Document Similarity Detection using Map Reduce

Prashant Kumar, Graduate student, Computer Science, Iowa State University, pku@iastate.edu

ABSTRACT

The amount of data is growing at an exponential rate, doubling every two years. Plagiarism cases are likely to rise in future with growing data. To protect the privacy of the documents we need to come up with an approach to develop tools that can run on huge amount of data available efficiently. One approach is to make use of MapReduce methodology of Hadoop to run our document similarity algorithm on huge amount of data distributed on different servers. In this project we have implemented n-gram algorithm using Hadoop MapReduce. We have tested it on single node and multi node.

Keywords

Hadoop, Apache, Big Data, MapReduce, n-gram, Document Similarity, Detection, clusters, stopwords.

1. PROBLEM STATEMENT

[1]Due to the advent of new technologies, devices, and communication means like social networking sites, the amount of data produced by mankind is growing rapidly every year. The amount of data produced by us from the beginning of time till 2003 was 5 billion gigabytes. If you pile up the data in the form of disks it may fill an entire football field. The same amount was created in every two days in 2011, and in every ten minutes in 2013. This rate is still growing enormously. Though all this information produced is meaningful and can be useful when processed, it is being neglected.

The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. Rather than rely on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures.

Hadoop runs applications using the Map Reduce algorithm, where the data is processed in parallel on different CPU nodes. In short, Hadoop framework is capable enough to develop applications capable of running on clusters of computers and they could perform complete statistical analysis for huge amounts of data as shown in Figure 1.

[2]Formally, a language model is a probability distribution over a sequence of words. N-gram model is a type of probabilistic language model for predicting the next item in a sequence which is a (n-1) order Markov model. Specifically, a Language Model (LM) estimates the probability of next word from the given preceding words. An N-gram can be simply defined as a sequence of N words. An N-gram language model uses the history of N-1 immediately preceding words to compute the probability (P) of the occurrence of the current word. An N-gram of size 1 is called unigram, size 2 a bigram (or digram), size 3 a trigram, and so on.

For example, the word “India is my country” can be decomposed using n gram as:

Character based

1) Unigram characters : (I), (n), (d), (i), (a)...

2) Bigram characters : (In), (nd), (di), (ia)..

3) Trigram characters : (Ind), (ndi), (dia)..

Word based

1) Unigram word : (India), (is), (my), (country)

2) Bigram word : (India is), (is my), (my country)

3) Trigram word : (India is my), (is my country)

The N-gram approximation for calculating the next word in a sequence is:

$$P(X_1....X_n) = P(X_1)P(X_2|X_1)P(X_3|X_1^2).....P(X_n|X_1^{n-1})$$

$$= \prod_{k=1}^{n} P(X_k|X_1^{k-1})$$

Probability of a complete string:

$$P(w_1^n) = P(w_1)P(w_2|w_1)P(w_3|w_1^2).....P(w_n|w_1^{n-1})$$

$$= \prod_{k=1}^{n} P(w_k|w_1^{k-1})$$

Most modern applications such as machine translation applications rely on N-gram based models. N-gram models are widely used in statistical natural language processing. In speech recognition, phonemes and sequences of phonemes are modeled using N-gram distribution. N-grams are also crucial in natural language processing tasks like part-of speech tagging, natural language generation, as well as in applications like authorship identification and sentiment extraction in predictive text input systems for cell phones etc.

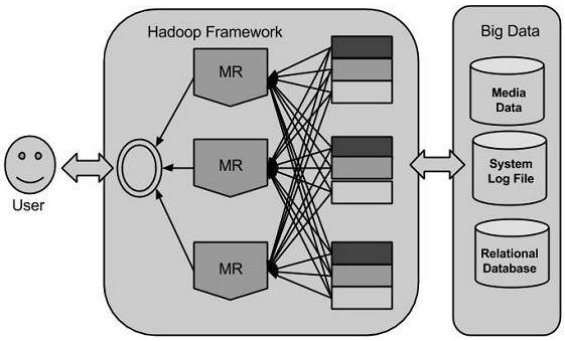


Fig. 1

The paper is organized as follows: in next section related work is presented briefly before we explain the application setup in section 3. In section 4, details of the machines used for Multi cluster setup. In section 5, technologies used and finally, we sum up the conclusion and future work in section 6.

2. RELATED WORK

2.1 Detecting Text Similarity over Chinese Research Papers Using Map Reduce

[3] This paper proposes a novel method to detect text similarity over Chinese research papers using Map Reduce paradigm. Their approach differs from the state-of-the-art methods in two aspects. First, they extract the key sentences from Chinese research papers by using some heuristic features and then generate 2-tuple, (document id, key phrase), as the representation of the documents. Second, they design 2-phrase Map Reduce algorithm to verify the effectiveness of the generated 2-tuple.

Algorithm 1: 2-tuple Generation Algorithm

Input: Chinese Research Papers
Output: (Document ID, Key Phrase) of Each Paper

BEGIN
For each Chinese research paper
Extract temporal key sentence according to the title, abstract and keywords ,and load them into set TKS;
Parse each sentence in set TKS, and generate parsing tree;
Extract the NP,VP and ADJP from above each parsing tree, and generate headwords set H;
Extract the key sentence using the headwords set H matching, and load them into set KS;
Parsing each sentence in set KS, and generate 2-tuple (document id, key phrases).
End For
End.

Algorithm 2: The Pseudo-code of 2-phrase MapReduce Algorithm

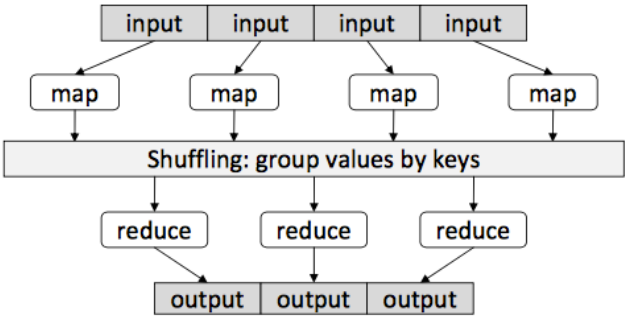
```
Map1(docid[i],key phrase of docid[i],cardinality of docid[i])
1. for all docid[i] do
2.   for all key phrase of docid[i] do
3.     EMIT(key phrase of doid[i],docid[i],cardinality of docid[i])
4.   endfor
5. endfor

Reduce1(key phrase of docid[i],docid[i],cardinality of docid[i])
1.  for all docid[i] do
2.   for all key phrases of docid[i] do
3.     EMIT(R, docid[i], cardinality of docid[i]])
//R is an empty placeholder to load balance usage.
4.  endfor
5.  endfor

Map2(R, docid[i], cardinality of docid[i])
1. for all match of docid[i] and cardinality of docid[i] do
2.  if(match of docid[i] contains docid[0])
//docid[0] is the document needs to be detected.
3.     EMIT(match of docid[i] and docid[0],sum of cardinality of
both)
4. endfor
```

2.2 Pairwise Document Similarity in Large Collections with Map Reduce

[4] This paper presents a Map Reduce algorithm for computing pairwise document similarity in large document collections. Map Reduce is an attractive framework because it allows us to decompose the inner products involved in computing document similarity into separate multiplication and summation stages in a way that is well matched to efficient disk access patterns across several machines. On a collection consisting of approximately 900,000 newswire articles, they algorithm exhibits linear growth in running time and space in terms of the number of documents.

Algorithm 1 Compute Pairwise Similarity Matrix

1: $\forall i, j : sim[i, j] \Leftarrow 0$
2: **for all** $t \in V$ **do**
3: $\quad p_t \Leftarrow postings(t)$
4: $\quad$ **for all** $d_i, d_j \in p_t$ **do**
5: $\quad\quad sim[i, j] \Leftarrow sim[i, j] + w_{t,d_i} \cdot w_{t,d_j}$



3. PROPOSED IMPLEMENTATION

In this project, document similarity detection using MapReduce, we have developed an application using Hadoop MapReduce n-gram algorithm for English documents.

3.1 The Pseudo-code of MapReduce using n-gram Algorithm.

mapper(Object key, Text value, Context context):

a. tokenize lines in the file
b. for each tokenizer
    a. filter punctuation marks
    b. filter stop words
    c. generate n-grams for value=3(Too high of a number, and your output will be a word for word copy of the original, too low of a number, and the output will be too messy)

reducer(Text key, Text values, Context context):

a. for each of the n-gram combination values
    a. compute total occurrences of the n-gram combinations
    b. write to output: (n-gram combination, total occurances)

3.2 Stop words

[8]In computing, stop words are words which are filtered out before or after processing of natural language data (text). Though stop words usually refer to the most common words in a language, there is no single universal list of stop words used by all natural language processing tools, and indeed not all tools even use such a list. For our application we have used following stop words:

"without", "see", "unless", "due", "also", "must", "might", "like", "]", "[", "}", "{", "<", ">", "?", "\"", "\\", "/", ")", "(", "will", "may", "can", "much", "every", "the", "in", "other", "this", "the", "many", "any", "an", "or", "for", "in", "an", "an ", "is", "a", "about", "above", "after", "again", "against", "all", "am", "an", "and", "any", "are", "aren't", "as", "at", "be", "because", "been", "before", "being", "below", "between", "both", "but", "by", "can't", "cannot", "could", "couldn't", "did", "didn't", "do", "does", "doesn't", "doing", "don't", "down", "during", "each", "few", "for", "from", "further", "had", "hadn't", "has", "hasn't", "have", "haven't", "having", "he", "he'd", "he'll", "he's", "her", "here", "here's", "hers", "herself", "him", "himself", "his", "how", "how's", "i ", " i", "i'd", "i'll", "i'm", "i've", "if", "in", "into", "is", "isn't", "it", "it's", "its", "itself", "let's", "me", "more", "most", "mustn't", "my", "myself", "no", "nor", "not", "of", "off", "on", "once", "only", "ought", "our", "ours", "ourselves", "out", "over", "own", "same", "shan't", "she", "she'd", "she'll", "she's", "should", "shouldn't", "so", "some", "such", "than", "that", "that's", "their", "theirs", "them", "themselves", "then", "there", "there's", "these", "they", "they'd", "they'll", "they're", "they've", "this", "those", "through", "to", "too", "under", "until", "up", "very", "was", "wasn't", "we", "we'd", "we'll", "we're", "we've", "were", "weren't", "what", "what's", "when", "when's", "where", "where's", "which", "while", "who", "who's", "whom", "why", "why's", "with", "won't", "would", "wouldn't", "you", "you'd", "you'll", "you're", "you've", "your", "yours", "yourself", "yourselves", "Without", "See", "Unless", "Due", "Also", "Must", "Might", "Like", "Will", "May", "Can", "Much", "Every", "The", "In", "Other", "This", "The", "Many", "Any", "An", "Or", "For", "In", "An", "An ", "Is", "A", "About", "Above", "After", "Again", "Against", "All", "Am", "An", "And", "Any", "Are", "Aren't", "As", "At", "Be", "Because", "Been", "Before", "Being", "Below", "Between", "Both", "But", "By", "Can't", "Cannot", "Could", "Couldn't", "Did", "Didn't", "Do", "Does", "Doesn't", "Doing", "Don't", "Down", "During", "Each", "Few", "For", "From", "Further", "Had", "Hadn't", "Has", "Hasn't", "Have", "Haven't", "Having", "He", "He'd", "He'll", "He's", "Her", "Here", "Here's", "Hers", "Herself", "Him", "Himself", "His", "How", "How's", "I ", " I", "I'd", "I'll", "I'm", "I've", "If", "In", "Into", "Is", "Isn't", "It", "It's", "Its", "Itself", "Let's", "Me", "More", "Most", "Mustn't", "My", "Myself", "No", "Nor", "Not", "Of", "Off", "On", "Once", "Only", "Ought", "Our", "Ours", "Ourselves", "Out", "Over", "Own", "Same", "Shan't", "She", "She'd", "She'll", "She's", "Should", "Shouldn't", "So", "Some", "Such", "Than", "That", "That's", "Their", "Theirs", "Them", "Themselves", "Then", "There", "There's", "These", "They", "They'd", "They'll", "They're", "They've", "This", "Those", "Through", "To", "Too", "Under", "Until", "Up", "Very", "Was", "Wasn't", "We", "We'd", "We'll", "We're", "We've", "Were", "Weren't", "What", "What's", "When", "When's", "Where", "Where's", "Which", "While", "Who", "Who's", "Whom", "Why", "Why's", "With", "Won't", "Would", "Wouldn't", "You", "You'd", "You'll", "You're", "You've", "Your", "Yours", "Yourself", "Yourselves"

3.3 Characters allowed:

"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"

4. HADOOP CLUSTER ENVIRONMENT

To setup multi node, we have used

a. Mac Book Pro, 2.3 GHz i5, 4 GB RAM as slave for datanode

b. Mac Book Air, 1.6 GHz i5, 4 GB RAM as master for namenode

5. Technologies used

The application was developed using Apache Hadoop 2.7.2 and jdk 1.8, eclipse IDE.

6. CODE

Mapper Implementation:

```
public void map(Object key, Text value, Context context)
                throws IOException, InterruptedException {
        String filenameStr = ((FileSplit) context.getInputSplit()).getPath().getName();
        filename = new Text(filenameStr);
        String line = value.toString();
        String word = "";
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
                word=tokenizer.nextToken();
                word=filterPunctuations(word);
                word=filterStopWords(word);
                if (word.length()!=0){
                word = getNGram(word,nGramValue);
                if (word.length()!= 0) {
                words.set(word);
                }
                }
                context.write(words, filename);
        }
}
```

Reducer Implementation:

```
protected void reduce(Text key, Iterable<Text> values, Reducer<Text, Text, Text, Text>.Context context)
                throws IOException, InterruptedException {
        int sum = 0;
        for (Text value : values) {
                int val = 0;
                try{
                        val += Integer.parseInt(value.toString());
                }
                catch(Exception e){
                        val = 1;
                }
                sum += val;
        }
        context.write(key, new Text(sum+""));
}
```

n-gram implementation:

```
public static String getNGram(String word, int nGramValue){
        String result = "";
        if (wordsCount< nGramValue){
                ngram = ngram+" "+word;
                wordsCount++;
        }
        if (wordsCount == nGramValue){
                result = ngram.trim();
                wordsCount = nGramValue-1;
                ngram = result.substring(result.indexOf(" ",0)+1, result.length());
        }
        return result;
}
```

github.com/prashant4nov/similarity-detection/tree/master/src

7. CONCLUSION AND FUTURE WORK

We successfully implemented Map Reduce n-gram algorithm using Hadoop and completed it's setup on multi node cluster and tested it with 2 input text files for n-gram value 3 to compare the number of combinations of n-gram same in the 2 documents. For future work, Jaccard Similarity can be computed and the performance of the implementation can be tested against other implementations of document similarity such as pair wise document similarity. Apart from this, it can be extended for other languages as well. The amount of the sample data input can be increased to test it on bigger inputs.

8. REFERENCE

[1] http://www.tutorialspoint.com/hadoop/hadoop_big_data_overview.htm

[2] http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6726777

[3] http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6063565

[4] http://www.ece.umd.edu/~oard/pdf/acl08elsayed2.pdf

[5] https://storify.com/Jonzey08/is-involvement-in-social-media-the-keys-to-success

[6] http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-multi-node-cluster/

[7] https://hadoop.apache.org/docs/r2.7.1/hadoop-project-dist/hadoop-common/ClusterSetup.html

[8] https://en.wikipedia.org/wiki/Stop_words