Information Retrieval: Vertical news search engine and text classification

Jamie Donnelly
MSc Data Science and Computational Intelligence
Faculty of Engineering, Environment and Computing
Coventry University
donnel39@uni.coventry.ac.uk

Abstract – The first objective of this paper is to outline the process of development of a vertical search engine for news articles. Following this, I will explain the steps involved in my text classification task relating to correctly labelling news articles using a naïve Bayes classification algorithm. All programming will be done in using the Python programming language.

Keywords – Search Engine, Information Retrieval, Text Classification, Naïve Bayes.

I. INTRODUCTION

This task is primarily an *information retrieval* task, which can broadly be defined as the process of finding material of an unstructured nature satisfying an informational need from within large collections of data [1]. In my case I am looking to retrieve relevant news items based on user-defined queries. This will involve creating a crawler to amass information from articles and scanning the collection to return the relevant documents. Following this, I will then need to create a data structure to store information retrieved from crawling. This would be done through the use of an *inverted index* implemented using Python's *dictionary* class which I shall explain in greater detail. Once the retrieved data has been collected into an organised data structure it can then be queried by a user following which the relevant information can be retrieved for the user.

The second task I will be looking at is a text classification problem in which I will aim to correctly identify the topic of a news article. This will be done in the fashion of a traditional machine learning task whereby I train a model on a collection of data and then test the efficacy of the model on a separate collection known as the *test data* and user generated queries. For this I will be utilising a type *naïve Bayes classification* algorithm.

II. CRAWLER

The crawler is the first part of my *vertical search engine* as it is responsible for obtaining the large collection of data from which the relevant documents will be obtained via querying. I will be implementing a simple web scraper which will search a collection of approximately 25 *RSS feeds* (These are RSS feeds from BBC News, Sky News and NY Times about a variety of topics). RSS is a type of web feed which allows me to access updates to websites in a standardized, computer-readable format [2]. These RSS feeds will update as more news comes out. The crawler will parse the links to the individual articles from each RSS feed.

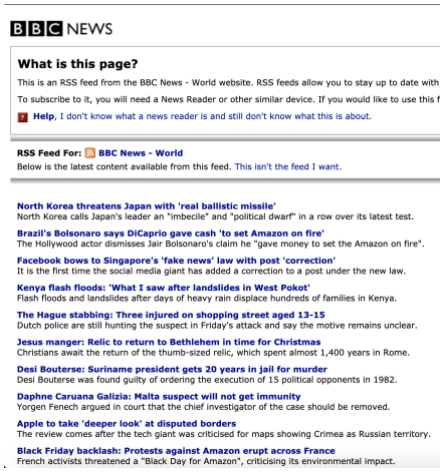

Figure 1: BBC World News RSS example

Figure 2: XML beneath the RSS feed

The above image shows what my RSS feed looks like and the XML code that it's built upon. There is a collection of links to various articles about world news events which will continually be replaced by newer articles. The XML feeds are incredibly easy for a computer to then parse and gather the individual links due to the data structuring, as shown by the tags such as *<link>… </link>*. I can then instruct Python to parse this feed to collect all the links which can then be used for the next step.

Once I have a collection of links obtained, I can then obtain all the *HTML* text for the articles each link represents and parse that to obtain the data I want in much the same way as I parsed the XML feeds due to the structure of the data. For instance, if I want to gather all the text in paragraphs, I can parse the HTML feed for the page and collect all that text contained in the paragraph tags * … *. I can do the same with other elements of each article such as the title and date it was published.

After attempting to loop through all the links and collect the information of interest I then end up with a *pandas dataframe* consisting of the article titles, links and publishing dates. I will also then store the text from the paragraphs of each article, known as the corpus, which will be used later. That is the general outline of how my crawler scrapes the articles on the collection of RSS feeds used. My results (data and corpus) are saved to storage as opposed to just being held on memory and I can then periodically *re-crawl the links* to gather new articles which will be added to my database and corpus and stored in storage, again. Additionally, when re-crawling I will perform a check to see if the articles are already contained in the database to avoid having multiple of the same articles in the database.

III. INDEXER

Having crawled the internet to obtain a corpus of documents, the next task to create a new data structure, an inverted index, which will later be queried. The first step in this process I decided to do was to do some *natural language processing* (NLP) on the retrieved documents. NLP is a vital element of information retrieval and there are many reasons for this. A first example of the importance in using it is when for example searching for a term such as 'cat', we may also want to retrieve documents containing the word 'cats' or vice versa. However, if we do no processing these 2 become distinct entities and search results will only result in one at a time when in reality both are likely to be relevant. An example of how to overcome this and many similar problems is through the use of *stemming*. In the previous example, a simple stemming algorithm would process text such that a plural noun such as 'cats' or 'dogs' just becomes the singular version: 'cat' and 'dog'.

I used a stemming algorithm called the *Porter* stemming algorithm when processing my collection of texts. The algorithm can be generalised as an algorithm used for removing suffixes from words. The idea is that words which share the same 'stem' will have similar meanings [3]. An example stem is:

$$\begin{cases} Connection \\ Connections \\ Connected \\ Connecting \end{cases} \rightarrow Connect$$

A second processing step is done to documents and that is to filter out those words such as prepositions which appear very frequently and may make up a large percentage of a document yet hold little relevance as to the content of the document. These words we will refer to as *stop-words*. I will import a dictionary of these English stop-words from the *nltk* Python library from there I can filter these out of my documents. Some example stop-words are: him, which, had, at, it, you, both. There were179 different stop-words I filtered out of my documents.

Now that I have a processed corpus of documents, I can move on to creating a data structure called an *inverted index* that I will be querying for results. An inverted index is a structure which stores 'content' (in my case words from the documents) and then counts of occurrences of the word in the entire collection and/or the document 'ID' for which the content appears in.

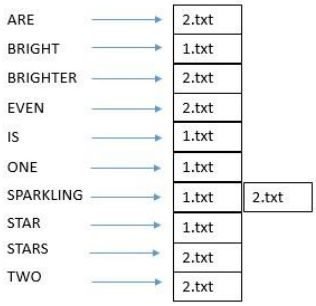

Figure 3: Example Inverted Index [4]

The above example shows the general idea where we have words for 'keys' and then document IDs where those keys appear for values.

I recreate this in Python using the dictionary class in which the words (separated on whitespace) in my entire *processed* corpus serve as the keys and the values are *lists* containing the document indexes in the corpus where said words appear. Using a simple *for loop* I can produce the index in $O(n^2)$ time by looping through each document in the corpus and then for each word in the document and performing a checking whether it is or isn't already in the index, and adding it if necessary. I decided on the python list class for storing the document IDs over something different like creating a linked list because I believe them to be more efficient in this case and avoid the need to traverse each entire list to obtain an ID. Once built I will again be storing this inverted index on my computer's storage for the items to be retrieved later. Along with the database of article information it will get updated when I re-crawl my links for new articles.

IV. QUERY PROCESSOR

The final part of my search engine is the *query processor*. This section deals with the user inputs and determines what data gets retrieved. Query processing represents one of the largest challenges in a task such as this. The reason is because the possibilities of how a user is going to use it (search) are endless. There can be many different ways users may attempt to search for the same thing. Lacking the skill, resources and time of a real search engine such as *google* I aimed for robustness over flexibility when it came to use my search engine. I support some basic *boolean queries* and for users to simply search for a word e.g. 'Brexit'. When simply performing a search for a term like in the previous example, I perform a *ranked retrieval* meaning the order in which results are retrieved are ranked by decreasing relevance.

I. Boolean Querying

Boolean querying involves searching using the boolean operators *AND*, *NOT* and *OR*. These operators are often combined for joint operations such as *AND NOT* or *AND* OR. This allows users to search for a combination of keywords with specified relationships to each other. For example, searching for 'x *AND NOT* y' will return only those results which *include x* but *strictly exclude y*. When I perform Boolean queries there is no element of ranked retrieval, I will return all the results in an indiscriminate order. The reason for this is that documents are either 100% relevant to the query or 0% relevant – they can be treated as binary – for example something either does contain x *AND* y, or it doesn't.

When initiating my search engine, I will firstly prompt the user to ask if they want to perform a Boolean query or not, and then following an answer of 'true' I will then ask them what sort of query:

```
Would you like to make a boolean query? True if yes. Type anything else for a generic term search. True
 '1. x AND y' '2. x AND NOT y' '3. x AND OR y'
Please indicate type of search:
```

Figure 4: Search Engine initialisation

The above image shows how if a user indicates they intend to do a Boolean query it will then ask them which of the 3 basic operations they would like to do.

```
Please indicate type of search: 1

Search terms must be single words.
First search term: Brexit
AND
Second search term: Corbyn
```

Figure 5: Example query

Then this is what would appear if selecting the first option and choosing to perform a *AND* query. This is a complete example Boolean query using my search engine. As previously mentioned, the results will be returned in no particular order but simply those results which are relevant to the query:

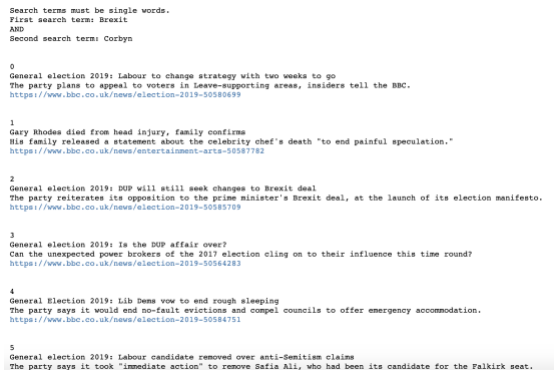

Figure 6: Example results

The Boolean querying roughly works upon simple *set* operations. For example, an 'and' query works by finding the intersection of list postings in the inverted index between 2 search terms. An 'or' query works using the union of list postings for 2 terms. An 'and not' query works by taking the list postings of the first term which are also in the *complement* of the second list postings.

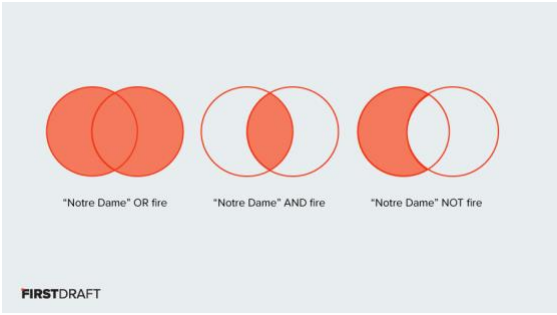

Figure 7: Illustration of boolean query processing [5]

This demonstrates how a Boolean query works and looks from start to finish using my search engine. While the functionality looks restricted, I was always forced to make somewhat of a trade-off between robustness and functionality. Greater flexibility would have resulted in more errors due to limited NLP and error handling when it comes to things like misspelt queries. When the search functions receive an unknown or invalid entry I try and use recursion for error handling where possible – meaning I will return an error message and call the function again at the same time so the user can try again:

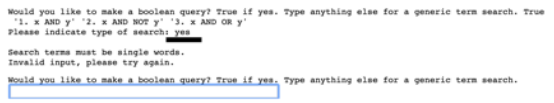

Figure 8: Recursion error handling

As the above image shows, 'yes' is an invalid response and so the query processor begins again.

II. Ranked Retrieval

The other type of search that can be performed using my search engine is just an ordinary search for a single term such as simply searching *'Brexit'*. When performing a search like this the results are returned in order of *decreasing relevance.* Relevance is a subjective term and so I choose to use the *Term Frequency – Inverse Document Frequency* (tf-idf) metric to measure a document's relevancy to the search. The metric aims to reflect how important a word is to a document in a collection or corpus [6]. The simplest way to measure term frequency is to simply take a raw count of the number of terms the search word appears in a document, *i.* As for inverse document frequency, it aims to provide information about the occurrence of the word across the entire collection of documents – does it appear frequently or rarely? How to calculate the metric is as follows:

$$tf\ (raw\ count) = f_{t,d}$$

Where $f_{t,d}$ represents the frequency of t in document d.

$$idf = log\left(\frac{N}{n_t}\right)$$

Where *N* represents the number of documents in the collection and n_t the number of documents which contain the term, *t.*

Combining these we arrive at:

$$tf - idf = f_{t,d} \times log\left(\frac{N}{n_t}\right)$$

Higher scores reflect greater relevance and lower scores the opposite. Therefore, when retrieving results, they are returned in order of highest to lowest tf-idf scores, with the exception of results with a score of 0 as they are not returned at all as deemed irrelevant.

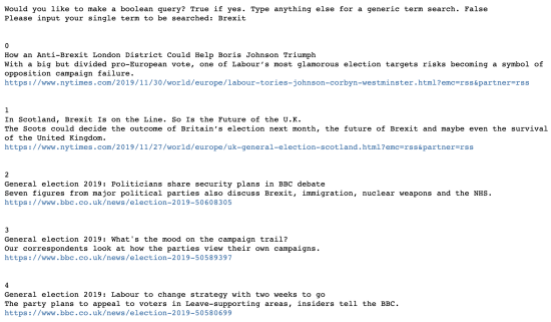

Figure 9: Ranked retrieval example

The above image shows an example search and results obtained using ranked retrieval with the most relevant result being first one.

The process of how these searches work is simple in that I take user input and specifically instruct the user to interact with the search engine in a predefined manner. I do this mainly for the purpose of having the search engine run smoothly and return as few errors/incorrect searches as possible. As previously mentioned, all article text is processed using NLP before being stored in the inverted index and so therefore I must use NLP on the user's queries so they're in the same 'format' as the stored data. The only NLP I use the queries to stem the query terms. Once I have obtained the user's search terms, I can then query my inverted index in storage meaning I return the list of document IDs stored under the dictionary key for the search term. Then once I have the IDs, I can then interact with the data base to obtain the information about the specific articles:

Query → Document IDs → Database → Results

V. TEXT CLASSIFICATION

The second part of my work involved being able to correctly classify text to one of a group of different labels – the labels being categories of news articles e.g. politics or sport. I gathered the data for this in much the same way as I did for my vertical search engine, using a crawler. However, rather than crawling the RSS feeds of various news sites I simply crawled a variety of google news searches with the labels as the search term, e.g. a google news search of 'politics' and then 'sport' and so on. I have five total categories: politics, sport, health, technology, finance and travel. I gathered the text from the articles returned via a google search which resulted in approximately ~100 articles for each topic.

Once I had scraped the text from my articles, I then again created an inverted index where again, the keys of the index (dictionary) were the words in the processed corpus of documents scraped (I used the same NLP as the search engine). However, this time instead of the values for each key being a list of document IDs it was simply a count for how many times the word appeared in the corpus. Unlike the search engine, the inverted index isn't the data structure that I will be querying, it is merely a way to store the collection of words in the corpus which I can then use to create the desired data structure, a *bag of words.* A bag of words is a large matrix-based data structure in which the rows represent documents and the columns represent words in the entire collection of documents:

	it	is	puppy	cat	pen	a	this
it is a puppy	1	1	1	0	0	1	0
it is a kitten	1	1	0	0	0	1	0
it is a cat	1	1	0	1	0	1	0
that is a dog and this is a pen	0	2	0	0	1	2	1
it is a matrix	1	1	0	0	0	1	0

Figure 10: Example Bag of Words [7]

The above image represents an example bag of words. The entries are always either 1 or a 0 – a 1 means the word *is* present in the document and a 0 that it *is not* present. Therefore, the bag of words is made up of a collection of word vectors. I must also add that I performed the same natural language processing to the documents collected for the bag of words as I did previously for the search engine – removing stop-words and stemming the remaining collection of words.

Having turned the corpus produced by my crawler into a bag of words I can then fit a *multinomial naïve bayes classifier* to the data. Naïve bayes classifiers are a group of probabilistic classifier algorithms based on Baye's theorem with independence assumptions between the features [8]. Baye's theorem can be stated simply as:

$$posterior = \frac{prior \times likelihood}{evidence}$$

The multinomial version of the classifier is a supervised learning algorithm that uses feature vectors which show the occurrence of certain events generated by a multinomial distribution. Avoiding going into depth about the derivation

of the formula I have used for classification, the parameter estimator is as follows:

$$\hat{P}(w_i|c_j) = \frac{count(w_i, c_j)}{\sum_{w \in V} count(w, c_j)}$$

This represents the probability of seeing a word, *i*, given the class of documents, *j*. This can be calculated using the fraction of the counts of the word in the class of documents divided by the total number of words in the class of documents, *j*. The algorithm works by maximising this simple probability. Rather than apply this manually in python I can make use of the *scikit-learn* machine learning library which possesses a class, *MultinomialNB*, which will take care of the work for me. The following code snippet shows how a user could simplistically fit and test this classifier algorithm:

```
from sklearn import MultinomialNB
from sklearn.metrics import accuracy_score

clf = MultinomialNB()
clf.fit(X=x_train, y=y_train)
y_predictions = clf.predict(X=x_test)
y_true = y_test
accuracy_score(y_true, y_predictions)
```

Figure 11: Python Multionomial bayes example

In the above, and in my work, the input data *X* would be the word vectors – the rows in my bag of words. The labels *y* would be the prespecified classes of news articles, hence why this is a *supervised* algorithm.

Having now explained the classification algorithm and a bit about how it works I can now easily explain how I converted this into a text classifier. Using python's *input* function to gather a user's input for text they wish to classify. I can then turn this text into a vector of the exact same format of my bag of words – meaning a 1 appears if a word in the collection is present and a 0 where it is absent. I will also perform the same type of NLP to the query as to the words representing the columns of my bag of words. The code snippet below shows how this is done:

```
def text_to_vec(df):
    text = text_processor(input('Please input text to be classified:'))
    tokens = word_tokenize(text)
    doc_vec = []
    for word in df.columns:
        if word in tokens:
            doc_vec.append(1)
        else:
            doc_vec.append(0)
    return np.array(doc_vec).reshape(1,len(df.columns))
```

Figure 12: Transforming a query into a word vector

This shows how I gather a process a user's input and then perform a comparison of the words (*which will run in* $O(n)$ *time*) in the input to those in the bag of words and then return a *NumPy* array which can be passed into the classification algorithm as the *X* input data. Once passed into the classifier it will return one of the 5 different labels of types of articles that I have trained the model based on.

```
text_clas('text_clas_database.csv')
Please input text to be classified: Manchester United vs Liverpool
'sport'
text_clas('text_clas_database.csv')
Please input text to be classified: Exit polls election
'politics'
text_clas('text_clas_database.csv')
Please input text to be classified: Holiday ideas 2020
'travel'
```

Figure 13: Collection of test queries

The above examples show how this works in practice, the function takes a user input e.g. 'Holiday ideas 2020' and will return a label such as 'travel'.

To test the general accuracy of the model rather than the accuracy of individual queries, I can split my bag of words into 2 parts, a training dataset and a test dataset. I can then train the model on the former and test the accuracy by predictions on the latter (comparing the predictions to the actual labels). Doing so I obtained an accuracy of *80.1%* on the test dataset which is a reasonably impressive statistic. The following *confusion matrix* plot allows me to the visualise the test results:

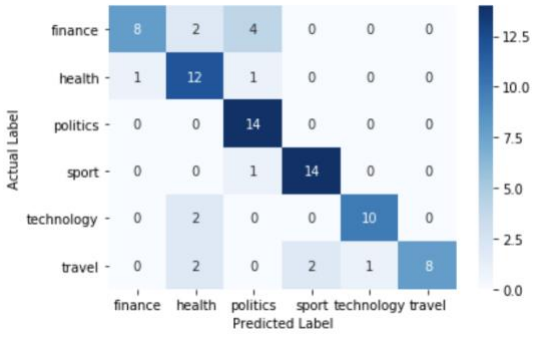

Figure 14: Confusion Matrix of Test Results

This plot shows that the algorithm was very effective classifying politics and sports articles and less effective in classifying finance and travel articles, with finance being mistaken for politics. This has demonstrated both how the text classifier itself works and also how

accurately it is able to predict the correct label either an entire article or a small amount of text in the form of a few keywords.

REFERENCES

1. Manning. C, Raghavan. P. (2008). Introduction to Information Retrieval.
2. En.wikipedia.org. (2019). *RSS*. [online] Available at: https://en.wikipedia.org/wiki/RSS [Accessed 2 Dec. 2019].
3. People.scs.carleton.ca. (2019). [online]. Porter Stemmer. Available at: http://people.scs.carleton.ca/~armyunis/projects/KAPI/porter.pdf [Accessed 2 Dec. 2019].
4. Anon, (2019). [online] Available at: https://codefying.com/2015/12/31/inverted-index-in-c/] [Accessed 2 Dec. 2019].
5. First Draft. (2019). *Boolean basics: How to write a search query for newsgathering that works*. [online] Available at: https://firstdraftnews.org/latest/boolean-basics-how-to-write-a-search-query-for-newsgathering-that-works/ [Accessed 2 Dec. 2019].
6. En.wikipedia.org. (2019). *Tf–idf*. [online] Available at: https://en.wikipedia.org/wiki/Tf-idf [Accessed 2 Dec. 2019].
7. O'Reilly | Safari. (2019). *Feature Engineering for Machine Learning*. [online] Available at: https://www.oreilly.com/library/view/feature-engineering-for/9781491953235/ch04.html [Accessed 2 Dec. 2019].
8. En.wikipedia.org. (2019). *Naive Bayes classifier*. [online] Available at: https://en.wikipedia.org/wiki/Naive_Bayes_classifier#Multinomial_naive_Bayes [Accessed 2 Dec. 2019].

APPENDIX

All code used and datasets can be found at: *https://github.com/jamiedonnelly/News-Search-Engine/tree/master*