# Meme Tracking in Scale with MapReduce

Hohyon Ryu
School of Information
University of Texas at Austin
hohyon@utexas.edu

Jae Hyeon Bae
Computer Science
University of Texas at Austin
metacret@gmail.com

Nicholas Woodward
School of Information
University of Texas at Austin
woodward.nicholas@gmail.com

## ABSTRACT

This article is to report our project progress for Fall 2011 Data-Intensive Computing for Text Analysis class. We have completed preprocessing of the Blogs08 collection by removing all HTML tags, boilerplate text, non-English documents, and duplicates. After the preprocessing, aproximately 4 million popular common phrases were extracted using an enhanced version of the technique suggested by [?]. The extracted phrases are clustered in 2 million meme groups. The meme groups will be ranked automatically using Support Vector Machine, based on the crowdsourced labels obtained from an Amazon Mechanical Turk.

## Categories and Subject Descriptors

H.4.m [**Information Systems Applications**]: Miscellaneous; H.5.4 [**Information Interfaces and Presentation**]: Hypertext/Hypermedia

## General Terms

MapReduce, Blogosphere

## Keywords

Keyword Extraction, Data-intensive Processing, Implicit links in Blogosphere, Meme tracking

## 1. INTRODUCTION

Web logs, also known as blogs, are a very important and interesting source of diverse information. They provide insight into the culture of the society in which they exist. People search blogs to find out how to cook, what to do, what to see, and to discover what others think. Because blogs are typically written in plain language by individuals enthusiastic for their interest, they may be more easily understood than longer, more-detailed news articles and cover a wider personal range of topics. As blogs are highly sensitive to fast-changing temporal issues, finding and tracking memes, pieces of information that travel across blog posts, we can mine variety of issues and fads in the blogosphere. In this project we will extract memes from blogs, cluster them by their similarity, and rank the meme clusters using crowdsourcing-powered machine learning.

## 2. RELATED WORK

## 3. MEME EXTRACTION

### 3.1 Test Collection

BLOGS08[1] is a TREC[2] test collection that crawled Web Logs from Jan/14/2008 to Feb/10/2009. It consists of 3 components: feeds, permalink documents, and homepage documents. We use only permalink documents that contain 28,488,766 documents and whose uncompressed size is 1445GB. Because the collection contains many duplicates, non-English blog posts, advertisement, and Blog website frameworks, we will use NLTK and Decruft to extract English blog contents without boilerplate text.

### 3.2 Preprocessing Blogs08

Preprocessing of the Blogs08 Collection was completed using Python with NLTK and Decruft packages. Decruft extracts only meaningful content from a blog page, filtering out navigation links, advertisements, sidebar contents, and links to other sites. NLTK uses a language identification module to determine if the extracted content is written in English. Blogs08 also contains a tremendous amount of duplicate documents and documents without a significant amount of text. We set a lower-bound of 5 words and removed duplicates, also using Python. After all these preprocessing steps were completed, extracted contents of a blog post are written into a single line per document for the MapReduce meme extraction process. The number of Decruft-applied, deduplicated, and HTML stripped English blog posts is 15,097,266 (43 GB).

- English only
- Decruft
- Deduplication
- HTML Strip

[1]http://ir.dcs.gla.ac.uk/test_collections/blogs08info.html
[2]http://trec.nist.gov/

## 3.3 Meme Extraction with MapReduce

Preprocessed Blogs08 data is comprised of key-value pair where the key is document number(doc_id) and the value is the preprocessed content of the blog posts. Using Hadoop Streaming, Python MapReduce code extracts common phrases and group documents by them. Overall algorithm design follows that of [?]. However, because of a scalability issue, lower bounds and upper bounds limit the output size of the data, and other optimizations are used such as stop word ratio check and preposition trimming. The three steps of MapReduce pseudo-code for Meme Extraction are illustrated in the following pseudo codes. The first MapReduce process splits the input text into sentences and extracts trigrams and their indexes within the document. The reducer removes the common phrases(*trigrams*) that appear less than 5 times or more than 300,000 times.

**MapReduce 1:**

```
Mapper 1:
for (doc_id, text) in [input file]:
    sentences ← split text by sentence boundary

    for sentence in sentences:
        index ← 0
        for trigrams in sentence:
            index++
            emit (trigrams, (doc_id, index))

Reducer 1:
lower_bound ← 5
upper_bound ← 300000
for (trigrams, [(doc_id, index)]) in [mapper 1 output]:
    len ← count([(doc_id, index)])
    if lower_bound < len < upper_bound:
        emit (trigrams, [(doc_id, index)])
```

The second MapReduce process sorts the trigrams and indices by document to concatenate adjacent trigrams in MapReduce 3:

**MapReduce 2:**

```
Mapper 2:
for (trigrams, [(doc_id, index)]) in [reducer 1 output]:
    for (doc_id, index) in [(doc_id, index)]:
        emit (doc_id, (trigrams, index))

Reducer 2:
for (doc_id, [(trigrams, index)]) in [mapper 2 output]:
    emit (doc_id, [(trigrams, index)])
```

The third MapReduce process concatenates the adjacent trigrams within a blog post. Mapper 3 concatenates adjacent trigrams into a meme. Before emitting the memes, a function trimPrepositions trims off prepositions at the beginning and end of the meme, and a function stopwordRate checks how many of the words in the meme are stop words. If stop words constitute more than 65% of the words in the meme, the meme is ignored. In the reducer 3, phrases that are too short ($\leqq 5$) or too long ($\geqq 200$) are ignored as they are often timestamps, advertisement, or boiler plate text. The final MapReduce is illustrated as follows:

**MapReduce 3:**

```
Mapper 2:
for (doc_id, [(trigrams, index)]) in [reducer 2 output]:
    prev_index = 0
    meme = “”
    for (trigrams, index) in [(trigrams, index)]:
        if index − prev_index ≦ 3:
            meme += trigrams[3-(index − prev_index):]
        else:
            meme=trimPrepositions(meme)
            if stopwordRate(meme)<0.65:
                emit(meme, doc_id)
            meme=trimPrepositions(meme)
            meme.append(meme)
            meme=trigram
        prev_index = index

    # clean up for the last one
    meme = trimPrepositions (memes)
    if stopwordRate(meme)<0.65:
        emit(meme, doc_id)

Reducer 3:
for (meme, [(doc_id)]) in [mapper 3 output]:
    if 5 ≦ len(meme) ≦ 200:
        emit (meme, [(doc_id)])
```

The final output is key, value pairs of a meme and the list of the document ids that the meme appears. In total, 4,610,100 memes are extracted from the TREC Blogs08 collection.

## 4. MEME CLUSTERING

### 4.1 Mahout LDA

### 4.2 Yahoo LDA

### 4.3 Mahout Canopy

### 4.4 Indexed MapReduce Canopy

Since the form of memes evolve over time across different media [?], similar memes need to be clustered together. For example, “lipstick look good on a pig,” “lipstick on the pig,” and ”lipstick on this pig” need to be seen in the same cluster. A canopy clustering algorithm [?] is used for this meme clustering. However, since we are clustering more than 4.6 million memes into many clusters of very fine granularity, different strategies should be adopted. Since Canopy is basically a single pass approach, the distance between every meme and every cluster needs to be calculated in $O(n^2)$ computational complexity. The meme clustering task is very different from an ordinary document clustering task as the number of documents is extremely high while the length of each document is usually fewer than a dozen of words. We devised the following optimizations and modifications to efficiently cluster 4.6 million memes:

- Binary vector: A vector is simply a set of words in a meme or a centroid. The set operation is O(1) while list operation is O(n). Also word order is ignored by this method.

- Centroid indexing: The centroid is indexed by the words it includes so that when a meme is compared to only with the cluster centroids that include common words.

- New similarity coefficient: Jaccard similarity coefficient

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

is inappropriate for meme clustering. We compare a meme with a cluster centroid, and when a cluster centroid is long, short memes cannot have high coefficient scores. Therefore, instead of dividing by the union, we divide the number of common words by the average length of the two vectors:

$$memeSim(A,B) = \frac{|A \cap B|}{(|A| + |B|)/2}$$

The similarity score should be higher than 0.6 to be clustered into a given centroid.

- Stop word removal: The memes should be compared by meaningful words instead of stop words. Otherwise, “You can put lipstick on a pig” and “You can put a book on the table” would be clustered together. Using an extended stop word list that includes month names and weekday names, we excluded stop words from the similarity computation.

- Distributed computation: To cluster 4.6 millions of memes efficiently, mapreduce-based distributed Canopy approach, explained in Apache Mahout Canopy Clustering[3], is used. The mapper takes in a partial list of the memes. Using canopy algorithm, it generates cluster centroids. The reducer does the same clustering process with the aggregated centroids. The pseudo code of the MapReduce process is illustrated as follows:

```
Meme-clustering:
Mapper:
$CIndex \leftarrow newHashMap$ # Centroid Index
$C \leftarrow newList$ # Cluster Centroids
for $meme$ in $memes$:
    # Initialize $C$ and $CIndex$
    # 0 is the index of the first cluster
    if $C$ is empty:
        # set() makes a set of words in a meme
        $C$.append( set(meme) )
        for $word$ in $meme$:
            $CIndex[word]$.add(0)

    # Get candidate centroids from $CIndex$
    $Candidates \leftarrow newSet()$
    for $word$ in $meme$:
        for $c$ in $CIndex[word]$:
            $Candidates$.add($c$)

    $flag \leftarrow False$
    for $c$ in $Candidates$:
        if memeSim($meme$, $C[c]$)>0.6:
            $C[c]$=$C[c]$ + set($meme$)
            $flag \leftarrow True$
            for $word$ in $meme$:
                $CIndex[word]$.add($c$)
            break
    if $flag == False$:
        $C$.append(set($meme$))
        $c$=len($C$)-1
        for $word$ in $meme$:
            $CIndex[word]$.add($c$)
emit($C$)

Reducer:
    The reducer is indentical to the mapper except that
    the input is the centroids($C$) instead of
    the memes($memes$).
```

[3]https://cwiki.apache.org/MAHOUT/canopy-clustering.html

## 5. MEME RANKING

Now that we have successfully clustered the memes, we still have many things to do in order to find valuable and meaningful information from the meme clusters. Essentially, we can classify every meme cluster as informative or not because there are many meaningless sequence of words such as ”2 1956 jamboree tues jan 3 1956 jamboree wed,oct 5th wed oct 4 tues“. If we extract some statistical features from meme clusters and build the training data indicating informativeness, we can predict informativeness of each meme cluster. Finally, if we build the model of classifier as a regression problem of probability of informativeness given a meme cluster, we can bring order into a set of meme clusters with the probability values. Currently, we are expecting the following features as discriminant to decide a meme cluster’s informativeness.

- the number of memes in the cluster
- average meme length
- average number of documents in the cluster
- average number of sources in the cluster
- meme cluster cohesiveness
- time span information of blog documents containing memes in the cluster
    - the number of days blog documents were written
    - the number of documents per day

Meme cluster cohesiveness can be defined as

$$\frac{the\ number\ of\ unique\ words}{the\ number\ of\ words}$$

which means how similar memes in the cluster are each other.

### 5.1 Building Training Set at Amazon Mechanical Turk

Ranking the meme clusters based upon their meaningfulness or informativeness as described above is a difficult and largely subjective task. Consequently, using machine learning for this task is highly problematic. Instead, we are implementing an Amazon Mechanical Turk that will ask users to answer a set of basic yes/no questions about sample meme clusters to determine their meaningfulness. Users will be asked questions such as "Is this a meaningful chunk of information?" and "Is this a common expression that could potentially appear elsewhere?" for approximately 1500 meme clusters. The composite score of each meme cluster will determine if it is meaningful or not. Using these labels we will build a training set of memes that can serve as input to the Support Vector Machine for automatic meme ranking.

### 5.2 Automatic Meme Ranking using Support Vector Machine

We will use Support Vector Machine for meme ranking. Utilizing the LIBSVM package[4], we will train C-SVM and regard each probability value as the ranking score. For better performance, we should scale feature values and use grid search to find suitable parameters.

## 6. EVALUATION

Based on the labels collected with the Amazon Mechanical Turk, the performance on SVM-based ranking will be evaluated.

## 7. CONCLUSION

## 8. LOGISTICS

### 8.1 Division of Work

- Hohyon Ryu: Data Preprocessing, Meme Extraction, Meme Clustering
- Jae Hyeon Bae: Theoretical Background, SVM-based Meme Ranking
- Nicholas Woodward: Theoretical Background, Crowdsourcing the meme cluster evaluation, Mock-up Webpage Build

### 8.2 Timeline

- Oct/6: Preprocessing - Done
- Oct/13: Meme Extraction - Done
- Oct/20 & 27: Meme Clustering - Done
- Nov/3 & 10: Crowdsourcing
- Nov/10: SVM-based Meme Cluster Ranking
- Nov/17: Finalizing Paper
- Dec/1: Presentation

[4]http://www.csie.ntu.edu.tw/ cjlin/libsvm/