# Search Engines and Algorithms

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

### True or False

At first search engines dealt only with Boolean operators. Either the document contained the search 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.

### Zone Indexed

This would consider where the search term was placed in a document by giving the elements different weights. For Example, if the search term was found in the title, content, description of a document, give it a weight of 0.3, 0.55, 0.15 respectively. This would allow the search engine to calculate a simple score for each document to determine how relevant they are. The issue with this for giant search engines was that document providers(blogs, websites, marketplaces) would abuse the rankings by placing more terms or irrelevant terms in favourable elements to have a higher ranking calculated. A disadvantage of zone indexing is that based on the number of elements considered, many documents would end up with the same score.

### 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 connectives like ‘and’ which appear multiple times but do not contribute to how relevant a document would be, the document which contained it the most would be scored as more relevant than another term which qualitatively was more relevant.

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.

### 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 and quality of links to a page to a calculate a value describing its importance. The underlying assumption is that the most important pages will have the highest number of links to them.

# Spark

We are implementing this project in spark...

Talk about benefits

# 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.

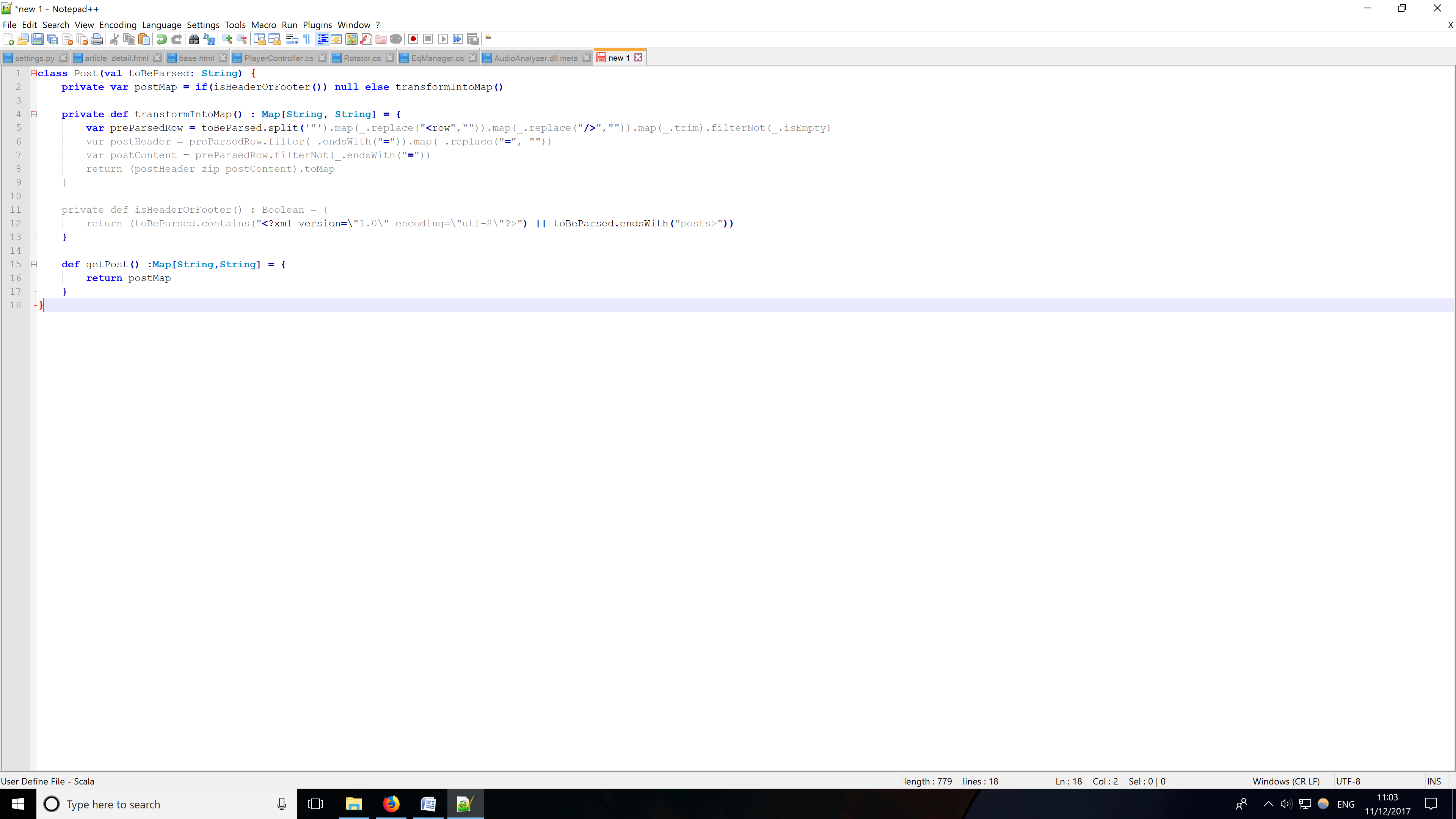
The data set being used is the Stack Overflow Data set from 2017. This can be found in QMUL's HDFS at /data/stackOverflow2017. The data is in the XML format which an example of can be found below:

<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 following code is our implementation of mapping XML into a map data structure:



After we tested that we could extract information required from the map we had to consider whether that data was fit for parsing.

### Data Cleansing

We need to check if all data in the dataset is valid and can be parsed by our code. Ideally, we would like to parse each row into its own class and access each attribute within the class via a map. Thus, the approach we are taking would be to run several checks on each of our dataset.

First we had to check whether the data contained any irrelevant tags as they would not be included in our search. For example, 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.

We also had to sanitise the contents of the body tag 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.

## 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). There are 4 distinct values that need to be calculated per term, per document. The follow is how we tackled that problem;

### Calculating Term Frequency

This measures how frequently a term occurs in a post. Due to the lengths of the post being different, to normalise the results it is divided by the total number of terms in the post.

#### Number of times term t appears in a post

This code below is how we calculated the number of times term t appeared in a post. The code extracts the content within the ‘Body’ tag of a row and splits the content into tokens using a single white space as a delimiter. It then creates a map with a word as a key and add a value ‘1’ everytime the word appeared. The map would then be reduced to summate all the instances of 1 to give us the number of times a word appears in a post.

-------Picture of code

Here is a sample of the values stored in the RDD which contains the number of timers a word appears in one post:

--- sample result rdd.take(5)/rdd.getSample?

#### Total number of terms in a post

This code below shows how was calculated the number of terms in a post. The code extracts the contents of the ‘Body’ tag, splits it with white space as a delimiter and counts how many elements in the RDD. Is it an RDD or a list?

---code

This is a sample of the number of terms in a single post:

---sample result of one post Long = 2096?

### Calculating Inverse Document Frequency

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

#### Total number of post

The code below is how we counted the total number of posts in the dataset that would be processable for our tf-idf calculations we sanitised the dataset using the methods mentioned in the data cleaning section. The code extracted the id of a row to use as a key and added an instance of 1 to the value. This was done to not only check how many rows we could parse in the dataset, but also to see if every instance of a row was unique so that we could use it as a document id in our search engine.

---implementation Code

The results of executing the code gave us the values of Long=37215528(3 Less than number of lines in file due to <xml> <posts> </posts>). Can prove this is correct by stating that fact.

--- Result out of terminal output

#### Number of posts with term t in it

This is the code used to determine the number of posts with the a term in it. The code maps each unique term from the dataset as key then iterates over each post and adds a 1 to the value of the key. The map is then reduced to summate the values giving us the total number of posts with a single term in it.

---implementation code

Here is a sample of unique words and the number of posts that contain them:

---sample results ,take(5)? .getSample?

### TF-IDF Values in our dataset

Using all the values calculated in the above section we created a tf-idf table with the follow structure \*INSERT ROW STRUCTURE\*. To deal with the issue of not have a default logarithm function of base e we used a change of base formula to calculate the idf. The code to calculate the tf-idf is shown below:

---implementation Code

Here is a sample of the results in the tf-idf rdd

---sample results

→ This is how we created our top 10 search

---implementation code

---sample results

Search result comparison \* This comes later if we implement more than tf-idf \*

---Speed to search different terms

--→ a short term

-----Presented as a graph

-----Discuss results

--→ a long term

-----Presented as a graph

-----Discuss results

--→ an illegal term?

-----Presented as a graph

-----Discuss results

Conclusions \* This comes later if we implement more than tf-idf \*

Up to here is 70%

/\* //If we want to get 110% then do this

→ This is how Owais did Cosine Similarity 🡨- Would this not be considered state of the art ranking? State of the art is given a +25, with this also being a combination we’d get +10 mark as well. Even more marks for comparing the results.

---implementation code

---sample results

//If we want to get 100% then do this

→ This is how Mohanad did pagerank

---implementation code

---sample results \*/

(Number of times term t appears in a document) / (Total number of terms in the document).

t\_Appearences→(jack, 7), 6

t\_Appearences(TF) → (jack, 7), 6/body.size

SAVE THIS SEXY SHIT TO A TEXT FILE

idf\_Appearences → jack, 8

idf → jack, plug in the formula(37215531/we gotta calculate)

tfIdf → (jack, 7) ,t\_Appearences.value \* idf(jack)

word,(id, tfidf)

jack, (7, tfidf)

jack, (8, tfidf)