => (eachWordTuple._1,((Math.log(posts_count.value) - Math.log(eachWordTuple._2))/Math.log(Math.E))+1))
```

Math.log functions call the natural logarithm of the value parsed to it which is the equivalent of base e. The variable idf_set contains a key-value pair where the key is a unique word and the value is the calculated idf. The implementation of this is done by taking a key-value pair from the wordTuple RDD, copying the key and calculating the idf as the value by dividing the broadcasted count of posts with the value of the key-value pair taken from wordTuple.

TF-IDF Value

An initial step was to reformat the structure of the variable holding the term frequency. A visualisation has been provided below:

(word1,Post1), 0.6 ➔ word1, (Post1, 0.6)

This was the code created to reformat the variable:

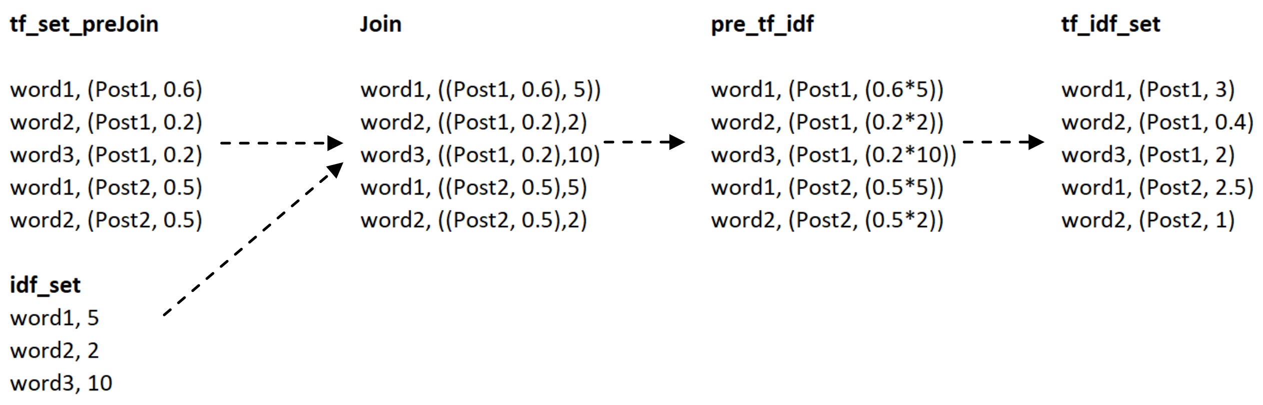
```
var tf_set_preJoin = tf_set.map(tuple => (tuple._1._1, (tuple._1._2, tuple._2)))
```

This gave us two advantages, firstly, it was much easier to iterate over the RDD in this structure as we would only be addressing the value to create our TF-IDF value tuple of (id, calculated tf-idf value). Secondly, it allows us to join the two RDD's on the keys as now both the term frequency RDD and inverse document frequency RDD have a single word as a key.

The following is the code to calculate the TF-IDF value per word, per document:

```
// Generate TF-IDF Set
// Method 1: Resulting Dataset is (word, (ID,TF-IDF)), (word, (ID,TF-IDF)), (word, (ID,TF-IDF)), ..., (word, (ID,TF-IDF))
var tf_idf_set = tf_set_preJoin.join(idf_set).map(pre_tf_idf => (pre_tf_idf._1, (pre_tf_idf._2._1._1, (pre_tf_idf._2._1._2 * pre_tf_idf._2._2)))
```

A visualisation of the line is provided below:



Cosine Similarity and Multi-Term Searches

$$\cos\theta = \frac{\vec{a}\cdot\vec{b}}{\|\vec{a}\|\|\vec{b}\|}$$

Where $\vec{a}$ is the sum of the tf.idf of all the query search terms within the query vector and $\vec{b}$is the sum of the tf.idf of all the query search terms within a document.

This calculation gives us a value representing the similarity between the query and a document. This is repeated for all documents in the dataset, the higher the similarity value the greater the ranking of the document. There are multiple steps we took to calculate cosine similarity.

The first step is to calculate the tf-idf of the query search terms which is done with the code below:

```
// Get TF-IDF for Query
val query_tf = query.map(query_term => (query_term, 1.0/query_size.value)).reduceByKey((a,b) => (a+b))
val query_tf_idf = query_tf.join(idf_set).map(word => (word._1, word._2._1 * word._2._2))
```

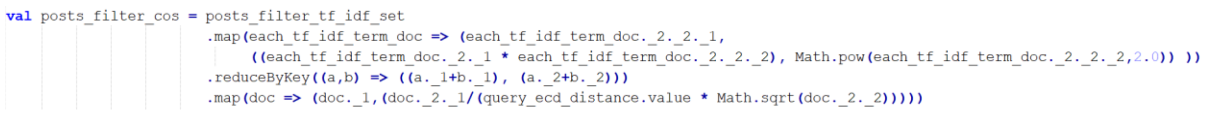
The second step is to calculate the Euclidean distance for the query by squaring all the tf-idf of the queries and summating all the values calculated. This only needs to be done once as it remains constant for the whole calculation, it used in the denominator of the formula. This is the following code to calculate the Euclidean distance; the value is broadcasted to all partitions:

```
val query_ecd_distance = sc.broadcast(Math.sqrt(query_tf_idf.map(eachQuery => Math.pow(eachQuery._2, 2.0)).reduce(_ + _))
```

The third step is to filter the process to calculate the value is the same as the one mentioned above for the tf-idf section.

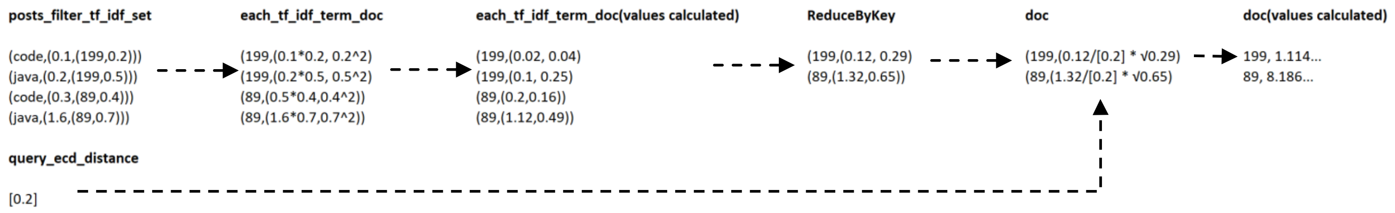
```
//Get Post TF-IDF
//val posts_idf = query_idf
val posts_filter_tf_idf_set = query_tf_idf.join(tf_idf_set_opt_list)
```

The final step is to calculate the cosine similarity of the posts containing any number of the search terms.



```
val posts_filter_cos = posts_filter_tf_idf_set
                        .map(each_tf_idf_term_doc => (each_tf_idf_term_doc._2._2._1,
                            ((each_tf_idf_term_doc._2._1 * each_tf_idf_term_doc._2._2._2), Math.pow(each_tf_idf_term_doc._2._2._2,2.0)) ))
                        .reduceByKey((a,b) => ((a._1+b._1), (a._2+b._2)))
                        .map(doc => (doc._1,(doc._2._1/(query_ecd_distance.value * Math.sqrt(doc._2._2)))))
```

An explanation of the code is provided with the comments accompanying the code by a visualisation is provided below:



The values above are example working values, during actual execution calculated will always fall in-between 0 and 1.

Efficient Implementation consideration

Our initial approach was to store a generated TF-IDF set into the HDFS cluster. However, there was a quota for each user in the HDFS that limited us on saving the generated index even after the quota was increased to 20GB. Our next approach was to change the TF-IDF approach by only calculating the TF-IDF for words that were present in the query. This allowed us to store the results as a fairly small subset of the giant dataset into RAM for futher processing. This also significantly decreased the time it took for our jobs to complete.

Top ten search results

We provide the top ten results by sorting the RDD containing the result into descending order by the value calculated directly after it is generated. This allows us to retrieve the first 10 results as our top search results using the .take method.

The code written for this function:

```
val posts_filter_cos_sort = posts_filter_cos.map(row => (row._2, row)).sortByKey(false).map(row => (row._2)).take(10)
```

Comparison

Single Term Search

We searched for "electron" on multiple web search engines that let you search multiple sites at once and our own. The web search engines that allowed such a function were google, duckduckgo, and stackexchange which the dataset was based on. We listed the top 10 results we retrieved by the id of the post extracted from the url and compared it to ours. In our engine the id of the post is used as our document id. Our project worked on the full dataset that was provided.

Here is the URL of the Spark Job:
http://studoop.eecs.qmul.ac.uk:18088/history/application_1515359599481_0293/jobs/

Top 10 Search result ID's for our selected search engines and project:

Query "**Electron**"

	Project	**Google**	**StackExchange**	**DuckDuckGo**
Post Ids	34433436	41574586	35416172	32621988
Top 10 Results	42103724	42284627	46500302	20187
	39729775	44589278	292872	46035296
	45680424	1128	38067298	43220321
	40506016	44998401	167612	36614776
	45737496	42996881	45811603	35660124
	45680425	43314039	36116638	38172308
	44227258	42342048	39610578	32290967
	34427749	42866775	37802245	38530293
	44277561	70691	32939802	30465034

All search engines gave us various different results. This could be due to the fact that a single search term is not enough to narrow down a very large dataset that is being used by the search engines. One similarity of the search engines is that they all provide very large post id's which show that the posts being returned most likely fall in the 2017 range. Key factors for the differences between the engines could be the information they have access to, i.e. Google has access to my web history and previous searches tied to my Gmail account to give greater context to the query. Another thing that should be considered is that for a single search term we are using tf-idf which is a very basic ranking method with a lot of flaws. The results could be better with multiple search terms due to our ability to use cosine similarity to rank documents. Another point that could be made is that our dataset is a snapshot in time while the other search engines always have access to the latest information, and thus the latest changes to the dataset.

Multi-Term Search

We searched for "nullpointerexception java programming" using the same methodology as the single search. Note that the full dataset was used.

Here is the URL of the Spark Job:
http://studoop.eecs.qmul.ac.uk:8088/proxy/application_1515359599481_0308/stages/

Top 10 Search result ID's for our selected search engines and project:

Query "**Nullpointerexception Java Programming**"

	Project	**Google**	**StackExchange**	**DuckDuckGo**
Post Ids	35186036	47581651	218384	218384
Top 10 Results	19944553	47055568	10464547	21170333
	18236339	47895335	19467202	29754685
	7322250	46958520	1922677	8866741
	22437201	44158940	35020137	6605545
	43341292	46512381	45378590	3988788
	32836104	47368596	3322638	40314944
	31548511	43803097	47751846	11369678
	10157880	43785889	45962690	22074978
	23713081	43785889	15146339	20061654

This was very similar to the single search term where various different results were acquired by the engines. Very large post id's feature prominent with the dataset which shows the search is working as expected on the dataset.

The same discrepencies apply with regards to the snapshot in time of our dataset. A combination technique is used for the ranking with multiple techniques which in theory provide a more accurate result.

Invalid Search Term

We searched for "knekjdwendnweiuhduwehfuhewofhweujhfdoiwejoifdjweoijdoiwejdoiwej-iofjoiwehighiuwrhgfuwehfkwehfk" which we expect to return no results in all search engines. This was true in all cases when used in our project and the previously used search engines.

Here is the URL of the Spark Job:
http://studoop.eecs.qmul.ac.uk:8088/proxy/application_1515359599481_0312/stages/

Evaluation

There are multiple aspects of the project that would be changed if repeated a second time. Firstly, for cosine similarity calculations if the post includes an equal amount of times, each query term, then the value calculated will be 1 which is not correct. Another issue of cosine similarity is that it does not work with one query term as the values calculated will always either be 1 or 0. This is no different from a true or false search which is the most primitive of search engine algorithms.

A limitation of our search engine is that it does not ignore common words i.e. a, I, the, and on. This could affect the final results by ranking longer posts simply because they contain the common words more often.

An improvement could be made to the sorting of the results are right now it sorts by ID in descending order. If a situation arises where there are eleven results at the top, all with the same rank then the top ten results would be chosen based on the ID number and not by its true relevance compared to each other. The eleventh result could be more relevant than the first result which is something our search engine does not take into account.

An improvement we could make is if we took into consideration the zones that query search terms existed in we could have greater context for our calculations allowing us a more context driven similarity. This would be based on the zone indexing search algorithm where greater weight is given to, in our case, different tags of the post i.e. higher weight to a term in the title element than in the body element.

References

Tfidf.com. (2017). *Tf-idf :: A Single-Page Tutorial - Information Retrieval and Text Mining*. [online] Available at: http://www.tfidf.com/ .

While we did not directly use any quotes from it, the bulk of our knowledge on cosine similarity and search engines was provided by the book below:

Widdows, D. (2003). *The geometry of meaning*. 1st ed. Stanford, Calif.: CSLI, pp.151 - 162.

Participation

Each member of the team was actively involved in all aspects of the coursework. This includes research, development and the report.

Extra Objectives

The extra objectives we have completed are, Multi-term searches, combination of multiple ranking strategies to compute results (tf-idf and cosine similarity), and thorough evaluation using comparison with results obtained from search engines.