Meme Tracker

Social Network Generation and Ranking

Eshwaran Vijaya Kumar
Dept. of Electrical & Computer Engineering
The University of Texas at Austin
eshwaran@utexas.edu

Saral Jain
Dept. of Computer Science
The University of Texas at Austin
saral@cs.utexas.edu

Prateek Maheshwari
Dept. of Computer Science
The University of Texas at Austin
prateekm@utexas.edu

Categories and Subject Descriptors
H.4 [**Information Systems Applications**]: Machine Learning, Text Mining

General Terms
Machine Learning, Text Mining, Mapreduce

1. INTRODUCTION

The term meme was coined first by Richard Dawkins in [**?**] to identify cultural ideas that replicates, mutates and moves across in a society. The internet as a social ecosystem provides an interesting space where memes can spread rapidly, mutate and die out in a matter of weeks. The purpose of this study is twin fold: We would like to design an information retrieval system that allows us to identify short phrases of text, try to recover and analyze the latent social network that could potentially have acted like a flow mechanism for the memes. As part of this, we explore some ways of modeling a meme, most of which are tied together by the fact that they consider that a meme is a short phrase of text.

The first section of the project is to model a meme. Our modeling techniques are inspired by prior work in this field namely by Leskovec et. al in [**?**] and Kolak and Schilit in [**?**]. Both works make fundamental assumptions which impose significant assumptions on the structure of the latent social network uncovered.

Leskovec et. al. [**?**] have designed a scalable system that emphasizes on the fact that a meme evolves and exhibits rich variations and try to study the latent social network or news cycle that is generated when temporal information is incorporated. Their model of a meme while accurately noting the fact that a meme might very well mutate makes a strong assumption that every such meme is enclosed in quotation marks. Another assumption that they make is that the evolution of a meme over time can be tracked by the increase in length of the meme. It is not clear why this assumption should be true in general, or why the inverse shouldn't happen. While the first assumption might be reasonable in the well formatted world of news sites, it doesn't appear to be so reasonable in the less grammatically inclined world of blogs. We relax this assumption and consider that a meme is less than or equal to a single sentence that is demarcated by a period. While, this assumption is not without its flaws, it helps us to demarcate a document or a site into smaller pieces from which phrases that appear in repeated documents can be identified. We then employ a variety of natural language processing techniques like word overlap and language modeling based similarity measures to identify similar chunks of text that appear in documents. The interesting thing about this assumption is that we feel that it will discover memes that are similar to each in that most of the characters in the meme match while there are a few characters that are changed. For example, the meme, "Lipstick on a pig" will be identified as a mutation of "Lipstick on a hog".

Kolak and Schilt [**?**] work on a system for identifying links between different books in an online library that gives us an inspiration for another meme model. Their approach makes an assumption that books that share quotations are related to each other which falls within Dawkins' original definition of a meme. They go on to make an assertion that there are situations where different parts of a single quotation might be shared between different books. This is analogous to considering scenarios where parts of the same idea gets utilized in multiple sites which ideally are part of the same latent social network that we are trying to uncover. For example, we would like to identify that"Lipstick on a pig" could be part of super meme that discusses the book: "Lipstick on a Pig: Winning In the No-Spin Era by Someone Who Knows the Game" [**?**].

Once the various meme-graphs are generated, evaluation of the quality of the graph needs to be done. In the absence of any "gold dataset" that we can utilize, we propose an indirect method of evaluation: We design a search engine that we hope will do better for the purpose of meme identification. The results returned by the search engine are ranked using the meme-graph in addition to the page rank algorithm. As a baseline, we utilize the fact that we have hyperlink information between the blogs to perform rankings.

This document serves two purposes: We describe and illustrate our project, explain progress made and talk about

challenges that we are currently facing and steps that we plan to take to overcome those. We also illustrate a plan of action to tie up the several components of the project together. It is organized as follows: We first describe two ways of generating a meme-graph and have a discussion on what the ways enforce on graph generation. We then devote some time to talk about our method for evaluation of the memes. We then terminate with an outline of the steps that we plan to take over the next few weeks to complete the project successfully.

2. MEME-GRAPH GENERATION I

2.1 Extraction of a Meme

The first part of the problem is that of automatically identifying and tracking memes from a corpus of documents that are potentially linked to each other by memes.

Kolak and Schilt [**?**] describe a scalable technique for generating quotations that are shared between pairs of documents that we employ for the meme generation portion. We describe this in the next section and talk about implementation challenges that we face.

2.1.1 Sequence Generation Overview

Given a "clean" corpus of documents (blog posts), we can generate key-value pairs where a key is a contiguous sequence of characters and value is a pair of documents that contains that key as follows: We first parse each blog posts and generate shingles composed of a few tokens where every token is a space delimited collection of characters. These shingles are used to build an inverted index where the keys are shingles and value is a bucket list. The bucket list is a data structure that is an array of buckets where each bucket contains a unique identifier for a blog post and a position marker which identifies the start of the shingle in the document. The next phase is to merge shingles that are contiguous to each other in a pair of document. This is done as follows: We walk through each document, generate all the shingles that exist in that document. We create a set of sequences as follows: We start from the first shingle in the source document and create a set of "active" sequences where each sequence has a unique identifier marking a document that shares this shingle and the position where this shingle is present in the document. Then for successive shingles, we iteratively walk through the sequences to determine if the next potential shingle in a given sequence is present in the successive shingle's bucket list and decide to either conclude the sequence or keep continuing the building of the sequence based on that result.

2.1.2 Implementation and Challenges

The first implementation challenge that we have worked on is the cleaning of the dataset to strip away HTML tags. We had initially built an in-memory python processor which utilized beautiful soup to strip away HTML. This unfortunately turned out to be too slow (even though all that needs to be done is a linear pass through the dataset) and we instead built code to do this in MapReduce using an equivalent library.

The second challenge that we are currently working on is trying to efficiently scale the sequence generation phase in Map Reduce. Our current approach involves a sequence of two jobs: The first job constructs shingles from the cleaned up text and generates a shingle table where the values are a list of buckets. In our second job, the map phase involves transmitting as key a bucket and value the entire bucket list. A partitioner ensures that all the keys corresponding to a single document go to a single Reduce task. Within a single reducer, we compute all possible sequences that are shared by a "source document" which was the key in the reducer and all other documents.

There are several issues with this approach that emerged (and that we are still working on) as we were trying to scale it up: We faced an initial issue with an overwhelming large number of out of memory errors. Our first approach had utilized text strings to represent the bucket lists, we have optimized that by using array of pairs which are wrappers around Integers. The next issue is the overwhelmingly large amount of intermediate data that is generated in the form of the bucket lists themselves: This we are trying to solve by using a splittable compression technique such as LZO. Since most of our testing is done on Amazon EC2, we predict that developing the AMI to sort this issue should be doable within the next week.

2.2 Grouping Of Sequences

An intermediate step that could potentially create a more richer graph structure is to generate a "Super-meme" that takes matches between portions of a meme (sequences) to generate a group of sequences that share some partial overlap and can be merged to form a single meme. The approach utilized in Kolak and Schilt [**?**] is as follows: For a given document, order all the sequences by start position in the document. We then initialize a group that contains the first sequence. We then iteratively merge any sequence that has an overlap with the group's current sequence and add all the corresponding documents into the group. A new group is created when we find that the current sequence has no overlap with the previous group. The end result of this Map Reduceable phase is to create groups as keys and values as an array of document identifiers that share that group.

3. MEME-GRAPH GENERATION II - DIRECT STRING SIMILARITY

The approach taken by Kolak and Schilit [**?**] , i.e., generating and extending shingles and then grouping the final results, has a few drawbacks. For one, they set the minimum size of shingles to 8 to reduce the amount of intermediate output. A long shingle length would require sentences containing meme to have a significant and exact overlap, an assumption that is not necessarily true in practice. For example, the following two sentences have the same information content, but would not be detected by the shingles approach for a shingle length greater than 5 (corresponding to the ngram "at the Intel Developer Forum"). One way to overcome this limitation would be to use a smaller minimum size for shingles. However, setting the shingle size lower would increase the amount of intermediate output and the size of each bucket list, and will also increase the amount of time spent extending the shingles for each document in the next phase. Also, in the shingling approach, we generate the shingles irrespective of sentence boundaries, which

generates many redundant shingles.

```
The processors were announced in San Jose at the In-
tel Developer Forum.

The new processor was unveiled at the Intel Developer
Forum 2003 in San Jose, Calif.
```

Another problem is that if the different variations of the meme have been paraphrased, the shingling approach would limit the detectable meme to the longest sequence that two sentences share. Furthermore, if the memes are generated by following a shingling approach, we need to group the memes that share common subsequences into their common "superstring" by grouping the shingles based on the words they share. This problem is equivalent to finding a hamiltonian path in a graph of the meme strings in which the strings are the nodes and all memes that share a substring are directionally connected. The problem of finding Hamiltonian paths in a graph is known to be NP-hard and the solution will not necessarily give accurate results since the meme strings might have an ngram overlap without being originally related.

To avoid these problems we propose an alternate approach to generating and grouping the memes based on a pairwise string similarity computation. For this approach, we consider a sentence to be the atomic unit that conveys information related to a meme. The idea behind the process is to shortlist the strings from the document collection that might be related to each other and compute one or more syntactic/semantic string similarity measures for those pairs of strings. If the score is above a certain threshold (to be determined experimentally), they will be considered to be the same meme. The process is described below in more detail.

3.1 Preprocessing for Shortlisting Pairs

In the pairwise string similarity approach described above, we need to shortlist the strings that might potentially be the same meme and compute their similarity. Given enough computing power and time, we could compute the similarity of all the strings in the corpus with each other. However, this is not practically feasible given the large number of strings, and hence we shortlist the strings we will be computing the similarity scores for. To shortlist the strings, we make an assumption that similar strings share at least one ngram (of a size to be determined experimentally) and vice versa, that each pair of string that shared the same ngram is a candidate for similarity computation. We construct an inverted index from ngrams to a postings list containing strings that contain that ngram from the document collection. If we take a cross product of the postings list for each ngram with itself and keep unique string pairs, we get a list of the string pairs that are candidates for similarity computations.

3.2 Experimental Results

The experimental results from an analysis of the effect of ngram size used for shortlisting pairs on the efficiency and effectiveness of shortlisting have been reported in another project report by one of the authors [cite it?]. The major conclusions from the analysis are that the accuracy of retrieval decreases linearly while the amount of output generated decreases exponentially for values of n >= 3. The exact value of n used for shortlisting the pairs depends on the size of the dataset, but we can increase the value of n if the amount of intermediate data grows too large without sacrificing too much accuracy.

3.3 Grouping

The output of the previous phase is, for each string, a ranking of similar strings by similarity scores. Using this output we can create a grouping of similar memes by computing a closure on top n similar strings in the following way:

1. Select a string Si whose group is to be computed. Choose a value of n. Create an empty grouping set. 2. For string Si, select the top n strings ranked by their similarity to Si. Let these strings be Sij, 1 <= j <= n. 3. If Sim(Sij, Si) > a threshold, add Sij to the grouping set. Mark Si as visited. 4. If all strings in the set are visited, stop. Else, for each string Sij in the set not yet visited, repeat step 2.

The threshold is required to remove strings with low similarity scores from groups. If we repeat this process for each string, we have a grouping of the memes discovered in the first phase.

4. GRAPH GENERATION

[Insert discussion about the possible effect of grouping on meme graph and pagerank here]

Depending on whether or not we group similar memes together, we can generate the meme graph in two different ways. Both approaches are discussed below.

4.1 With Grouping

With either of the two approaches for discovering memes, we have two ways of constructing the meme graphs for computing the pagerank. With both approaches, we can either group similar memes together, or we can use individual memes to induce a graph.

If we have a grouping of similar memes, we construct the edges between blog posts that contain those memes, and the direction of the edge is determined by the time stamps of the blog posts. The edges are directed from the post that appears earlier to the post that appears later, following that idea that the newer post borrows from the older one, and hence is less important. From a grouping of posts that contain similar memes, we construct a single directed path from the newest post in the group to succesively older ones. (Should we create an edge from a new post to ALL other older posts, following the idea that oldest will have the most precedence (since pagerank gives emphasis to incoming links, and reduces emphasis for outgoing links), followed by the 2nd oldest and so on? If we have a single edge than the weights computed by page rank would be very small for intermediate posts since each intermediate post would have one inlink and out outlink for a meme. On the other hand, each post can contain multiple memes, although that would be much rarer).

Another approach to construction a graph is not grouping the memes together and instead creating a single edge for each meme string, directed from the newer post to the older post that shares that string.

Irrespective of whether grouping is done or not, we generate an adjacency list as follows: We take a pair or array of documents and impose directed edges between them. The direction of the edge is determined by the time stamp of the blog. We expect that this will be possible in an in-memory program where an index that contains document ID and time stamp is loaded and utilized to generate the edge directions.