CS F469 IR Assignment – 1

Date : 2nd October, 2017

Topic : Plagiarism Detection

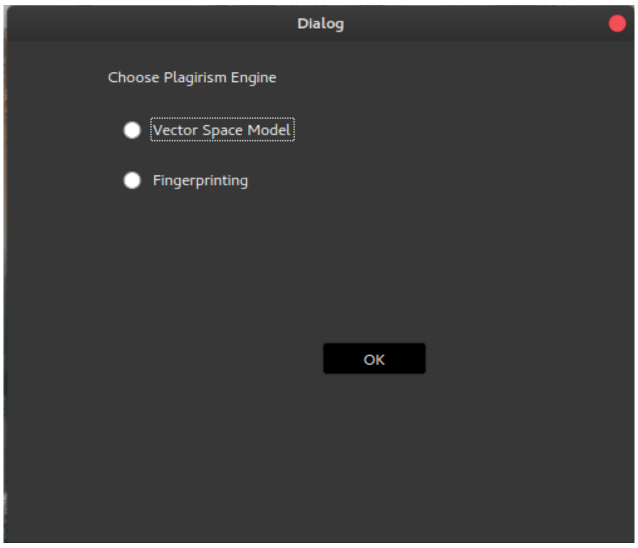
Group Members : PVR Sai Aasrith (2015A7PS0016H)
Rahul Polisetti (2015AAPS0240H)
Hitesh V. Bhagchandani (2015A7PS0023H)

Aim : Given a query document, find the closest documents in the corpus.

Corpus : Wikipedia articles on various Computer Science articles.

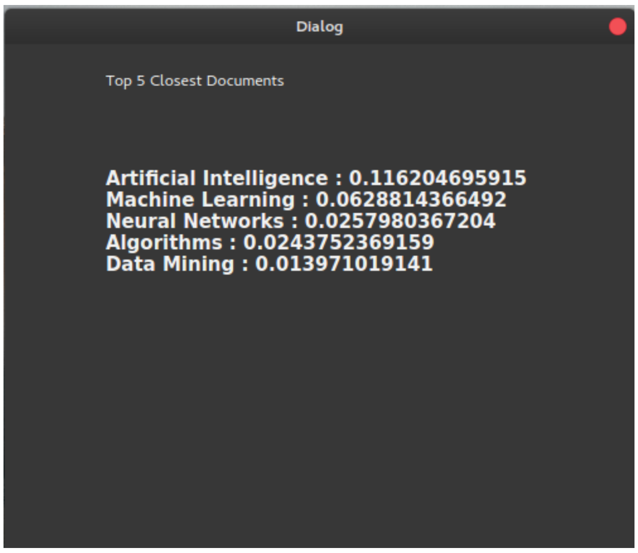
Query : Preferrably, (for reasonable assessment of the model) an article written on Computer Science topic(s).

GUI :



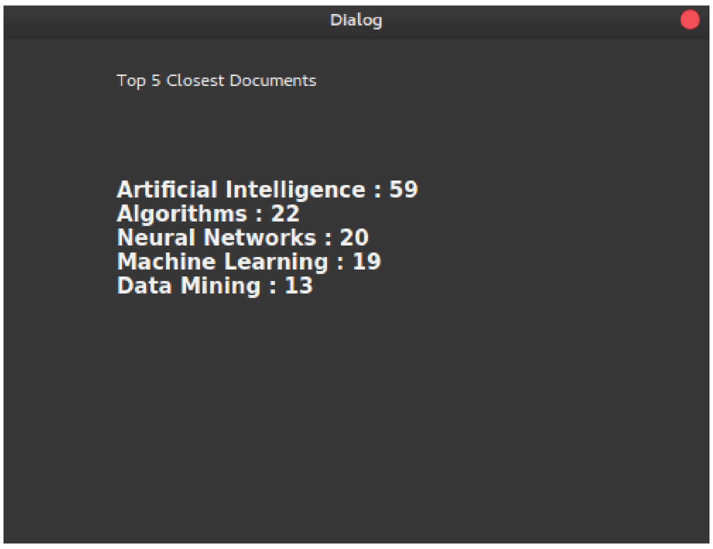
Choose Plagirism Engine

- Vector Space Model
- Fingerprinting



Top 5 Closest Documents

```
Artificial Intelligence : 0.116204695915
Machine Learning : 0.0628814366492
Neural Networks : 0.0257980367204
Algorithms : 0.0243752369159
Data Mining : 0.013971019141
```



Top 5 Closest Documents

```
Artificial Intelligence : 59
Algorithms : 22
Neural Networks : 20
Machine Learning : 19
Data Mining : 13
```

Approach :

The documents are first 'cleaned'. This involves the following, performed in the same order :

1) Tokenisation : Given a character sequence and a defined document unit, tokenization is the task of chopping it up into pieces, called tokens, perhaps at the same time throwing away certain characters, such as punctuation.
The list of punctuation marks chosen are
. , = ? ! ( ) [ ] & @ + " #

2) Stopword removal : certain words like 'the', 'it', 'has' and so on, do not add much context to the document/sentence in hand. It hence, makes it convenient for further processing to remove them.

3) Stemming : Stemmers remove morphological affixes from words, leaving only the word stem. PorterStemmer was used for this purpose. For eg : 'plotted' becomes 'plot', 'sensational' becomes 'sensat' etc.

Two approaches to find document-similarity have been used : Vector-Space model and Fingerprinting for external plagiarism detection.

1) **Vector-model approach** :

First, each document is scanned to get all the terms and their frequencies in that document. The data-structures use to do the same are -

i) maps / dictionaries :
- mp : term-frequencies in each document
- all_terms : inverted index for the terms
- docid : map documents to their ids.
- vectors : stores the document vectors

ii) lists :
As all unique terms could be accessed using the all_terms map, that too in a sorted order using standard sort functions, we decided to store the document frequencies in a simple list rather than a map. Each index corresponds to the terms when sorted in increasing lexicographical order.
- df : stores the document frequencies.

The dictionary : 'vectors' maps each docid to its tf-idf vector (implemented as a simple list). These lists also make it easy to perform dot-products and additions using the 'numpy' module.

2) **Fingerprinting** :

(Please find attached the PDF named 'MOSS paper.pdf' which we referred to, for a more elaborate discussion of the topic).

Each document is cleaned initially using the 'preproc' function. The document is taken and the content is split into k-grams. Now, each k-gram is subjected to hashing and an integer is resulted which is the 'hash_value' of this k-gram. Since all k-grams can be enumerated by a 'sliding-window' approach, hash_value of only the first k-gram is calculated. Now the window is slid by one character and subsequent hash_values are calculated for the windows using dynamic programming.
If the length of the document string was 'x' characters long, we get about 'x-k' hash values. Thsi technique resembles closely, the Rabin Karp rolling Hash function.
Certainly, if each hash_value is stored as a 32/64 bit number, the document hash size would be larger than the document ! (assuming each character occupies only one byte). So the hash_values that would represent the document are found using a technique known as 'winnowing'.
Under this algorithm, windows of hash_values are considered (window size can be varied.). In each window select the minimum hash value. If there is more than one hash with the minimum value, select the rightmost occurrence. Now save all selected hashes as the fingerprints of the document.
Consider a database of fingerprints (obtained from k-grams) generated by winnowing documents with window size w. Now, query documents can be fingerprinted using a different window size. Let $\mathbf{F_w}$ be the set of fingerprints chosen for a document by winnowing with window size $\mathbf{w}'$. The advantage of winnowing query documents with a window size $\mathbf{w}' \geq \mathbf{w}$ is that $\mathbf{F_w}' \subseteq \mathbf{F_w}$, which means fewer memory or disk accesses to look up fingerprints.

NOTE : here, we used window = 10 for the query document and window = 5 for the corpus documents.

For the similarity calculation:
The first step builds an index mapping fingerprints to locations for all documents, much like the inverted index built by search engines mapping words to positions in documents. In the second step, each document is fin-gerprinted a second time and the selected fingerprints are looked up in the index; this gives the list of all matching fingerprints for each document.
Now the list of matching fingerprints for a document d may con-tain fingerprints from many different documents d 1, d2 , ... In the next step, the list of matching fingerprints for each document d is sorted by document and the matches for each pair of docu-ments (d, d1), (d, d2), . . . is formed. Matches between documents are rank-ordered by size (number of fingerprints) and the largest matches are reported to the user.

The data-structures used to implement the above are:

i) maps / dictionaries :
    fingerprints : used to map the document name to its fingerprint-array.

ii) lists :
    hash_val : used to store the selected hash values with each element of the form : [hash-value selected,position in the doc].

Usage of the above seemed reasonable choices because we need each document's fingerprints mapped to it, acessible to other functions in an easy manner; hence the usage of a map.