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

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.

A meme is a short phrase which originates, travels and mutates in a social network. The meme is a potentially undiscovered path through which information flow takes place in a social network. We would like to automatically extract short phrases which are present in some possibly mutated form in a group of blog posts, group similar phrases together to identify nodes (blog posts) that share edges. This meme-graph is then evaluated as follows: We compare the performance of the meme-graph as a ranking mechanism when compared to a baseline social-graph, the hyperlink graph exploited successfully by Google.

The project has several phases which have related works that we review and incorporate in our work: Prior work that can be applied to solve our problem has been looked at by Leskovec et. al in [**?**] and Kolak and Schilit in [**?**]. There are several ways to define a meme which impose fundamental assumptions on the design of the entire system.

Leskovec et. al.[**?**] make two fundamental assumptions: 1) They assume that every meme is demarcated by quotation marks. While, this assumption might be true for a subset of the memes, it would result in a system that fails to capture a large number of memes that don't possess this characteristic. 2) They assume 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. However, for the purposes of evaluation we impose a more relaxed assumption in one of our empirical models: We make the assumption that a meme is one single sentence demarcated by a period and make no assumption regarding relation of length and temporal information. We then employ string similarity based measures to group memes. (TODO: Prateek write more stuff) 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 Schilit [**?**] approach a similar problem which can be applied to solve the problem of meme extraction: They generate shingles from books and use the shingles to generate a sequence of contiguous characters that are shared between a pair of documents. Grouping in this case can identify the fact that some blogs may use parts of the whole meme and can identify situations where the statement "Lipstick on a pig" has been split due to sentence structure changes as for example "Lipstick on a" and "pig" as separate chunks in different documents.

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.

2.3 Graph Generation

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.

3. 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. (Is this true?)

```
The processors were announced in San Jose at the Intel 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(or complete?) 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 thise 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

Grouping for string similarity