COVID-19 Article Clustering and Keyword Search Tool

VAAKESAN SUNDRELINGAM* and WENJUAN QI, University of Waterloo

Currently as part of the COVID-19 Open Research Dataset (CORD-19), there are over 29,000 scholarly articles available to the global research community. In order to aid in research efforts, we have developed a set of search tools which can be used to quickly find relevant articles for scientists, researchers, and the general public. The tools allow the user to search by keyword or article, and allow the user to specify any similarity metric (such as Euclidean, Jaccard, cosine, or any custom metric). The search tool scales linearly with the number of articles in the corpus but suffers from some latency due to large computational and network communication overhead.

ACM Reference Format:
Vaakesan Sundrelingam and Wenjuan Qi. 2020. COVID-19 Article Clustering and Keyword Search Tool. 1, 1 (April 2020), 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The COVID-19 Open Research Dataset Challenge (CORD-19) was issued on March 12, 2020 on Kaggle.com in partnership with The White House Office of Science and Technology Policy, the Allen Institute for AI, the Chan Zuckerberg Initiative, Georgetown University's Center for Security and Emerging Technology, Microsoft Research, and the National Library of Medicine - National Institutes of Health. The data can be found here: https://www.kaggle.com/allen-institute-for-ai/CORD-19-research-challenge. Below is a subset of the tasks that the challenge aims to tackle:

- What is known about transmission, incubation, and environmental stability?
- What do we know about COVID-19 risk factors?
- What do we know about virus genetics, origin, and evolution?
- What do we know about vaccines and therapeutics?
- What has been published about medical care?
- What do we know about non-pharmaceutical interventions?
- What do we know about diagnostics and surveillance?
- What has been published about ethical and social science considerations?
- What has been published about information sharing and inter-sectoral collaboration?

The theme for each problem can be summarized by specific keywords. For example, the task: *What do we know about vaccines and therapeutics?* might require a survey of articles with the keywords *vaccines* and *therapeutics*. To direct research efforts, challenge participants and researchers might need a fast way to search the entire library of articles. There are over 36,000 full text articles in the data set, and searching efficiently will require distributed solutions.

The goal of this project is to create a searching tool to quickly find articles which match the needs of the researcher. There are two approaches that will be considered:

- The user has a topic or theme in mind and wants to search for articles based on some keywords.
- The user has begun with a single article of interest and wants to find similar articles.

The solutions that will be explored are as follows:

- Using the idea of inverted index (discussed in Section 2.2), take a keyword of user's interest and output not only a posting list but also similar documents based some similarity metric (discussed further in Section 2.3).
- Measuring document similarity, either output a posting of similar documents by some measure, or determine a cluster of documents to which the article belongs (discussed in Section 2.1).

2 METHODOLOGY

The dataset contains over 36,000 full text articles in `json` format and a metadata CSV file which contains information about the article's:

`Title, Abstract, Publish date, Authors, Journal, URL, etc.`

For the purposes of the search tool, the title and abstract are sufficient to represent the contents of the article. These strings need to be tokenized and then represented as a set of numerical features in order to perform the necessary algebra on these articles. The tokenizer used was part of the Python `nltk` Natural Language Toolkit Library.

The tokenized data was converted into a numerical vector using the bag-of-words approach. The corpus dictionary was calculated and each document was transformed into a numerical vector where each element represents the count of the corresponding word in the dictionary. The size of the corpus dictionary was 19,577 and the number of articles in the corpus was 29,561. The resulting sparse `title_unigrams_bag_of_words` matrix was $29,561 \cdot 19,577$.

This processing was repeated using bigrams to preserve some of the sentence structure in the titles, which resulted in a matrix of size $29,561 \cdot 149,902$. For development and testing purposes, the bigram matrix was too computationally expensive, and similarly for the unigram and bigram matrices for the abstracts. With additional computational resources these same search tools could be used for larger corpuses (see Section 3.3.2).

Two approaches were tried to reduce the size of the dictionary, both using tools from the Natural Language Toolkit:

- The `stopwords` corpus was used to eliminate common english stopwords such as: *a*, *the*, etc.

*Both authors contributed equally to this research.

Authors' address: Vaakesan Sundrelingam; Wenjuan Qi, , University of Waterloo, 200 University Ave W, Waterloo, Ontario, N2L 3G1.

- The stemming package was tried to reduce words to their root, such as *infect* for *infected*. However this produced unexpected consequences such as removing terminal *s's* on words that would have caused issues in our search tool, and was not used.

Some exploratory analysis was done to understand the corpus. The most common 5 words and their counts are as follows:

[('virus', 5666), ('respiratory', 3321), ('coronavirus', 3196), ('infection', 2661), ('human', 2164)]

And the rarest 5 words with their frequency and the documents they appeared in are as follows:

[('retraction', 1, [(28, 1)]), ('notify', 1, [(41, 1)]), ('wenliang', 1, [(41, 1)]), ('quicker', 1, [(50, 1)]), ('abierto', 1, [(66, 1)])]

2.1 Clustering

To guide the article search, it is useful to cluster or group similar articles together. This allows researchers to quickly find related articles on a single topic. To accomplish the clustering goal, variants of the k-Means algorithm were used. The end product of which is an interactive tool where the user can input a single document and a level of similarity, and receive as output a list of all articles that are within the specified level of similarity to the article of interest. The first approach used to create this tool was a k-Means algorithm.

The k-Means algorithm is a point assignment clustering method. The algorithm works in the following steps:

1. The k centroids are initialized, based on some criteria (Section 2.1.1).
2. Each article is mapped to the closest centroid, based on some similarity measure.
3. The centroids are recalculated by averaging the articles mapped to the same centroid.
4. the process is repeated until some stopping criteria is reached.

The centroid are interpreted by ranking each token, and looking at the top 10 tokens for each centroid.

2.1.1 Initialization Procedure. Performance of the k-Means algorithm is highly dependent on the instance order and initial clustering [Cao et al. 2009]. Therefore the initialization of the centroids is an important aspect of the algorithm to explore. Two initialization procedures were compared:

1. Random selection. In the implementation used, a sample of k articles were used as the initial k centroids.
2. Maximum distance. This implementation involves selecting an initial article at random as the first centroid. The next centroid is chosen to be as far away from the first as possible. The third is chosen to be as far from the previous two centroids as possible. This repeats until k centroids are chosen.

The implementation of the random selection initialization procedure was made in Spark following the skeleton:

Algorithm 1 RandomER Selection

```
$random\_indicies \leftarrow random.sample(range(data.count()), k)$
$centroids = data.zipWithIndex()$
    $.filter(random\_indicies)$
```

The k-Means algorithm was run with two clusters using this initialization procedure. The algorithm completed in 250.94s on average and the centroids selected by this method were summarized as described above by mapping the centroids to their tokens and ranking tokens based on the largest value:

```
[[(0.19168442775466016, 'virus'),
  (0.11235156805034, 'respiratory'),
  (0.10812273757569607, 'coronavirus'),
  (0.09002334314422003, 'infection'),
  (0.07320951317703576, 'human'),
  (0.07063838424845224, 'influenza'),
  (0.06958963429074055, 'viral'),
  (0.06421056192699347, 'protein'),
  (0.056125038059474275, 'disease'),
  (0.05389221556886228, 'syndrome')],
 [(1.0, 'remodeling'),
  (1.0, 'pathological'),
  (1.0, 'loss'),
  (1.0, 'induced'),
  (1.0, 'ii'),
  (1.0, 'dysfunction'),
  (1.0, 'cardiac'),
  (1.0, 'augments'),
  (1.0, 'apelin'),
  (1.0, 'angiotensin')]]
```

We see an issue with this implementation in that one centroid occurs by itself i.e. all other articles belong to the first cluster and the second centroid is a cluster on its own. This suggesets there is some sparsity in the articles which will be discussed later (Section 3.3.2).

The implementation of the maximum distance initialization procedure was made in Spark by the following skeleton:

Algorithm 2 Maximum Distance

```
$centroids \leftarrow articles.takeSample(1)$
**while** $len(centroids) < k$ **do**
    $newCentroids = centroids.append($
    $articles.map(distanceFunction)$
        $.sort().take(1)$
    $)$
**end while**
```

The k-Means algorithm was run with two clusters using this initialization procedure and the centroids were summarized below. The results are as follows:

```
[[(0.19167794316644113, 'virus'),
  (0.11234776725304466, 'respiratory'),
  (0.108085250338295, 'coronavirus'),
  (0.08998646820027063, 'infection'),
  (0.07320703653585926, 'human'),
  (0.07063599458728011, 'influenza'),
  (0.06958728010825439, 'viral'),
  (0.0641745602165088, 'protein'),
  (0.056089309878213804, 'disease'),
  (0.05389039242219215, 'syndrome')],
[(1.0, 'types'),
 (1.0, 'therapy'),
 (1.0, 'suspected'),
 (1.0, 'support'),
 (1.0, 'stock'),
 (1.0, 'spread'),
 (1.0, 'role'),
 (1.0, 'responsibilities'),
 (1.0, 'reliable'),
 (1.0, 'relevant')]]
```

Below are the results with three clusters using a random initialization and only three iterations. The top five tokens in each cluster show that there is some non-obvious patterns in the data emerging:

```
[[(0.12514235654840122, 'coronavirus'),
  (0.11331581252737626, 'respiratory'),
  (0.07503285151116951, 'viral'),
  (0.07459483136224267, 'infection'),
  (0.0745072273324573, 'human')],
 [(1.0, 'health'),
  (0.3678899082568807, 'public'),
  (0.14678899082568808, 'global'),
  (0.11284403669724771, 'care'),
  (0.08256880733944955, 'disease')],
 [(0.9976954440702003, 'virus'),
  (0.16486438574720794, 'infection'),
  (0.14731430597411807, 'influenza'),
  (0.12178691721326006, 'respiratory'),
  (0.11363233469243042, 'protein')]]
```

The above result may be a lucky initialization. In such a high dimensional space with high sparsity, there is a high probability of *singleton clusters* due to the existence of outliers.

2.1.2 Distance Metrics. Part (2) of the algorithm in section 2.1 involves the mapping of each article to the closest centroid, based on some similarity measure. Some of the commonly used distance metrics for document clustering are Euclidean distance, cosine similarity, Jaccard coefficient, Pearson correlation coefficient, and averaged Kullback-Leibler divergence [Zhang et al. 2011]. We explored the first three of these measures, and their implementations are discussed in more detail under section 2.3.

In the MLlib k-Means implementation, there is no option to specify explicitly or supply your own distance function. Our results show that there is a large variation in the final clusters between the three distance metrics that were compared. This variation in performance was also shown between Euclidean, Manhattan and Minkowski distance in previous studies [Singh et al. 2013]. Therefore, the careful choice of distance metric is an important consideration, and was included in our implementation.

2.1.3 k-Means Algorithm. The k-Means algorithm as discussed above has an inherent sequential implementation. Ignoring the initialization step (which was discussed in detail earlier in section 2.1.1), at each iteration articles have to be mapped to a centroid and a new centroid has to be calculated. Both of these operations have a distributed implementation that should improve performance:

(1) At each iteration k centroids are calculated. In Spark, a map transformation can use any desired `distance_function` to determine the closest centroid to each article.
(2) Recalculating the centroid requires grouping all articles belonging to the same cluster and computing the mean. Using the `groupByKey` transformation, bringing all of the articles into the memory of one worker node resulted in memory overflow. Instead, articles were combined pairwise using `reduceByKey`. Articles are summed pairwise along with a running count of the number of articles in that cluster. A final map transformation is required to calculate the mean.

Algorithm 3 k-Means

```
$centroids \leftarrow initialize\_function()$
**while** $i < num\_iterations$ **do**
    $new\_centroids =$
    $articles.map(closest\_centroid, (x[0], 1)$
        $.reduceByKey((x[0] + y[0], x[1] + y[1]))$
        $.map(x[0]/x[1])$
    $centroids \leftarrow new\_centroids$
    )
**end while**
```

2.1.4 Interactive Search. The goal of the research was to develop a searching tool to quickly find articles which match the needs of the researcher. One of the use cases described had a user that has begun with a single article of interest and wants to find similar articles. Given an article title input by the user, there are two search functions:

(1) Return the top `num` most similar articles by the user-specified similarity measure (e.g. cosine, Jaccard, Euclidean, etc).
(2) Return a random sample of `num` articles from all articles which belong to the same "cluster" as the article of interest. The clusters can be trained before-hand and cached in order to speed query time.

In the solution for (1), the user inputted title is processed (converted into a bag-of-words vector) and then the distance between the title and every article in the corpus is computed in a map function. The top `num` results as specified by the user are returned. Below is a sketch of the algorithm:

Algorithm 4 Similar Articles

```
$title\_input \leftarrow user\_input()$
$title \leftarrow bag\_of\_words(tokenize(title\_input) - stop\_words)$
$closest = titles.map(tokenize(x) - stop\_words)$
    $.map(bag\_of\_words)$
    $.map(distance\_function)$
**if** $distance\_function = Jaccard\_sim$ **then**
    **return** $closest.takeOrdered(num, -x[1])$
**else**
    **return** $closest.takeOrdered(num, x[1])$
**end if**
```

Note that at each step of the algorithm, the original titles have to be carried forward so that the output is useful for the user.

The solution for (2) involves processing the user input then performing k-Means clustering on the corpus and determining the cluster to which the title of interest belongs. Finally what is output is a random sample of `num` articles from the same cluster. The sketch of the algorithm follows:

Algorithm 5 Article Cluster

```
$title\_input \leftarrow user\_input()$
$title \leftarrow bag\_of\_words(tokenize(title\_input) - stop\_words)$
$centroids = kMeans()$
$centroid = closest\_centroid(centroids,\ title)$
$cluster = titles.map(tokenize(x) - stop\_words)$
  $.map(bag\_of\_words)$
  $.map(closest\_centroid(centroids,\ x)$
  $.filter(x == centroid)$
$random\_sample = random.sample(cluster.count(), num)$
**return** $cluster.zipWithIndex()$
  $.filter(random\_sample)$
```

In this implementation, the user defines the number of clusters and a distance function. The performance and scalability of the algorithm are discussed later.

2.2 Inverted Index

Inverted index is an index data structure which directs you from a word to location in collection of documents. In our analysis, the type of inverted index we used is record-level type and we are interested in inverted index for unigrams which is adequate in sizes to perform operations. For example, for a particular token i, the format of output is give as:

$$T_i, df_i, [(document_{i1}, tf_{i1}), \cdots, (document_{in}, tf_{in})] \tag{1}$$

T_i is token, df_i is document frequency, tf_{ij} is term frequency of token i in document j. The procedures of inverted index construction is summarized as follows:

(1) Tokenize titles to remove symbols and retain words.
(2) Remove stop words such as 'an', 'it' because stop words are frequent and cannot carry useful meanings.
(3) Remove empty tokens.
(4) Reference each document an index of sequential order.
(5) Count frequency of each token in each document (term frequency).
(6) Move document ID to term frequency as a value pair and token as key.
(7) Group by keys (token) to remove duplicate tokens.
(8) Sort keys based on either ascending order or descending order of document frequency on each token.

We used inverted index idea to construct a function with the purpose to show top n tokens and corresponding posting list. The function allows user to choose between "sort by the most frequent tokens" or "sort by the rarest tokens" and how many tokens they want to show based on their preference.

2.3 Document similarity

The inverted index function will be useful for researchers to have an idea of the summary of a collection of documents by going through the top k tokens. Furthermore, if researchers are interested in a particular keyword, it will act as a map to refer them a list of documents of interest. This idea is implemented using concept of "document similarity". We firstly show all posting lists which contain the keyword, including document IDs and titles. Next, we will let users to specify one document ID of their interest from the posting list, then all documents in the list will form a pair with the particular document. Three types of similarity measures are implemented and the documents which pass the threshold will be given to users, including documents titles and similarity scores. In an exception of small lists of similar documents with length less than 10, threshold will not be used and all documents will be given in the output.

For threshold value of similarity, there is no exact rule to determine it and justify the choice. We decided to calculate different thresholds in different word searchs, so it is not fixed. Threshold is set as α units of standard deviation above the mean. It is formulated as follows:

$$min(sim) = avg(sim) + \alpha * \sigma$$

Where, $min(sim)$ is the threshold similarity score, α is the parameter, and σ is standard deviation. Similarity calculations are based on pairs of the selected document with each of the rest documents. In our analysis, we choose $\alpha = 0.75$. For Euclidean distance, we use -0.75 as the parameter and look for a similarity smaller than threshold, otherwise, we extract the upper area. By assuming similarities follow a Normal Distribution, the threshold value can capture the top 23% similarities.

We consider three commonly used similarity measures in natural language processing: Jaccard Similarity, Cosine Similarity and similarity based on Euclidean distance. We will go through each in detail.

2.3.1 *Euclidean Distance.*

Euclidean distance was used as the baseline distance metric for all algorithms. The formula is simply given by:

$$\sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \cdots + (x_i - y_i)^2}$$

$\forall x_i, y_i$. The issue with the Euclidean distance metric is that articles with smaller titles are given higher similarity scores. These titles limit the number of pairs (x_i, y_i) and therefore minimize the sum. For this reason, we also explore and compare the results from other distance metrics.

2.3.2 *Jaccard similarity.*

Jaccard index is a statistic used to measure how two finite sample sets are similar to each other. Jaccard Similarity is always bounded between 0 and 1. The higher the Jaccard index, the more similar the documents are. The Jaccard index is defined in the following formula:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|} \tag{2}$$

where A and B are two finite sample sets. Jaccard index is size of the intersection divided by the size of the union of the two sets. Specifically, in the search engine context, formula (2) can be simplified as follows:

$$J(doc_i, doc_j) = \frac{p}{p+q}$$

where p is the number of ($doc_i = 1$ and $doc_j = 1$) and q is the number of ($doc_i = 1$ or $doc_j = 1$).

In big data applications, implementation of actual Jaccard Similarity is sometimes replaced by approximated or alternative version called the "MinHash" technique [Wang et al. 2014]. The Jaccard Similarity starts from a bag of words (a list of tokens and occurrences in each document). Regular Jaccard Similarity requires pairwise comparisons across all documents so it needs $N(N-1)$ comparisons assuming N is the number of documents. The updated version of Jaccard Similarity can shorten the length of the bag of words and it just uses one number called the "Signature" to replace it, but requires more procedures than the actual Jaccard Similarity. The idea behind the $min - hash$ function can be listed as following steps:

- Generate a random sample with length of unigrams.
- $RDD_1.map(min{-}hash)$, with $min{-}hash$ function as following process:

(1) Assign key from random sample to each bag of words.
(2) Sort based on key in ascending order.
(3) Use counter to record first occurrence of 1. It is called signature.
(4) End up with an signature $k_{doc_i,\pi}$ for document doc_i for permutation π.
(5) Repeat the $min - hash$ function for permutation $i+1$ until specified permutation n is reached.

As n goes up, approximated signatures will approach actual Jaccard Similarity. In our analysis, we set n=50. To avoid pairwise comparison, we perform transformations on each permutation parallel in Spark. We call this version without pairwise *"Jaccard signature variation 1"*. The idea is summarized as follows:

- $RDD_2 \leftarrow$ Store the output of index $k_{doc_i,\pi}$ for all doc_i and π as key-value pair $(docid, k_{doc_i,\pi})$.Perform transformation on each permutation π :
- $RDD_2.map(f(x))$, given user specified document ID: $DOCID$, with $f(x)$ function as following:

(1) If value of user input ID equals value in $(DOCID, k_{doc_i,\pi})$, add $(docid, 1)$ to list.
(2) $RDD_2.reduceByKey(sum) \leftarrow$ flatmap output list for all permutations.
(3) End up with $(docid, sim = \frac{sum}{n})$, where sim is Jaccard similarity which is calculated as total number of permutations in which the value equals user input document's value.

To compare results and efficiency difference between actual Jaccard Similarity and approximated Jaccard using Signatures, we also implemented actual Jaccard Similarity in a pairwise manner. Specifically, we used bag of words for each document to form a Cartesian pair between doc_i and doc_j. We put all Cartesian pairs into an RDD as an element. Then we perform actual Jaccard Similarity calculations on each element parallel. This is also the same way we used to implement other similarity measures such as "Cosine Similarity" and "Euclidean Similarity". In addition, we applied the approximated Jacard Similarity algorithm in pairwise fashion in order to compare running times. For each pair of two documents, we implemented a Python code to transform a bag of word list into a list of two signatures for doc_i and doc_j and then end up with a list of Boolean value to test equivalence of the two for all permutations. The parallel procedure is performed as above on all pairs of documents. We call this version *"Jaccard signature variation 2"*

2.3.3 *Cosine similarity.*

Cosine Similarity is a gauge to measure how two documents are similar to each other independent of their document sizes.

Cosine Similarity is useful in plagiarism detection. Suppose we have two documents Doc_a and Doc_b but Doc_b is just a smaller portion of Doc_a. Using Euclidean distance between these two documents will not give us a desirable similarity simply because number of common words are not large. However, we expect these two documents to have very high similarity because they share the same contents. Cosine Similarity is a good candidate in this situation because it calculates cosine value of angle between the two vectors. The smaller the anger the higher the similarity. Cosine Similarity is bounded between 0 and 1. A higher cosine value of the angle implies two documents are more similar. The cosine Similarity is formulated as follows [Sitikhu et al. 2019]:

$$cos(\theta) = \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\| \|\vec{b}\|} = \frac{\sum_1^n a_i b_i}{\sqrt{\sum_1^n a_i^2}\sqrt{\sum_1^n b_i^2}} \tag{3}$$

For this approach we used occurrence of each token (bag of words), and then applied Cosine Similarity to calculate similarity score between each pair of documents.

3 EVALUATION

3.1 Keyword Search

We implemented three types of similarities in the keyword search function. Users will be interested in similarities and differences between them in terms of search results and running time. First of all, we found that some articles have same titles but different doi number. For example, 1598 : *'Comparative phyloinformatics of virus genes at micro and macro levels in a distributed computing environment'* appears four times in the dataset with document number 1599, 1560 and 1561. In our similarity result, we can see all three document pairs (1598, 1599), (1598, 1560) and (1598, 1561) are given with identical similarities: Jaccard similarity outputs $1's$ for all the pairs, and cosine similarity is 0.999. This is consistent with what we expect.

To evaluate how Jaccard signature similarity using $min - hash$ approach compares to actual Jaccard similarity using number of permutations 50, we have tried different keywords and document IDs. We fixed keywords so that all documents contained the keyword and we fixed one document ID in the pair, looping through all remaining documents:

keyword	document ID	Mean difference
'canada	508	0.069149
'covid'	5	0.09694
'virus'	1598	0.04150

The mean difference is calculated using the formula:

$$\text{Mean difference} = \sqrt{\frac{\sum_{i=1}^{k}(sim_{1i} - sim_{2i})^2}{k}}$$

where sim_{1i} is actual Jaccard Similarity for document pair i, sim_{2j} is approximated Jaccard Similarity for document pair i, k is total number of document pair containing the keyword. This result demonstrates that using $min - hash$ approach approximated Jaccard Similarity are very close to the actual Jaccard Similarity.

However, in the aspect of run time, run time for actual Jaccard is smallest among the three. Run time between two approximated versions are very interesting: *"Jaccard* signature variation 2" is less than *"Jaccard signature variation 1"* when keyword is common and contained in many articles because version 1 includes pairwise comparison, and calculation in each pair includes sorting and two for loops so run time is $2n + nlog_n$. When users are looking for rare keyword, number of pairs the algorithm needs to loop over are fewer. In this situation, communications over network between different nodes are manifest. Here is an example to illustrate the idea:

similarity	'canada (82)'	'genes' (267)
'Jaccard'	160	268
'Jaccard variation 1'	177	841
'Jaccard variation 2'	220	301
'Cosine'	212	211
'Euclidean'	210	213

Where run times are denoted in seconds. We expect that as document number increases or if keywords are common words, Jaccard variation 2 run time will be lower than the actual Jaccard and Jaccard variation 1.

In the three actual similarity measures, we can see Cosine Similarity can ignore size of two documents and only focus on common tokens between them so sometimes long documents paired with short documents will tend to have small similarity but in other times it is reversed. Euclidean distance of a document pair containing a shorter title actually has smaller distance. Here is an example: [(4.123105625617661, ('Comparative phyloinformatics of virus genes at micro and macro levels in a distributed computing environment', 'Pulmonary function and bronchial reactivity 4 years after the first virus-induced wheezing'))), ('1598-23466', (3.1622776601683795, ('Comparative phyloinformatics of virus genes at micro and macro levels in a distributed computing environment', 'Bat and virus'))] Shorter article is actually more closer to reference document.

Run time for actual Jaccard Similarity, Cosine Similarity and Euclidean Similarity are very close. In one keyword search of "virus" (very common word in articles) for document ID 1598, Jaccard Similarity requires $223s$, Cosine Similarity requires $295s$ and Euclidean Similarity requires $382s$.

3.2 Article Search

Every day there are more articles being added to the corpus as part of the COVID-19 Open Research Dataset Challenge (CORD-19). In order to continue to be useful, the search tools that were built should scale with a larger corpus. In order to evaluate the scalability, both article search tools were run ten times each on three different corpus sizes: 100, 1000, and 29,561. The results for the article search function are summarized in Figure 1:

A linear regression was fit to the observed runtimes. Although the relationship is strong, the most important observation is that the relationship is linear. In other words, the runtime of the algorithm scales linearly with the number of articles.

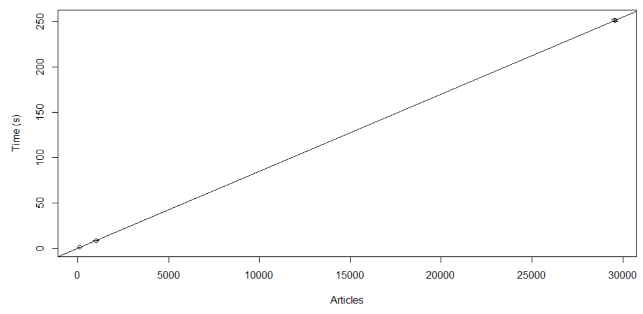

Fig. 1. Article search run 10 times each for various corpus sizes.

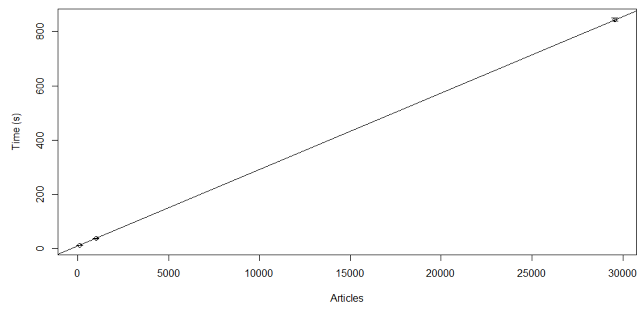

Fig. 2. Article cluster search run 10 times each for various corpus sizes.

In terms of absolute value, the average runtime is still slow (250.94s for the full corpus of 29,561 articles). However, we believe this is partly attributable to the overhead in network communication, and partly attributable to having run the program on just a single worker node.

Figure 2 shows a summary of the performance of the article cluster search algorithm under the same setup.

We note again that there is a strong linear relationship, and also that in terms of absolute value the algorithm is slow (843.38s average runtime for the full corpus). There is additional computational overhead in the article cluster search attributable to having to compute the clusters. However this demonstrates again that the search tool scales linearly with the size of the corpus.

3.3 Future Work

3.3.1 *Generalizability.*

For the keyword search engine, we should generalize the idea into bigrams or n-grams and apply the similarity measure to abstracts or whole articles. In this way, the difference between similarities will be manifest and similar documents found by this search engine will be better supported. With even larger document sizes, we can also assess how Jaccard variation approximates actual Jaccard Similarity score. Furthermore, we could use *term frequency-inverse document frequency* instead of bag of words to give a different weight to each token to mitigate the impact of common words appearing in most of documents.

3.3.2 Outlier Detection. The `randomER initialization` function that was determined to produce the best clusters cannot guarantee convergence [Cao et al. 2009]. Using `smart initialization`, we found that there was a sparsity problem with respect to the data. In such high dimensional space, there is a high probability of creating clusters of one.

The existing k-Means package in the MLlib library from Spark uses `setInitializationMode` as either random initialization or the k-means|| algorithm which is a variant of the k-means++ algorithm [Bahmani et al. 2012]. The k-means++ algorithm selects the initial centroids sequentially based on the total distance within the clusters created by those centroids [Arthur and Vassilvitskii 2006].

The potential issue of the k-Means++ algorithm is that it is possibly sensitive to outliers. The objective function given [Arthur and Vassilvitskii 2006] seeks to select the centroids so as to minimize the total distance between the centroid and all points in the cluster determined by that centroid:

$$\Phi = \sum_{x \in X} \min_{c \in C} ||x - c||^2$$

In order to increase sensitivity to outliers (centroids which will inherently have a low value for Φ), we attempted to change the objective function to include not only the distance between the centroid and all points in the cluster determined by that centroid, but also the total number of points in the cluster:

$$\Psi = \frac{1}{||X|| + 1} \sum_{x \in X} \min_{c \in C} ||x - c||$$

In our formulation, the objective function was inversely proportional to the size of the cluster X generated by centroid c. To avoid the issue of dividing by zero, an adjustment of 1 was made to the size of the cluster X. The objective function is also, like the k-Means++ formulation, directly proportional to the total distance between the centroid and every point in the cluster. Note that this quantity is potentially zero (clusters of one) and so our implementation filtered out these cases. The rough implementation was as follows:

Algorithm 6 smartER Initialization

```
$centroids \leftarrow RDD.takeSample(False, 1)$
**while** $len(centroids) < k$ **do**
    $cartesian =$
    $RDD.cartesian(RDD).map(objective\_function)$
        $.reduceByKey(lambda\ x, y:\ x + y)$
        $.filter(objective\_function\ != 0)$
        $.max()$
    $centroids.append(cartesian)$
**end while**
```

However, this algorithm was unable to terminate in a reasonable amount of time. In future work, we would like to explore this initialization criteria further and build a fast, scalable implementation.

3.3.3 Performance and Scalability. For the article search, the runtime on one worker node was slow. We would like to compare the performance on a cluster with many worker nodes, and look for other opportunities to improve runtime in general for both the article search tools and the keyword search tools.

REFERENCES

David Arthur and Sergei Vassilvitskii. 2006. *k-means++: The advantages of careful seeding.* Technical Report. Stanford.

Bahman Bahmani, Benjamin Moseley, Andrea Vattani, Ravi Kumar, and Sergei Vassilvitskii. 2012. Scalable k-means++. *arXiv preprint arXiv:1203.6402* (2012).

Fuyuan Cao, Jiye Liang, and Guang Jiang. 2009. An initialization method for the K-Means algorithm using neighborhood model. *Computers Mathematics with Applications* 58, 3 (2009), 474 – 483. https://doi.org/10.1016/j.camwa.2009.04.017

Archana Singh, Avantika Yadav, and Ajay Rana. 2013. K-means with Three different Distance Metrics. *International Journal of Computer Applications* 67, 10 (2013).

Pinky Sitikhu, Kritish Pahi, Pujan Thapa, and Subarna Shakya. 2019. A Comparison of Semantic Similarity Methods for Maximum Human Interpretability. (2019).

Jingdong Wang, Hengtao Shen, Jingkuan Song, and Jianqiu Ji. 2014. Hashing for Similarity Search: A Survey. (2014).

Taiping Zhang, Yuan Yan Tang, Bin Fang, and Yong Xiang. 2011. Document clustering in correlation similarity measure space. *IEEE Transactions on Knowledge and Data Engineering* 24, 6 (2011), 1002–1013.