Source recommendation system for fact-checking texts

Joy Ghosh, Saurabh Deotale

Introduction

In today's age of online journalism, there are numerous sources for blogs, news articles, and other kinds of information media. Daily millions of articles are being published on the internet via blogs, news sites, etc. Establishing the veracity of these pieces of information and their sources is not an easy task. An everyday consumer of such media has no means of verifying the correctness of the information provided.

The rise of social media platforms has been tremendous but it also presents opportunities for coloring and shaping mindsets and opinions of the general population. There are previous instances of false information being spread to manipulate the outcome of certain events [5]. These events lead to the need for fact-checking published news from different sources. Certain organizations (IFCN) [4] have come forward for doing this very task. These organizations agree upon certain principles [4] for fact-checking published information.

Fact-checking is a manual process, and it takes a significant amount of time and resources. As a result, agencies who provide fact-checked articles cannot cover everything that is being published. There are numerous websites that provide these services, and currently, there are 71 active signatories registered with IFCN [4]. There are other agencies and websites apart from these signatories that also do fact-check on various media such as tweets, statements, published articles, etc. The process of fact-checking involves the study of source materials referenced in the information being inspected. The first challenge in this process is finding these sources. Even though a lot of information has been digitized, there is no central repository common across various agencies. These are often privately held, behind paywalls and an ordinary viewer does not have ready access to these databases.

Another aspect of fact-checking is the time taken for manual verification, by the time any piece of information has been verified manually its impact period might already be over. The research in this area has been extensive. A lot of techniques use machine learning algorithms to establish the veracity of the information [6] [7]. Some platforms also provide services to find fact-checked articles [8] [9]. The major part of popular research in this area focuses on building robust machine learning systems that can provide some form of labeling (mostly true, true, conspiracy, etc).

Major hurdles with this approach are; the bias within the readers' minds, the trust in the fact-checking system, and other systems designed to fool the detection systems [6]. One solution to such problems is to recommend a list of sources, to the users, compiled from already fact-checked articles. We propose a system that can analyze an article and suggest a list of verified articles (along with their labels) from a huge dataset that the user can go through. This system can be used by a general user as well as a person responsible for fact-checking news articles. It can help in reducing the redundant work of fact-checking closely related articles that are already verified.

Problem characterization

The process of fact-checking first requires verifying the sources referenced in the article in question. This task can be looked upon from two perspectives. One, from a general user's perspective where the user needs information about verified sources which in turn helps the user in making an informed judgment. Two, from an inspector's perspective who has to fact-check news articles. As this manual task is time-consuming, it is important to avoid working on duplicate or similar documents. Another important part is finding the correct sources while trying to verify the correctness of new articles.

We propose a system that can recommend news articles that can be identified as sources for the article in question. This system works with a large collection of verified documents/articles from various sources [1]. These documents/articles are already labeled by different fact-checking organizations. We start with an input document for which we are trying to find sources for. We then process the documents in the dataset to find articles that can be sources for different parts of the input document and then recommend a set of articles from the dataset as probable sources along with other meta-data such as their labels, similarity with the input document, etc.

The first challenge is to have a single dataset that consists of a diverse collection of verified articles collected from different sources. It is important to verify articles from trusted as well as controversial sources. This helps in identifying information over the complete spectrum in a non-biased manner. This kind of dataset is compiled by Jeppe Nørregaard, et al [1]. They have compiled the dataset from various sources which contains over 8.5 million articles.

The next challenge is to identify method(s) of processing the content of these articles in an efficient manner with reasonable accuracy. This step is very important as it will decide not only the quality of the output but also the system's performance. There are various methodologies in natural language processing (NLP) that can be used to find similar text in documents, but this task is time-consuming as it involves a lot of computation. And performing this kind of computation on more than 1 million documents at a time, where each document is anywhere between 300 to 2000 words long, is a monumental task.

Once this task is completed we would have a list of documents in ranked order, which are then processed against the input document for a similarity score. It is very crucial to select a strategy for ranking documents that not only can be performed fast enough but can also provide reasonable accuracy.

Dominant approaches to the problem

This problem came to the surface when fake news generated from untrustworthy sources started influencing outcomes of certain events [5]. The most popular approach in tackling this issue is detecting fake news through various machine learning algorithms as well as data mining and feature extraction[12][13].

In the first approach, using a machine-learning algorithm to identify fake news has some challenges [12] [7] where the system is designed to generate text/video [14] to fool this algorithm that they are generated by humans. There is a recent breakthrough that can still overcome this challenge to a certain extent [12], but there is no guarantee that there will not be other ways to fool such systems. Another method is to extract features from large datasets and then create models based on these features that can be used to detect fake-news [13].

One hurdle in both of these approaches is that the result still has to be verified manually, and there are very few resources available to aid the verification process. If the verification step is not completed, there is no way of evaluating how trustworthy the system is. Another challenge is the volume of data that has to be processed, which makes the existing dataset obsolete after some time [3]. This leads to modifying the way metrics are calculated based on which these algorithms can be evaluated.

We are trying to design a system (with reasonable accuracy) that can aid the process of identifying sources for manual verification, and still be relevant for the ever-changing dataset. If we can identify the sources that can be used to verify the veracity of such articles, we reduce the time of discovery. The result of such a system is based on language processing models that can find linguistic and conceptual similarities rather than probabilities.

Methodology

The first step is to pre-process the dataset to extract key-phrases from all the articles (8.5 million in total). The FakeNewsCorpus [3] dataset's size is around 27 gigabytes containing a single CSV. The challenge was to handle that single huge file to extract keywords/phrases in a single iteration. We were unable to finish the pre-processing task on a single large file due to the memory issues for multiple workers. So, we decided to partition that single csv file into 58 chunks of csv files each having a size of around 500 MB. Another issue with pyspark is it only supports stand-alone cluster deployment [20].

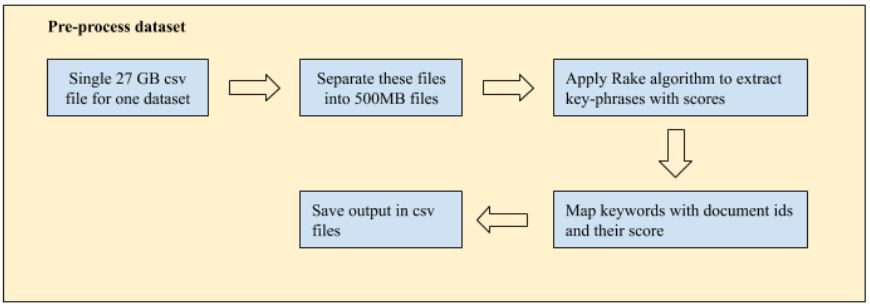

Fig 1. Pre-processing the dataset

We planned to extract keywords from each of the files separately and then combine those keywords together. The code for partitioning csv file can be found in */src/PartitionFakeNewsCorpus.py* file.

The important columns from the csv files are -

- Id
- Type (contains label like unreliable, fake, conspiracy, political, etc.)
- Content (contains the main text from the article)
- Title

For extracting keywords, we tried two approaches

- Lemmatization and chunking combined together [21]
- RAKE (Rapid Automatic Keyword Extraction) [11]

We decided to go with the RAKE algorithm as it doesn't modify the keywords that occur in the document.

We extracted keyword/phrases from 'content', 'meta_description', 'summary' columns using the Rake algorithm. Keywords were also collected from the 'title' column, as it is highly likely that most of the words in the title are important, we included each word in our keywords collection. FakeNewsCorpus also includes 'keywords' and 'meta keywords' column for very few of the articles. We included them in our keyword collections. But all these supplementary columns (except 'content' and 'title') were mostly empty. That's why we primarily depended on keywords/phrases extracted from the 'content' column. Then we selected the top 50% keywords with a high score to save.

We had our own implementation of the Rake algorithm. But it was taking more than 20 minutes for each 512 megabytes of files. So we used a PyPI library named rake-nltk [10] combined with repartitioning the RDD. Doing this we were able to extract keywords from 512 megabytes of csv file in around 1 minute.

We also considered another dataset named 'BuzzFeed-Webis Fake News Corpus 2016' [15]. The zipped file size is around 5.7 gigabytes. But as it turned out this dataset contained a bunch of warc [17] files. Later they processed these files and extracted the articles in articles.zip file, which left us with around 10MB of articles. We used the same keyword extraction approach for this dataset also.

The final output of this step is a csv file where each record contains a key-phrase mapped with all the document ids the phrase was found in along with the score for that phrase for the respective document.

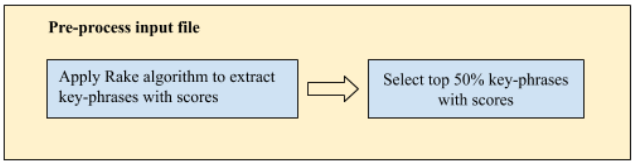

Fig 2. Processing input file to get key-phrases

When a user submits an input text file containing content/main body/text of the article for which we are trying to find sources, the same workflow is followed to extract ranked key-phrases from the input file. We selected the top 50% of these key-phrases for this step as well.

There are two reasons for selecting just the top 50% of the keywords for both dataset and input file:

- If we traverse through the scores calculated for each key-phrases only the top 30%-35% have scores greater than 1, and 1 is the minimum score for any key-phrase identified.
- On an average document having a length of 500 words map to around 130-150 key-phrases, which increases computation time for each document significantly.

The average key-phrase number for the document was calculated by averaging data for 200 documents taken randomly from the dataset. By taking only half of key-phrases identified the runtime was significantly reduced (from 12 min per 500 MB data sliver to around 2 min, the reduction in time also includes further computation done for each document in the dataset).

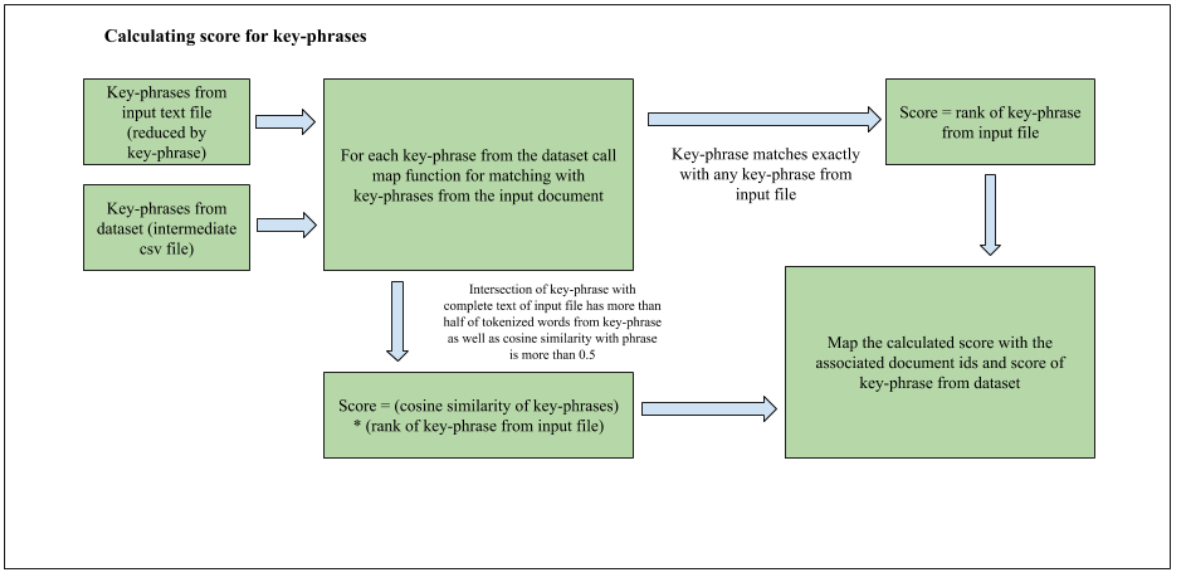

Fig 3. Calculating score for key-phrases

While matching keywords, the computation for each flatmap call is as follows:

- Check if the key-phrase exists as it is in the key-phrase list for the input file
    - Map the score of the key-phrase identified from the input file with the list of document ids and score tuple associated with the key-phrase being processed
- If the key-phrase doesn't match exactly then check the intersection of tokenized words [16] from key-phrase with the words from the input text

- If the intersection has more than half the words in key-phrase then we can start matching phrases from the input file
- For each phrase in the input file, we find cosine similarity [18] between two phrases and if the ratio is greater than 0.5 we calculate the score for it.

$$Score = cosine\ similarity\ ratio * score\ associated\ with\ the\ phrase\ from\ input\ document$$

- We map this score with the document ids and score tuple associated with the key-phrase being processed

The reason for checking the set intersection ratio is to ensure that we have enough similarity between the phrase and the input text before computing cosine similarity. Another trade-off for improving execution time is to make sure that the cosine similarity ratio is greater than 0.5. This ensures that keywords having significant similarity will contribute towards the next step as well as reduce the total computations performed altogether. These threshold checks play a part in reducing the execution time.

The next step is to perform a flatmap and reduce operation to calculate score for each document.

- In the flatmap operation we add the score calculated in the previous step with the score tupled with the document id. Thus generating an output tuple of document id and its score from each row
- In the reduce operation we reduce on document id by adding all scores calculated for that document id. Thus we get a cumulated score associated with each document id.
- Then we extract the top 15 document ids based on the cumulated score.

For calculating the cumulated score we use add between two scores rather than multiplication because the multiplication operation took more time but yielded the same results for all test runs (20 input files in total).

Recommending the source articles based on the cumulated score itself is not good enough. We do an additional step to evaluate the similarity of the top 15 documents with the input text using cosine similarity with Tf-idf vectorizer [19]. This step in itself is not good enough to gauge similarity between two documents/articles, but combined with the previous step we get a good enough result.

After getting row id for expected output articles from the keyword matching phase, we collect type, content, and title from the partitioned csv file using the row id. Then we save these also information along with similarity score in a separate text file for each article following the descending order of the keyword scores.

Experimental Benchmarks

The following spark configuration was used to run the jobs

Number of Workers	30
Driver Memory	2G
Executor Memory	2G
Number of Executors	60
Number of cores per executor	1

Average time for running the jobs are listed below - (test runs based on 20 input files with multiple runs)

Partitioning 27G file into 58 512MB file	32 min
Extracting keywords from 512MB file	1.2 - 1.4 min
Extracting similar articles using keywords from articles of 512MB file	2.3 min
Extracting similar articles using keywords from articles of 1GB file	3.5 min
Extracting similar articles using keywords from articles of 2GB file	6 min

For each 512 MB file we have around 535,000 articles with varying length.

When we start increasing input file size from 2GB the jobs are getting stuck at the 'Shuffle Read phase' in-between stages. At this point, some of the workers stop responding and some start throwing an error stating missing configuration for pyspark. We could not resolve this issue as one of our clusters has stopped accepting any jobs from the command line whereas the same jobs were on another cluster.

We were unsure of how to check the accuracy of the system so we chose 20 documents from the dataset at random and then tried to find articles closely related to them online. The similarity between these documents was varying. So we tried for related articles with very less similar text (word for word) and for next steps we started interlacing the text from the original file and tried to find the threshold where it failed to identify the original document. Following are the results (more about results in conclusion)

% Text similar in input file	**Recommended article containing the original text**
100% text	Yes
50% text	Yes
30% text	Yes
25% text	Yes
<15% text	No

Insights gleaned

Computationally intensive language processing tasks took a lot of time. For finding similar documents we started with comparing all phrases across all documents and the execution time was in hours. Trying to figure out simpler NLP algorithms which do not work really well by themselves and combining them to get good enough results was very important. We had to experiment with the threshold values finding the least possible value for getting results and then buffering them for unseen scenarios helped in reducing the execution time significantly.

Another thing of note was that across datasets there are very fewer articles on the same topics with high similarity, which in hindsight is obvious. This dataset contains verified articles and it is expensive to verify the same articles again, hence we find in recommended articles that one or two articles have more similarities than the rest (in the top 15). This resulted in the large size of output files from the pre-processing step as there is more uniqueness across documents.

We have to be very vigilant in deciding which computations can be skipped if the task is computation-intensive, without having any impact on the quality of output. This can help in improving performance significantly.

How the problem space will look like in the future

To be fair, when one searches something on Search Engines, those search engines perform similar tasks. The search engines take input the keywords from the user and output related articles from the database they have crawled from the internet. Here we are taking the whole document as input from the user. So we had to perform an extra step of extracting keywords from the input text. Moreover, the search engines use keywords from the meta tags of the Html files and headings.

Our algorithm will only work better with the improvement of the keyword extraction algorithm. The Rake algorithm extracts a lot of phrases which does not necessarily contain any important keyword. The only threat to validity to our program is the part where the keyword/phrase extraction takes place. So future work can be to find other algorithms that can outperform Rake in this context.

The results we have achieved are not fully acceptable. Example case: When we are trying to find articles specifically looking for ‘war in Cambodia’, except the initial result which has text similarity other results paint an interesting picture. The other documents recommended had the theme of war-time events or civil unrest related details but not specific to any events in Cambodia. This implies that the algorithm is able to identify the context but not specific events. Improvement in the linguistic and contextual model designed in the system can be improved to identify specific events, which will help in improving the quality of the result

Conclusion

We have an end to end system that can recommend articles having a similar context for an input article, even though the quality of results is not of the desired quality. But the results we have achieved shows that the model used is able to identify conceptual similarity. There is a need to identify a better algorithm for establishing similarities in a better way without increasing execution time.

To some extent, we are able to perform these tasks in a distributed manner. While extracting keywords from 500 MB files, we are repartitioning the file right after reading it from the hdfs. As a result, each of the subsequent operations like map, reduce are happening on the more worker nodes. Also the portion where we are extracting the keywords using the RAKE algorithm, is also happening as a parallel process across all the different workers. Only one part of our algorithm is not happening in a distributed manner. While doing post processing, we have to read the content of top matched articles from the partitioned csv file, calculate similarity score, sort those articles according to their similarity score and save those those files in local hard disk as output to the user. All these tasks are generating 15 text files at max as output. These tasks are being performed by the master node alone. But it seems like it is not really that much of an issue, as the tasks are really simple, doesn't require lots of memory and doesn't take that much time.

But more experiments are needed to check the scalability of the system. We were not able to run our keyword/phrase extraction algorithm from single 27 GB file in single iteration. Also, while matching keywords we were not able to run them for all the keywords from 58 files. To do this we needed more executors to overcome the memory issues.

The system recommends the articles related to the theme of input document but better algorithm can improve the quality as well as scalability.

Bibliography

[1] Norregaard, Jeppe & Horne, Benjamin & Adali, Sibel. (2019). NELA-GT-2018: A Large Multi-Labelled News Dataset for The Study of Misinformation in News Articles. (https://www.aaai.org/ojs/index.php/ICWSM/article/download/3261/3129)
[2] https://zenodo.org/record/1239675#.XazcyfdME5k
[3] https://github.com/several27/FakeNewsCorpus
[4] https://ifcncodeofprinciples.poynter.org/
[5] https://en.wikipedia.org/wiki/Fake_news_website
[6] http://news.mit.edu/2019/better-fact-checking-fake-news-1017
[7] https://reporterslab.org/tag/automated-fact-checking/
[8] https://developers.google.com/search/docs/data-types/factcheck
[9] https://www.facebook.com/help/publisher/182222309230722
[10] https://pypi.org/project/rake-nltk/
[11] Rose, Stuart J., et al. "Rapid automatic keyword extraction for information retrieval and analysis." U.S. Patent No. 8,131,735. 6 Mar. 2012 (https://patentimages.storage.googleapis.com/42/53/56/9e63eb36ee8412/US20110060747A1.pdf
[12] Tal Schuster, Roei Schuster, Darsh J Shah, Regina Barzilay: Are We Safe Yet? The Limitations of Distributional Features for Fake News Detection (https://arxiv.org/pdf/1908.09805.pdf)
[13] Kai Shu, Amy Sliva, Suhang Wan, Jiliang Tang , and Huan Liu: Fake News Detection on Social Media: A Data Mining Perspective(https://dl-acm-org.ezproxy2.library.colostate.edu/citation.cfm?id=3137600)
[14] https://en.wikipedia.org/wiki/Deepfake
[15] https://zenodo.org/record/1239675#.XeqNz3VKgUF
[16] https://en.wikipedia.org/wiki/Lexical_analysis#Tokenization
[17] https://en.wikipedia.org/wiki/Web_ARChive
[18] https://en.wikipedia.org/wiki/Cosine_similarity
[19] nlp.stanford.edu/IR-book/html/htmledition/scoring-term-weighting-and-the-vector-space-model-1.html
[20] https://becominghuman.ai/real-world-python-workloads-on-spark-standalone-clusters-2246346c7040
[21] towardsdatascience.com/natural-language-processing-in-apache-spark-using-nltk-part-2-2-5550b85f3340