Similar Sentence Detection Using MinHash and RandomProjection on Wikipedia*

Zhixiang Meng
University of Waterloo
Dept. of Computer Science
z9meng@uwaterloo.ca

Mengyun Zheng
University of Waterloo
Dept. of Computer Science
m24zheng@uwaterloo.ca

ABSTRACT

In this report, we replicate a novel similarity detection algorithm to identify near-duplicate sentences in Wikipedia articles based on Weissman's work [12]. This is accomplished with a MapReduce/Spark implementation of minhash and random projection, which are locality sensitive hashing (LSH) techniques, to identify sentences with high Jaccard similarity and low hamming distance respectively. Our experimental results appear to support the conclusion of Weissman's[12] clustering result and part the manual inspection.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: [Information Search and Retrieval]

General Terms

Algorithms, Design, MapReduce, Spark

1. INTRODUCTION

The free Internet encyclopedia Wikipedia is the largest and most popular general reference work on the Internet, that is accessible to everyone to make a contribution to [4]. Benefit from everyone's effort, Wikipedia has multilingual versions among the world and has more than 40 million articles. On the other hand, due to the open environment, people may edit most of the contents by copying information from the other source or even from the other Wikipedia articles without any penalty. As a result, it is not a supervise to notice the phenomena that the passages with high-similarity appear in many articles.

In this course project, we attempt to replicate the result of near-duplicate sentence detection from Weissman etl work [12]. They accomplished the task using minhash to identify the Jaccard similarity on the MapReduce framework. The output of the minhash is then utilized to generate sentence clusters. In order to explore more details of locality-sensitive hashing (LSH) techniques, we also implemented the random projection hash to detect near-duplicate sentences using MapReduce and Spark framework respectively.

The rest of the report is organized as follows: In Section 2, we will review the work have done by Weissman [12] and related background of minhash and random projection, in Section 3, we explain the two LSH hashing techniques,minhash and random projection, attempted to be implemented on MapReduce in detail, in Section 4, we present the two techinique implementations including data sorce, data preprocessing, and generate final clusters, in Section 5, we discuss our experimental results according to final clustering result and running time. The final section contains concluding remarks and future work.

2. BACKGROUND

In order to find near duplicate sentences on Wikipedia, the most important part is to define the measurement of the similarity between documents (sentences in our context). In Weissman's work [12], they used minhash which is one of the well-known common LSH techniques to generate signature and used to measure the document similarity. They calculated a minhash signature on each text document (sentence with a Wikipedia article) from the whole collection using a parameterized family of hash functions F_i. Each sentence in an article is split into n-gram "shingles" to develop multiple shingle sets for a sentence. For each shingle set S, they calculate the minimum hashes over the N hash functions then use a vector of K minhashes as the signature on a document d which is randomly chosen from N minimum hashes.

In the implementation of Weissman'swork[12], they parsed raw Wikipedia page text which are in Wikipedia XML dump format by using utilities from the Cloud9 library [1]. During this process, raw data is transformed into a SequenceFile format which is defined in Cloud9 library. After the Wikipage preprocessing, thy designed that each mapper to receive an article in WikipediaPage format with unique docid. Inside the mapper, they used a regular expression to match sentences and then compute M minhash signatures through implementing hash functions by using a "Multiply Shift" scheme[2]. Then each signature with the sentence id are emitted as the intermediate key-value pair. In the reducer part, each reducer receives a signature as the key and as

*This effort is part of the CS651, Data-Intensive Distributed Computing (Winter 2018), class project. For coding details, we make it an open source `https://github.com/zxmeng/NearDuplicateDetection`

values all sentence ids that share the signature.

Random Projection is another well-known method of LSH techniques to generate signatures to represent document[7]. In random projection, the original high-dimensional data is projected onto a lower-dimensional subspace using a random matrix whose columns have unit lengths. In our project, we will implement this method to encode document hyperplanes as fixed-size bit vectors on MapReduce and Spark framework.

3. METHODOLOGY

In this project, our purpose is to replicate Weissman's work [12], which is to detect similar sentences on Wikipedia. Based on what they had explored, we referred their MapReduce implementation of minhash method to compute the signature and reimplement on Spark framework. Moreover, we explored Random Projection hashing method to compute the signature to detect similar sentences on the same Wikipedia source as in minhash method and implement MapReduce framework and Spark framework respectively. In rest of this section, we will discuss more about the algorithms in detail.

3.1 MinHash

Minhash signature for a document is calculated using N hash functions defined as hash family. Instead of term level, we split a sentence in to shingles (character level) and compute the hash values over the hash family. For each shingle set S we calculate the set $\{min_{s \in S}(F_i(s)\}$ of minimum hashes over the hash family. The signature on a document d is represented as a vector of K minhashes, chosen from the set $\{min_{s \in S}(F_i(s)\}$ of minimum hashes jsut as what Weissman did in their work. The relationship between minhash collision (i.e., documents that share the same signature) and their Jaccard similarities has been proved by Broder in his work[6]. We present the minhash MapReduce implementation pseudocode as in Algorithm 1.

3.2 Random Projection

Random projection signatures are computed based on a series of random vector with normalized length. The series of generated random hyperplane serve as hash functions to represent a document text as binary code vector with fixed size. Usually, the random real-valued vector is generated based on the vocabulary size V of the text collection. In order to obtain a D-bit signature, we need to generate such random vector for D times and then use the generated random vectors to map the document vectors. In current step, we assume that we have converted a raw text document into document vectors already.

After the generation of random vectors, for each document vector u, we can compute a D-bit signature s_u by an inner product of u and i^{th} random vector to determine the i^{th} bit of the signature.

Given D random vector $r_1 ... r_D$, the i^{th} bit of the signature s_u is computed as follows:

$$s_u[i] = h_{r_i}(u) = \begin{matrix} 1, \text{if } r_i \cdot u \geq 0 \\ 0, \text{if } r_i \cdot u < 0 \end{matrix} \quad (1)$$

The cosine similarity between two documents can be computed through hamming distance between their signatures as follows:

$$Sim(u, v) = cos[(\frac{\pi}{D}) hamming(s_u, s_v))] \quad (2)$$

The relationship between random projection hash collision and their cosine similarities has been proved by Goemans and Williamson in their work[9].

In order to detect similar sentences in Wikipedia in MapReduce framework more efficiency, we apply sliding window algorithm after computing D-bit signature of document vector. In map function, we permute the D-bit signature for each document vector for Q times, then emit a pair of permutation number and signature as key and document vector as value. The reduce step is designed so that all keys with the same permutation group number are sent to the same reducer, and they are sorted according to the signature bits. In reduce function, we set up two first-in-first-out queue to keep signatures and document id respectively. For each new bit signature, we compute the hamming distance with all signatures have been stored in queue and emit the distance if it is less than our distance threshold. If the queues are full after adding new bit signature to end of queue, we deque the oldest signature. We present the random projection MapReduce implementation pseudocode as in Algorithm 2. The sliding window algorithm pseudocode part is referred from the Lin's work[11]. We implement on Spark framework based on the same algorithm.

4. IMPLEMENTATION

In general, we implement minhash and random projection to detect similar sentences of dump Wikipedia source on MapReduce and Spark framework respectively. In each framework, we firstly convert the data source from raw XML dump format into plain text. Moreover, we apply word embedding on plain text to convert it into document vector on sentence level before implementing random projection. Finally, we analyze the detection result (dedup-ed pairs) by forming sentence clusters as WeissmanâĂŹs work [12]. The process is summarized in Figure 1.

4.1 Data Source

In this project, we do experiment on two datasets which are both in XML dump format. We firstly do test on part of the full wiki dump file which is obtained from the open source of Weissman's work[12]. We then obtain enwiki-20151201-pages-articles.xml.bz2, a full wiki dump file, from Altiscale environment accessible for CS651 students.

4.2 Data Preprocessing

4.2.1 MinHash

In order to send the plain text to our MapReduce job (and Spark), we need to process the Wikipedia XML dumps. In this step, we use the lintool.WikiClean which is a Java Wikipedia markup to plain text converter and is an open source [3]. With this application, we clean the references, image captions, tables, infoboxes, remove the sections "See also", "Reference", "Further reading", and "External links" and the article title. As a result, we obtain the plain text in Text format and then append document id for each article.

Algorithm 1 Minhash MapReduce Pseudocode

```
function INITIALIZE
    F ← hash family;
    sigLen ← shingle length;
    draw ← number of draw from hash family;
    N ← number of signatures to emit;
end function
function MAP(docid d, text t)
    sentenceCount ← 0
    while s ← nextSentence(t) do
        shingles ← shingleSet(s,sigLen);
        minhashes = new List(|F|);
        for shingle ∈ shingles do
            for i ← 1...|F| do
                minhashes[i] ← min(F_i(shingle), minhashes[i]);
            end for
        end for
        for i ← 1...N do
            sig = select(draw, minhahses);
            emit(sig,(docid,sentenceCount));
        end for
        sentenceCount++;
    end while
end function
function REDUCE(signature sig, (docid docid, sentenceCount S))
    if |S| > 1 then
        emit(sig, S);
    end if
end function
```

Algorithm 2 Random Projection MapReduce Pseudocode

```
function INITIALIZE
    D ← D-bit signature;
    V ← length of generated random vector;
    Q ← number of permutation;
    B ← sliding window size
    T ← distance threshold;
    RandomVector = new List(D);
    for i ← 1...D do
        RandomVector[i] ← generateRandomVector(V);
    end for
    docids ← new QUEUE(B)
    sigs ← new QUEUE(B)
end function
function MAP((docid d,sentenceid senid), docVector t)
    while s ← nextSentence(t) do
        SIG = new List(D);
        for i ← 1...D do
            product ← DotProduct(t, RandomVector[i]);
            if product > 0 then
                SIG[i] ← 1;
            end if
            if product > 0 then
                SIG[i] ← 0;
            end if
        end for
        for i ← 1...Q do
            sigPermute ← SIG.permute;
            emit((i, sigPermute),(docid,sentenceid));
        end for
    end while
end function
function REDUCE((permno p, sig s), (docid d,sentenceid senid))
    for all senid ← in d do
        for i ← 1...sigs.size() do
            dist ← hammingDistance(s, sigs[i]);
            if dist < T then
                emit((docids[i], docid), dist);
            end if
        end for
        sigs.ENQUEUE(s);
        docids.ENQUEUE((docid d,sentenceid senid));
        if sigs.SIZE() > B then
            sigs.DEQUEUE();
            docids.DEQUEUE();
        end if
    end for
end function
```

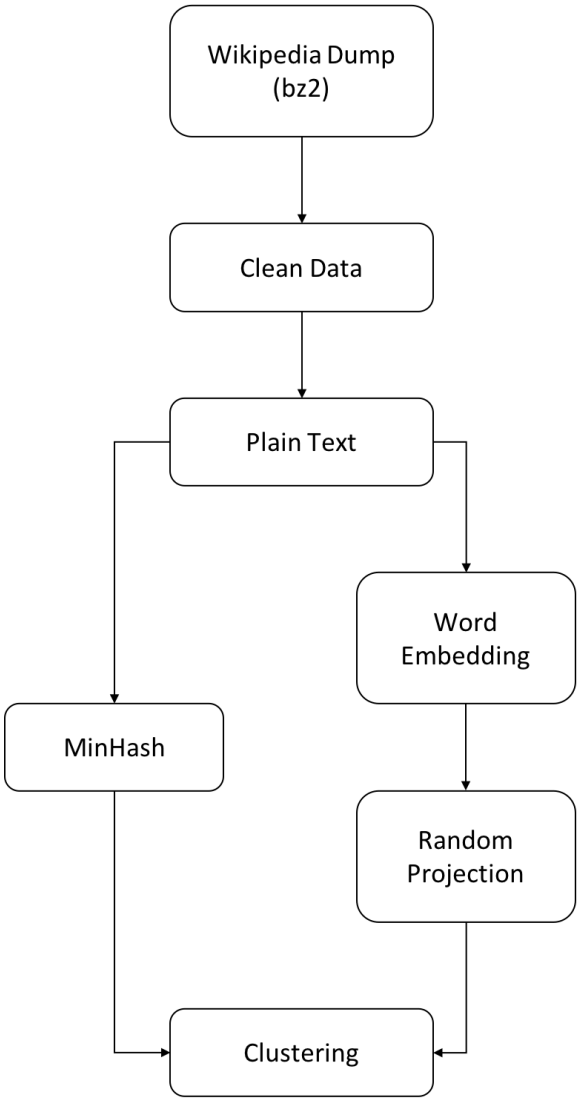

Figure 1: Overview of Implementation

4.2.2 Random Projection

In the implementation of random projection, we use the same application to do the transformation on Wikipedia XML dumps. Moreover, we apply a regular expression to match sentences in an article and generate a sequence of sorted integers as sentences id. Each article is then split into sentences, and each sentence is with document id and sentence id. As a result, we obtain a sequence of sentences as plain text.

In order to compute random projection signatures, we need to convert each document into a document vector, then we can apply inner dot product of the document vector and each generated random vector. Usually, the real-valued random vectors are generated with length V, where V is the vocabulary size of the document collection. Then, each document could be represented as one-hot vector. Since we are dealing with Wikipedia source, the vocabulary size of our collection can be expected as large as the dictionary vocabulary size which is really. On the other hand, we attempt to detect similarity on sentence level, which means our document (sentence in our context) may not contain many terms. Therefore, the document vector will be very sparse if we use one-hot representation. Instead, we will use word embedding as document vector to represent each sentence.

Word2Vec[8] and is one of the most popular word embedding algorithms to obtain dense vector representation of each term[10]. Instead of training our own embedding, we use the publically available pre-trained model from Word2Vec. The Word2Vec model we choose was trained on 400 m tweets in English with 400-dimension representation. Before fitting the Word2Vec model, we remove the stopwords, numbers, any other signs except english word, turn each word into lower case, do the stemming and tokenize each sentence using Python NLTK[5] library. Each sentence is transformed into a vector of tokens. After implementing the Word2Vec, each word is then mapped into a float number. Final vector representation of a sentence is calculated as an average of the sum of vectors of all the words in a sentence. As a result, for each sentence, we obtain a sequence including document id, sentence id, and sentence vector with length 400. Therefore, we generate the random vector with length 400 as well.

4.3 Implementation Details

The Input for MinHash is pairs of Doc ID and Wikipedia articles per line and the output is pairs of sentence signature and sentence ID. The sentence signature is an array of Long and the sentence ID is a string which contains Doc ID and the sentence's position in that article in the format of [DocID]:[Position].

The input for Random Projection is pairs of Doc ID and one sentence from Wikipedia articles per line and the output is pairs of hamming distance and sentence ID. The hamming distance is an Integer and the sentence ID is the same as stated above.

Table 1: Input Format

	Input Format	Data Type
MinHash	(Doc ID, Article)	(Long, String)
Random Projection	(Doc ID, Sentence)	(Long, String)

Table 2: Output Format

	Output Format	Data Type
MinHash	(Sig SentID)	(Array[Long], String)
Random Projection	(Dist, SentID)	(Array[Int], String)

4.3.1 MinHash

Table 3: Program Configuration

Option	Description	Default Value
hashfuncs	number of hash functions	20
hashbitsnubmer	number of hash bits	60
siglen	length of signature	10
draw	draw times	10
shingle	length of shingle	12
rseed	random seed	1123456
max	max length of sentence	600
min	min length of sentecne	75

4.3.2 Random Projection

Table 4: Program Configuration

Option	Description	Default Value
veclen	length of vector	400
siglen	length of sinature	100
permutate	permutation times	10
threshold	distance threshold	5
window	sliding window size	10
rseed	random seed	1123456

4.4 Generate Final Cluster

In Weissman's work[12],since the output of MinHash is a set of similar sentence pairs, they merge all pairs that share at least one common sentence into clusters, where each cluster represents a signature collision. They accomplish the cluster merging using union-find data structure outside MapReduce. In order to compare the result of our project, we merge the cluster in the same way.

As the output from MinHash and Random Projection is sentence pairs which have signature collision or hamming distance below the threshold, and there may exist duplicate pairs as one sentence will be emitted multiple times during the MapReduce process, we implemented another MapReduce program to merge these sentence pairs into clusters. The basic idea is that if two pairs/clusters share one common sentence then put them into one cluster.

5. EXPERIMENTS RESULT AND DISCUSSION

Our experiments were conducted on an English Wikipedia XML dumps on December 1^{st} in 2015 and test part of Wikipedia XML dumps. The entire collection is around 12.52 GB in bz2 format and contain 5,699,192 articles. We run our Hadoop and Spark implementation of minhash and random projection on the entire collection on Altiscale. It took almost three hours to run implementation of minhash. Due to the memory problem, we failed to run the implementation of random projection on the entire collection.

After minhashing and deduplification, we found a total of 19,143,279 similar sentence pairs. After clustering these pairs, the total number of similar sentence cluster was 2,791,865.

In our clustering result of minhash, the cluster sizes vary from 2 to 3,964. Most sentence clusters are small, and the distribution of the cluster sizes presents a long tail. We put parts of the clustering result in the following Table 5 and following Figure 2. Same as Weissman's result[12], 98% of the cluster have size below or equal to 20.

Table 5: Cluster Size Counts

Cluster Size	Counts	Cluster Size	Counts
2	1,466,897	3	312,827
4	179,540	5	125,002
6	99,925	7	83,545
8	76,992	9	74,766
10	96,461	11	150,173
12	14,086	13	11,644
14	9,559	15	8,051
16	6610	17	5,596
18	5042	19	4,727
20	4,034	21	3,470
22	3,291	23	2,680
24	2,519	25	2,414
26	2,309	27	2,029
28	1,861	29	1,438
30	1,436	31	1,551
32	1,534	33	1,194
34	1,062	35	1,114
36	1,025	37	932
38	822	39	757
40	683	41	748
42	722	43	604
44	673	45	583
46	605	47	594
48	512	49	474
50	529	51	411

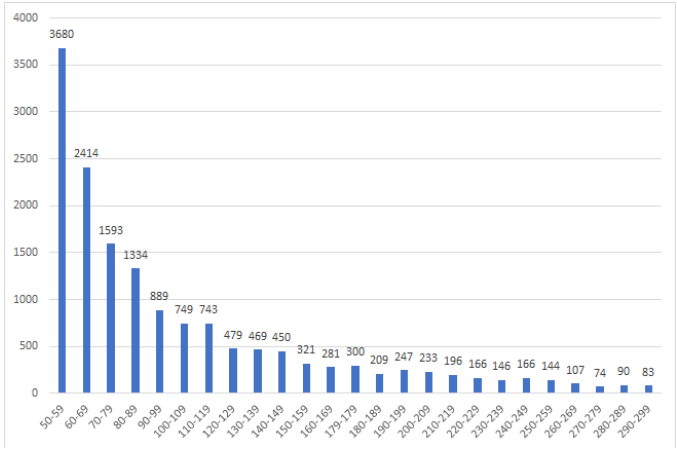

Figure 2: Histogram of Cluster Sizes

Due to the failure of running random projection, we have not obtained any reliable result of so far. In order to compare the efficiency of two LSH techiniques, we use the running result on test dump data to compare two implementations. On the test dump dataset, we have 103 articles and extract 12,966 sentences after applying wikiclean. The running time of each implementation is as follows:

Table 6: Running Time of Implementations

	MapReduce	Spark
minhash	5.766 s	6.982 s
random projection	37.927 s	41.832 s

As a result, we can make a conclusion that minhash implementation is more efficient than the random projection implementation. Due the limitatio of the test dump data set, we found the result of two implementation are similar and according analysis is nor reliable enough.

In Weissman's work[12], they manually inspected the output the cluster and explored different similar sentence types. They summarized types as Templatification, Reference, Direct Copying, Direct Copying with Copy Editing, and Direct Copying with Factual Drift. In our project, we did not replicate this part of the work. We did find that such types of similar sentences appear in our detection result. For example, we detected similar sentences of Templatification type:

- Literacy is quite low, with 67.4% of the population over the age of 15 able to read and write in Portuguese.
- Education Literacy is quite low, with 67.4% of the population over the age of 15 able to read and write in Portuguese

6. CONCLUSION AND FUTURE WORK

This report presents our work on replicating Weissman's result [12], the detection of similarity sentences of Wikipedia articles. We use MinHash and Random Projection, which are locality sensitive hashing (LSH) techniques on MapReduce and Spark framework. In the implementation of Random Projection, we use word embedding to transform each document(sentence in our context) into document vector with fixed length and generate real-valued random vector with the same length. We avoid the sparsity of the document vector and random vector. Instead of selecting first K bits from D-bit signatures with m times, as described in class, we implement sliding window algorithm to avoid the loading imbalance. Our experimental results verify that we successfully replicate the MinHash and implement the Random Projection to detect the near-duplicate sentences.

In the future, we plan to do more efficiency and effectiveness analysis on our result of near-duplicate detection on Wikipedia articles and running time of implementing minhash and random projection.

References

[1] lintool:cloud9. `https://github.com/lintool/Cloud9`, 2018.

[2] Multply shift. `http://en.wikipedia.org/wiki/Reliability_of_Wikipedia`, 2018.

[3] Wikiclean. `https://github.com/lintool/wikiclean`, 2018.

[4] Wikipedia. `https://en.wikipedia.org/wiki/Wikipedia`, 2018.

[5] S. Bird. Nltk: The natural language toolkit. In *Proceedings of the COLING/ACL on Interactive Presentation Sessions*, COLING-ACL '06, pages 69–72, Stroudsburg, PA, USA, 2006. Association for Computational Linguistics.

[6] A. Z. Broder. On the resemblance and containment of documents. In *In Compression and Complexity of Sequences (SEQUENCESâĂŹ97)*, pages 21–29, 1997.

[7] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*, 2002.

[8] F. Godin, B. Vandersmissen, W. D. Neve, and R. V. D. Walle. Multimedia lab @ acl wnut ner shared task: Named entity recognition for twitter microposts using distributed word representations. *Proceedings of the Workshop on Noisy User-generated Text*, 2015.

[9] M. X. Goemans and D. P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *J. ACM*, pages 1115–1145, 1995.

[10] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013.

[11] F. Ture, T. Elsayed, and J. Lin. No free lunch: Brute force vs. locality-sensitive hashing for cross-lingual pairwise similarity. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Informatio Retrieval*, 2011.

[12] S. Weissman, S. Ayhan, J. Bradley, and J. Lin. Identifying duplicate and contradictory information in wikipedia. In *Proceedings of the 15th ACM/IEEE-CS Joint Conference on Digital Libraries*, 2014.