Parallel String Similarity Join with Word Vectors Using MapReduce

Final Report, 4th Jan '16

Selim Eren Bekçe
Dept. of Computer Engineering
Bilkent University
Ankara, Türkiye
eren.bekce@bilkent.edu.tr

***Abstract*— *In this paper, we will show how to efficiently perform string similarity joins in parallel using the MapReduce framework. We will also use a semantic word to vector data set on extracted words to investigate semantic similarity of each record pair. If we compared every record with each other (e.g. the brute force method), the runtime complexity quickly becomes infeasible, so we will apply a filtering function first to prune dissimilar pairs.*

I. Introduction

Finding similar pairs of strings has very wide usages, such as plagiarism detection, finding similar tweets for social media analytics, detection of near-similar web pages for pruning web crawling output and finding matching pairs for data management applications. Some data management applications have data merging needs where the same attribute is duplicated across many tables and needs to be merged as automatic as possible.

As the data in our world grows exponentially (e.g. big data) the amount of data usually exceeds the memory capacity of one machine. Therefore, we need to scale the computation horizontally to multiple machines (e.g. scale out) in an efficient manner. MapReduce is the popular and effective framework of choice when it comes to parallel computation, we will propose an algorithm to run on Hadoop MapReduce framework.

The brute force method to find similarity set is to compare every item with other, which needs n(n-1)/2 comparisons in total, leading to runtime complexity of O(n^2). There are several optimization methods (e.g. filtering) to prune the number of pairs to compare so that the complexity drops significantly for many types of data sets. We will employ at least one filtering method to reduce number of comparisons to calculate.

The simplest idea for the similarity join algorithm is as follows. Tokenize the record into multiple tokens (words). Tokenization is a very important step which defines the quality of the join and affects the number of compared pairs. We prune the pairs of records which share no common tokens. This is the filtering part. Then, compare the remaining pairs of records and calculate the similarity of records based on their common token counts.

Text mining and NLP techniques had evolved to find semantic meanings of texts in context. There are tools and data sets to run on corpus data and map words to vectors in multidimensional space w.r.t. their context in sentences. We can thus calculate the cosine similarity of word pairs from this multidimensional word space to find the k-NN of each word. We can thus use this data in our similarity join problem to find semantically close pairs of records.

For example, consider two sentences “Our country is brimming with energy” and “My nation is a bright star”. With plain count filtering similarity-join, we would find no correlation between those sentences because there are no common word pairs (assuming we prune some common words such as 'is', 'a', 'with', etc). However, by using the word to vector data, we would find a similarity between those sentences because they are semantically close.

II. Background Information

A. String Similarity Join

There are two main methods to investigate similarity of two strings. Character based methods calculate edit distance (i.e. the number of character addition, deletion or replacement operations to transform one string to another). Token based methods apply set-similarity join techniques after transforming the text into set of tokens [1]. We will explore token-based similarity join with Jaccard metric on our project.

B. Word Vectors

Word2Vec is a technique which produces high dimensional vectors from a large text corpora. The output is a set of vectors for each word in the given corpus, located in the multi dimensional feature space with respect to their semantic meaning. Such that, the cosine similarity value of similar words (vectors) tends to be higher than dissimilar pairs.

The method is a series of vector operations to preserve linear regularities amongst words. It employs some deep learning and neural network techniques. [3]

Earliest Neural Network Language Models (NNLM) date back to 1986. However, these methods were not efficient for the scale of today's data, some recent scalable methods (skip-gram and continuous bag of words) have emerged since then. [4] [5]

Example runtime output of finding similar tokens: Word: chuckles Position in vocabulary: 40984

word	similarity
laughs	0.808566
chuckle	0.720890
guffaws	0.691016
laughter	0.681049
snickers	0.670588
smirks	0.657066
giggles	0.656259

The similarity value is the cosine difference value of the vector pair.

III. Work Done

A. Tokenization

Tokenization is a very important aspect which directly affects the runtime performance of the algorithm because the output of the tokenization affects the number of pairs compared.

An excessive tokenization logic would produce large number of tokens from each record, leading to high number of comparisons where performance degrades sharply with small increases in number of records, thus discarding the benefits of count filtering.

A conservative tokenization, on the other hand, would produce poor results that even clearly similar records would go unnoticed by the algorithm because the tokenization did not consider punctuation, accents, casing, etc.

However, excellence of a tokenization technique is highly dependent to the record nature, so there is no single good way to do it.

Our evaluations were performed on Twitter tweets. We have decided to tokenize on non-alphanumeric characters, that is, whitespace plus punctiation. However, special care taken to not split urls (http://www…) because splitting urls leads to excessive tokenization. We then omitted short tokens (less than 3 characters) and lowercased all tokens. We did not have a chance to evaluate other tokenization options such as Treebank-style, Sentiment-aware [10], etc.

B. The Algorithm

We take set of records and produce signatures from each record:

1. Tokenize the record, do appropriate data cleaning and get list of tokens, with their occurrence counts.
2. Contact the word to vector data set and get the semantically close words w.r.t. cosine similarity for each token on the list.
3. For each semantically close word, multiply its occurrence count with cosine similarity value and append it to list. Note that the occurrence value has become decimal.

At the end we will have a mixed list of tokens and counts, emit them as `key → (record, total_count, count)`. total_count is the sum of all token counts for given record. We need this value in our last step, where we calculate the Jaccard similarity values.

On the next step, emit each record pair:
`((record 1, total_count), (record 2, total_count)) → (key, count)`

On the final step, calculate similarity of each record pair w.r.t. Jaccard filtering and given threshold value. Jaccard count filtering simply selects pairs when this condition is met:
`| a ∩ b | / | a u b | >= t`

This is a pseudo-code for the similarity function:

```
is_similar (r1, r2, threshold) : boolean
define shared as number_of_shared_tokens_in(r1,r2)
define similarity as (r1.total_count + r2.total_count
- shared) / shared
return similarity >= threshold
```

C. Similarity Join Example

We will work through an example case. Input records:

R1: A B C A B A
R2: B A D B D
R3: E F F D
R4: D E G E G G D
R5: A A B

Phase 1: Tokenize records then group by token

A → (R1, 6, 3), B → (R1, 6, 2), C → (R1, 6, 1) //R1
B → (R2, 5, 2), A → (R2, 5, 1), D → (R2, 5, 2) //R2
E → (R3, 4, 1), F → (R3, 4, 2), D → (R3, 4, 1), //R3
D → (R4, 7, 2), E → (R4, 7, 2), G → (R4, 7, 3), //R4
A → (R5, 3, 2), B → (R5, 3, 1) //R5

Then,

A → ((R1, 6, 3), (R2, 5, 1), (R5, 3, 2))
B → ((R1, 6, 2), (R2, 5, 2), (R5, 3, 1))
C → ((R1, 6, 1))
D → ((R2, 5, 2), (R3, 4, 1), (R4, 7, 2))
E → …, F → …, G → …

Phase 2: Produce combinations of each pair then group by pair. Generate record pair combinations for A, take minimum occurrence for each pair then emit:

((R1, 6), (R2, 5)) → (A, 1) // min(3,1)=1
((R1, 6), (R5, 3)) → (A, 2) // min(3,2)=2

((R2, 5), (R5, 3)) → (A, 1) // min(1,2)=1
((R1, 6), (R5, 3)) → (B, 1) // min(2,1)=1

Then, group by pair and sum the common occurance counts,

((R1, 6), (R5, 3)) → 3 // (A,2) + (B,1)

Phase 3: Calculate similarity. By now, we have everything to calculate Jaccard Similarity score: 3 / (6 + 5 - 3) = 0.375

$$\frac{|R1 \bigcap R2|}{|R1| + |R2| - |R1 \bigcap R2|} > t$$

If the value is bigger than some threshold, we report the pair as similar, else, we discard it.

D. Similarity Join with Word2Vec Example

We will query Word2Vec in Phase 1 and get semantically close words to the current word, then multiply the occurance count with cosine difference.

Input Records:

R1: B A D B D
R2: E F F D
R3: G G H

Word Vectors: Here we show 3 word pairs with their cosine difference in paranthesis. Reverse mappings were omitted for clarity.

A → G (0.65)
B → A (0.55)
B → G (0.45)

Phase 1: We have modified Phase 1 so that for each token in the record, we query Word2Vec data and get its nearest words. We use their cosine difference and emit them as normal tokens: their cosine difference now become their occurance count in the output. The total size of the record is also updated to match total output.

B → (R1, 7.65, 2)
A → (R1, 7.65, 2.1)
D → (R1, 7.65, 2)
G → (R1, 7.65, 1.55)

Phase 2 and 3 are the same as plain similarity join.

In this example, before Word2Vec, there was no correlation for pairs R1 and R3. However after we consult word vectors, R1 and R3 become eligible for similarity calculation as they now have token 'G' in common. This will lead to more pairs to be eligible for similarity calculation.

E. Pig Algorithm

We have developed the similarity join algorithm in Pig Latin. Apache Pig is a project in Hadoop ecosystem, which provides an easy way to develop MapReduce programs, without regular hiccups of Java programming. It is an imperative programming style which lets users to specify how to perform operations on data in an intuitive manner. It has built in functions that can perform common data processing operations like grouping, filtering, sorting, transforming, flattening, joining, etc. [8]

Pig transforms Pig Latin programs to Hadoop jobs at runtime, which directly uses the underlying HDFS to achieve MR-style parallel programming.

We have implemented both plain Similarity Join and the one with word2vec in Pig Latin. We have planned to run both versions for comparison.

Apache Pig has User Defined Functions feature which allows users to write custom functions to apply to data. We have written two UDFs to implement Phase 1 and Phase 2 as in my original design. The sources are attached to the report.

F. Spark Algorithm

Apache Spark is an engine for large-scale parallel data processing. Users can write jobs in Scala which takes advantage of in-memory computation (if possible), which happens to be faster than using Hadoop, because Hadoop is more dependent on I/O operations. Spark introduces the concept of resilient distributed data sets (RDD) and each operation is either a transformation (like map, filter, sample, groupByKey, reduceByKey) or an action (like persist, dump) which marks the end of computation. [9]

We have also implemented similarity join in Spark, without word vector feature. The source of the spark code is attached to the report.

IV. Evaluation & Discussion

Operations were performed using one computer with 16 GB of RAM.

A. Data Collection (Tweets)

Input data was collected using Twitter API's sample endpoint, which streams a portion of global public tweets all the time. We have written the data collector in Java, and programmed it to discard non-English tweets and retweets of other tweets. Retweets of other tweets will not constitute anything to the result as we know that they are equal (except a "RT" at the beginning).

In approximately one day, it collected near 500K tweets, which met our goal. We then formatted the tweets as a TSV file with two columns: tweet id and tweet content.

B. Word2Vec

There are two implementations of Word2Vec in public domain, one is the original C implementation [7], the other is a Java implementation in the open source Deeplearning4j project [6]. We have done experiments with both of them.

The original one runs very efficiently but it is poorly written so it is hard to refactor for external use. The latter has a Java API which can be easily consumed by our Pig UDF, but currently its performance suffers such that a query takes 60 seconds to run, which made it impossible to use in production.

We have also contacted with the developers of the deeplearning4j, they recognized the problem but offered no current solution. The next step would be to refactor and port the C code to Java, which will enable the use in production environment.

We have used the pre-trained vector data trained on Google News dataset (about 100 billion words), which contains 300-dimensional vectors for 3 million words and phrases [6][7]. This pretrained dataset is not exactly perfect for our needs. Vectors are case sensitive: since we chose to lowercase all tokens, we miss some important relations such as 'İstanbul' and 'Turkey'. It has ngrams (multiple word tokens) that we don't employ in our algorithm. Data is not filtered the same way as we do. However, training such vectors take very long time on common hardware so even though it wasn't the best fit, the data is big enough to catch good semantic similarities.

On loading the model, Word2Vec reads the compressed binary model file then constructs in-memory lookup table to serve nearest word queries. However, this process needs a large portion of memory (6-8 GB) to be available in the worker node. Since MapReduce works distributed, Word2Vec model needs to be loaded on every worker node, which also means that every worker node will be required to have specified amount of RAM available. This extra requirement may affect some cluster configuration because many common jobs wouldn't require that kind of memory. Distribution of the Word2Vec model file can either be done manually to every node or it can be placed in HDFS where participating nodes can read from.

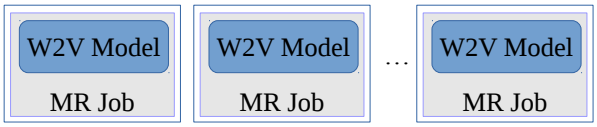
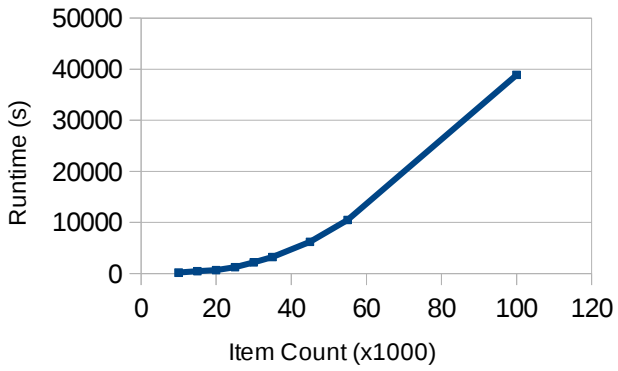


Figure: Memory of worker nodes while running SimJoin with Word2Vec. Each node loads W2V model at the start of the job.

C. Performance: Pig Algorithm

Experiments of plain similarity join algorithm were performed with different number of input data on a single node.



Input size (x1000)	Runtime (s)	count (and percentage) of similar pairs, t = 0.6
10	186	1866 (0.00373 %)
15	465	3814 (0.00339 %)
20	695	5094 (0.00255 %)
25	1243	9940 (0.00318 %)
30	2201	14273 (0.00317 %)
35	3248	19070 (0.00311 %)
45	6212	31187 (0.00308 %)
55	10514	45074 (0.00298 %)
100	38895	116749 (0.00234 %)

Third column denotes the number of similar pairs reported for each case. The percentage value in paranthesis denote what percentage is the similar pairs to all possible pairs (n^2-n)/2, which suggests that similar pairs are very sparse (approx. 1 in 31000 pairs).

The reason of the sharp increase for 100K data is that the intermediate data to reduce was too big (around 50 GB observed) for one node to process efficiently, so the reduce took too much time to complete. We need more nodes to test this.

The line is somewhere between O(n) and O(n^2) just as expected: Count filtering technique prunes dissimilar pairs so average runtime complexity is less than the worst case complexity O(n^2). This can be improved with multiple nodes and/or other aggressive pruning techniques.

D. Performance: Spark Algorithm

Input size (x1000)	Runtime (s)
5	137
10	1269
20	6681

Spark application took more time because of high garbage collection needs while processing RDDs. Every transformation creates many objects on the fly so it leads to high amount of GC activity. The code can be tweaked to achieve better timings.

E. Sample Output & Discussion

Table below shows partial output of the algorithm.

Tweet 1	Tweet 2	Score
@Ntshalie thank you	>thank-you! MG https://t.co/OKQd1C4qdt	0.6667
Happy birthday!!!:)) hope you have a great day @alexbradbury33 https://t.co/tTX84X1tzs	@alexissopata happy 18th birthday! Hope you have a great day	0.7

one person followed me // automatically checked by https://t.co/bc1Vm47Cho	one person unfollowed me // automatically checked by https://t.co/vGocgQxKng	0.6667
I've harvested 1,750 of food! https://t.co/K83OFXYxeK #android, #androidgames, #gameinsight	I've harvested 377 of food! https://t.co/WzlHXGASgF #android, #androidgames, #gameinsight	0.7143
Just posted a photo https://t.co/7rGxjVujHm	Just posted a photo https://t.co/oHRJKWmQm5	1.0

High scored pairs are mostly auto generated tweets from various applications and websites as they generally offer a 'tweet this' button which offers a template which most people do not bother to change before posting it.

There are also authentic pairs such as “happy birthday” and “thank you” messages in the table. They have high score because they are short and shared tokens constitute a high percentage of all tokens. If either one was longer, the score would decrease so they may not be able to reach the threshold.

By intuiton, it is also unlikely that human created content to have high Jaccard score without direct copying involved. Filtering out top score (> 0.9) pairs will likely prune some of the non-authentic tweet pairs.

V. Conclusion & Future Remarks

This project has helped us to learn and apply much about similarity join with MapReduce and word vectors.

Still, there can be many improvements:

- Other token-based metrics such as OVERLAP, DICE, COSINE can be experimented.
- Aggressive pruning (filtering) can be implemented such as prefix filtering. This will greatly help lower runtime costs.
- Trials with multiple nodes can be performed to explore the algorithm scalability.
- Maximum score upper bound can be used to prune machine-generated record pairs.
- Use a better tokenization technique for high quality results.
- Experiment with different datasets such as articles, homework assignments, other social network posts.
- Port Word2Vec to Java for better performance.
- Retrain Word2Vec vector data with relevant optimizations.

VI. References

[1] Yu Jiang, Guoliang Li, Jianhua Feng and Wen-Syan Li. "String Similarity Joins: An Experimental Evaluation".

[2] R. Vernica, M. Carey, Chen Li. "Efficient Parallel Set-Similarity Joins Using MapReduce".

[3] Tomas Mikolov, Kai Chen, Greg Corrado and Jeffrey Dean. "Efficient Estimation of Word Representations in Vector Space". arXiv:1301.3781 [cs.CL]

[4] Yoav Goldberg, Omer Levy. "word2vec Explained: deriving Mikolov et al.'s negative-sampling word-embedding method". arXiv:1402.3722 [cs.CL]

[5] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, Jeffrey Dean. "Distributed Representations of Words and Phrases and their Compositionality". arXiv:1310.4546 [cs.CL]

[6] Deeplearning4j: Word2Vec. http://deeplearning4j.org/word2vec.html

[7] word2vec: Tool for computing continuous distributed representations of words. https://code.google.com/p/word2vec/

[8] Apache Pig. http://pig.apache.org/docs/r0.15.0/start.html

[9] Apache Spark. http://spark.apache.org/docs/latest/programming-guide.html

[10] Sentiment-aware tokenizer. http://sentiment.christopherpotts.net/tokenizing.html#sentiment