A Comparison of Three Document Clustering Algorithms: TreeCluster, Word Intersection GQF, and Word Intersection Hierarchical Agglomerative Clustering

Kenrick Mock
9/23/1998
Business Applications
Intel Architecture Labs

Abstract

This work investigated three techniques to automatically cluster a collection of documents: Word-Intersection with GQF, Word-Intersection with hierarchical agglomerative clustering, and TreeClustering. The Word-Intersection algorithms have been previously described in the literature while the TreeClustering technique is novel to this work. The TreeCluster algorithm idea comes from rule induction techniques and is used to generate a shallow tree of clusters that a user can browse. This algorithm is also O(n) when used with a fixed tree depth, as opposed to $O(n^2)$ as the other two algorithms. Experimental results on a collection of Mail and Web documents indicate that the agglomerative clustering algorithm performed the best, but also the slowest. The GQF algorithm performed well on the Mail domain, but performed poorly on the Web domain without tuning to its heuristic. The TreeCluster algorithm performed reasonably well on both domains, and was also the fastest algorithm out of the three algorithms tested.

Introduction

Many search engines and information retrieval system return a list of documents that match a query. As the number of available documents increases, it has become a difficult task to not only search for relevant documents out of the global document pool, but also to search through the list of documents returned from a search query. Results from a search may return dozens or hundreds of matches that can be nearly as daunting to search as the original document pool. In addition to the search problem, it is also difficult for a user to browse the collection of documents. Since they are unstructured, the user must known the proper search terms in order to find a particular document of interest. Otherwise, the user is relegated to scanning through a large number of documents by hand.

One approach that has been applied to this problem is automatic clustering. This approach is useful when no a-priori structure has been assigned to the document collection, or when it is too expensive to manually categorize the documents. The primary goal of automatic clustering is to determine natural and logical groupings of documents without user intervention. The list of documents is then displayed in terms of these groupings in order to ease the task of browsing through the collection.

This technique is typically applied to either a database of documents or to the results of a search query. When used after a search query, users will typically want immediate

results. Consequently, a secondary goal of the clustering method is that it must run quickly, typically within seconds. Additionally, if hundreds or thousands of documents are returned, the clustering method must be scalable with respect to the size of the document collection.

Prior Work

Hierarchical Agglomerative Clustering

A common technique to cluster documents is the hierarchical agglomerative clustering (HAC) approach (Willett, 1988, Voorhees, 1986). In this approach, each document in the collection or list is treated as a cluster and added into a pool of available clusters. Consequently, with *n* documents, there are initially *n* clusters. Next, all pairs of clusters in the pool are compared and the most similar pair is selected. Document/cluster similarity is typically computed using a metric such as the Cosine, Dice, or Jaccard formulas (Rorvig, 1998). The most similar cluster pair is then merged into a single cluster, and added back into the pool of clusters. The process continues until some stopping condition is reached. Typical stopping conditions include a threshold for the number of remaining clusters (e.g., stop if 4 clusters remain), or a threshold for the required similarity between clusters (e.g., stop if the two most similar clusters are not very similar).

The HAC algorithm has been applied in many different ways. In all cases, the base algorithm is the same. The differences lie in the method used to compute document similarity and the stopping criteria. While this algorithm produced some of the best clusters in our experiments, it suffers in terms of speed and scalability. The process of merging pairs of clusters results in a $O(n^2)$ runtime that is often unacceptable in moderately sized document collections. Additionally, the HAC algorithm is suspect to creating either elongated clusters or many small clusters. Elongated clusters are large sets of clusters where two documents within the cluster may have nothing in common. Conversely, the algorithm may also produce many small clusters with only two or three documents within the cluster. Neither case is particularly desirable.

The HAC algorithm implemented in these experiments used the similarity metric of word intersection. Word-intersection clustering (Word-IC) was defined by Zamir, et. al. in the context of snippets from web documents (Zamir, et. al., 1997). They actually found that Word-IC clustering performed better than the more clustering using the common Cosine Group-Average similarity function when operating upon noisy snippets of web documents.

In Word-IC, similarity is determined by maximizing *cohesion*. Cohesion is simply defined as the number of words that intersect when two clusters are unioned together. The termination criterion used in our implementation was to stop when the most similar pair of clusters had a cohesion less than 2. That is, stop if one or no words were common between clusters.

Global Quality Function Clustering

One of the innovations introduced by Zamir et. al. is a global quality function (GQF) that is used as a heuristic to guide the clustering process. In Word-IC using GQF, a heuristic function evaluates the goodness of the current set of clusters. Just as the HAC algorithm, the GQF algorithm begins with each document defined as its own cluster. For each pair of clusters, the algorithm then computes the GQF heuristic function if that pair had been merged. The pair of clusters that best maximizes the GQF heuristic function is then merged, and placed back into the pool of clusters. The process repeats until there is no increase in the GQF heuristic. This new process has a well-specified termination criterion and that may make it better than HAC.

Key to the algorithm is the definition of the heuristic function, GQF. The resulting clusters depend highly on how this function is defined. Zamir defined the GQF to balance the number of clusters vs. the cohesion of an individual cluster. In general, we want a low number of clusters all with high cohesion. However, this results in a tradeoff since initially all the clusters have very high cohesion, but are all singletons. In most cases, merging clusters dramatically lowers cohesion but only slightly decreases the number of clusters. To address the tradeoff, Zamir defined GQF as:

$$GQF(ClusterSet) = \frac{f(ClusterSet)}{g(|\ ClusterSet\ |)} \sum_{c \in ClusterSet} s(c)$$

In this definition, *f* is a function that is proportional to the fraction of documents in non-singleton clusters (clusters with one document). That is, clusters with one document are bad, so we want to maximize the number of documents that are in non-singleton clusters. *g* is a function that increases with the number of non-singleton clusters and attempts to minimize the number of overall clusters. *S(c)* measures the score of each cluster, where the score is based on the cohesion.

More formally, each function is defined as:

$$f(ClusterSet) = \frac{documents\_in\_non\_singleton\_clusters}{total\_number\_documents}$$

$$g(|\ ClusterSet) = \sqrt{\#\_non\_singleton\_clusters}$$

$$s(c) = |\ c\ | \times dampened\_cohesion(c)$$

Zamir derived these formulas via experimentation and found that they returned good results. One detail that was omitted is the definition of dampened cohesion. In this paper, we tried using direct cohesion (# of common terms in the cluster), the square root of the direct cohesion, and the log of the direct cohesion. We found that using the log of the cohesion gave the best results, as the cohesion may be a large number that dominates the heuristic if not scaled down.

Similar to HAC, Word-IC with GQF also runs in $O(n^2)$ time. One advantage to this approach is that all documents in a cluster must share common words. These words can be used as a centroid for visualization purposes that describe the contents of the cluster to a user browsing the list of clusters.

Term Reduction for HAC and GQF

One major difference between this work and the Zamir study is the terms used from each document. In Zamir's work, each document was comprised of 40 word *snippets* from web documents. These snippets were extracted using the MetaCrawler search engine. In this work, we used the full text of the document and reduced the term set through a stop list and statistical means.

Rather than use all terms in a document, this work only used terms that are likely to be relevant. First, stop words were removed from each document. Then, *tf-idf* values were assigned to each term and the terms were sorted by *tf-idf* value. Terms with a *tf-idf* less than 0.05 were discarded, since they likely are not relevant. Terms with a *tf-idf* equal to 1 were also discarded, since these terms are unique to only one document. Consequently, they would not contribute to the cohesion score when documents are merged.

After processing each document in this manner, the top 200 terms was retained. Document similarity and the clustering process operated only upon the terms retained within each document. These terms should be fairly relevant to the document, but will also contain no unique terms and few terms that also appear in other documents.

Inductive TreeCluster

Based upon the HAC and GQF methods, along with prior experience in rule induction and how people create their own clusters, we designed a novel algorithm that attempts to "induct" clusters in a manner similar to the induction of rules. The idea is to generate a conjunction of conditions (i.e., a rule), typically composed of 1 to 3 terms, that cover a number of documents. For example, the terms may be ("algorithm" AND "genetic" AND "agent") to describe a set of documents about genetic algorithms. The documents that match the rule are turned into a cluster. The process is repeated to generate a tree of clusters. The intent is to quickly generate a shallow tree of clusters that can be hierarchically browsed by the user. A sample of the results is shown in figure 1.

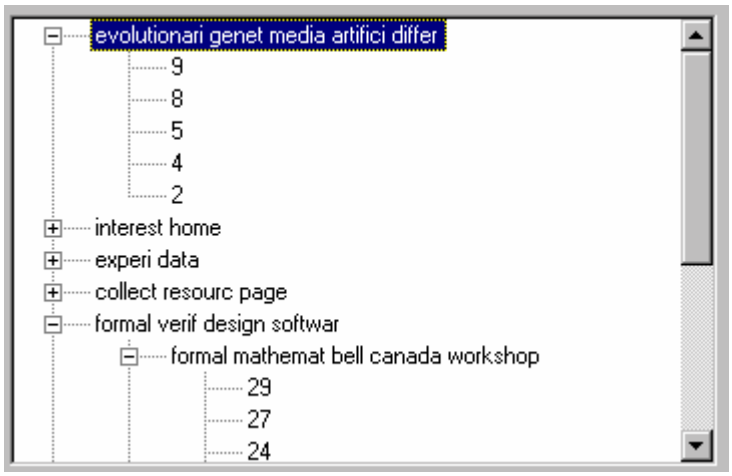

Figure 1: Sample TreeCluster.
Data is organized hierarchically by common keywords. In a real application, subject lines would be displayed instead of document ID numbers.

Term Reduction

The success of this algorithm depends upon a good set of terms. For example, if common terms are used to generate the conditions, there will likely be a single non-informative cluster that is based upon the word "the". Consequently, the first step of the algorithm is to determine what terms to use for generating the rules. We wanted to exhaustively search the space of rules that can be created using a conjunction of up to three terms. To be tractable, this requires a relatively small number of terms to keep the number of rules down. As a result, we generated a total of 100 terms to use in a global term list to generate rules.

The first step that we implemented to determine the term list was identical to the term reduction process used for the GQF and HAC algorithms. For each document, the tf-idf values were computed and candidate terms between 0.05 and 1 were retained in a global pool of terms. This phase removes most common terms, but perhaps not all. The next step was to sort this pool of terms by their global document frequency. This is not the total term frequency, but the number of documents that contain each term. Consequently, a term that is in all *n* documents would be at the top of the list with a document frequency value of *n*. Starting with the most frequent terms and working down to the least frequent terms, a term was added to the final term list if $doc\text{-}freq(t) < PERCENTMAX$ and $doc\text{-}freq(t) > 2$. Only the first 100 terms were selected.

The latter expression guarantees that a term must appear in at least three documents. This restricts the list to terms that have a chance of creating multi-document clusters. The former expression is used to control the creation of elongated clusters. In our implementation we set PERCENTMAX to 0.25. This meant that no term was added to the global set of terms if it appeared in more than 25% of all documents. This

guarantees that no cluster will be created based upon a single term that contains more than 25% of the documents. This threshold could be varied depending upon the total number of clusters desired.

TreeCluster Algorithm

Once the global term list has been determined, the process of creating rules can begin. Unlike traditional rule induction algorithms that proceed from the general to the specific, this algorithm must proceed from the specific to the general. The strategy is to build the cluster tree from the bottom-up by first finding the most specific clusters possible. Grouping the more specific clusters together creates more general clusters until a top level of clusters is created.

The most specific set of rules we generate contains three terms. For example, the rule ("algorithm" AND "genetic" AND "agent") is more specific than ("algorithm" AND "genetic"). If d is the number of terms in our candidate rule and n is the total number of terms available, then initially the total number of rules possible is n choose d:

$$TotalRules = \binom{n}{d} = \frac{n!}{d!(n-d)!}$$

For n=100 and d=3, this means that initially there is a total of 161,700 rules to examine. Although large, this is a fixed number as long as n and d are kept constant. We used a maximum value of 3 for d, which corresponds to the depth of the generated cluster tree.

After the rules have been generated, each rule is applied to all clusters. Initially, all documents are made into a singleton cluster. The rule that covers the most clusters is removed from the set of rules. Similarly, all clusters that are covered by the rule are removed from the set of available clusters and merged into a single new cluster. This new cluster is then added into a new pool of clusters for the next iteration, and its set of terms is set to the common terms among all documents in the cluster. The process repeats until no available clusters remain or the best rule covers only a single cluster.

At this point, there is a new set of fairly specific clusters that all share at least three common terms. These clusters will become leaves on the cluster hierarchy. Any singleton clusters that were not covered by rules are added into the new set and the entire process of determining a term set and cluster set is repeated. However, instead of generating rules with three terms, we only generate rules with two terms. That is, the value of d is decremented by one. This creates a set of clusters out of the more specific clusters that all share at least two common terms. The process is repeated one more time where d=1. The final result is the tree depicted in figure 1 where at the top level, all documents within a cluster share at least one common term, at the second level all documents within the cluster share at least two common terms, etc. Since the tree was built from the bottom-up, higher-level clusters are restricted to terms that are common to all subclusters and hence are likely to be relevant keywords and not spurious terms that

happen to index a large number of documents. Similarly, by building the tree from the bottom-up, higher level clusters are more likely to group relevant subclusters together than if the algorithm proceeded top-down.

The preprocessing phase of determining the term set requires O(*t*lg*t*) time to scan and sort the terms, where *t* is the total number of terms in all clusters. The clustering phase actually runs in O(*n*) time, where *n* is the number of documents. However, there is a large constant factor of (*termsize* choose *d*) that will dominate the runtime.

To reduce the runtime, we found good results starting with *d*=2 instead of *d*=3. This dramatically reduces the runtime but does result in a shallow tree. Nevertheless, we surmise this setting will be more popular due to the increase in speed while retaining good cluster performance, and in some cases, better generalization. Additionally, we examined clustering with *d=3* where only a random percentage of level 3 terms were examined. For example, with a percent of 0.25, only a random selection of 25% of the total possible rules was generated. A lower percentage decreased runtime at the risk of missing potential rules. However, if many good 3-word cluster combinations exist, the risk of missing potential rules is small.

Overlapping Clusters

A simple modification to the Tree Cluster algorithm allows the creation of overlapping clusters. In overlapping clusters, a document may exist in multiple clusters. This is not possible in the GQF and HAC algorithms. It is often desirable, since many topics do overlap.

To generate overlapping clusters with the TreeCluster approach, the process remains identical except when clusters are merged together. Rather than apply a rule only to the current set of clusters, the rule is also applied to all other documents, even if those documents have been removed from the current set after being covered by a previous rule. If these documents match a new rule, they are also added to its cluster. We hypothesize that this will be preferable to the users, and have enabled overlapping clusters by default in our algorithm.

Preliminary Experiments

A number of preliminary experiments were conducted to examine cluster speed, performance, and scalability. While these experiments give some indication of the performance, additional experiments are necessary for more data on how well the clustering algorithms perform under different circumstances.

Experimental Methodology

The evaluation of clustering algorithms is particularly difficult because no standard test data and test procedure exists. Moreover, the notion of what makes a good cluster is

subjective. Consequently, different clustering algorithms may not be easy to compare to each other.

In our experiments, we created two test domains: email and web documents. For each domain, three collections of 50 documents were created. For each collection, the author hand-selected 5 clusters with 10 documents per cluster. Cluster performance was measured against the expected clusters. The experimental results for each domain was micro-averaged across the three collections within each domain.

In the email domain, email documents were selected randomly from pre-designated folders. These folders included topics such as "Administrative", "Mail List", "Presentations", "Research", and "Personal". In the web domain, web pages were selected randomly from search queries. Searches were made on specific topics within computer science and artificial intelligence. For example, "genetic algorithms", "cache coherency", "formal design logic", or "ethernet token ring". These searches returned web pages guaranteed to contain a common set of keywords. Ten of these pages were randomly selected and saved as an expected cluster. The process was repeated for both the web and email documents until three collections per domain were created, each collection with 50 documents. Unlike Zamir et. al.'s work, the entire text of each document was used for clustering instead of a snippet. On average, the size of a document in the mail collection was 520 words while the average size of a web document was 640 words. In contrast, Zamir's snippets were 40 words in length.

Experimental Results - Speed of Clustering

Without regard to the quality of clusters, the first metric we examined was the time required generating the clusters. Results are shown in figure 2 for the three algorithms.

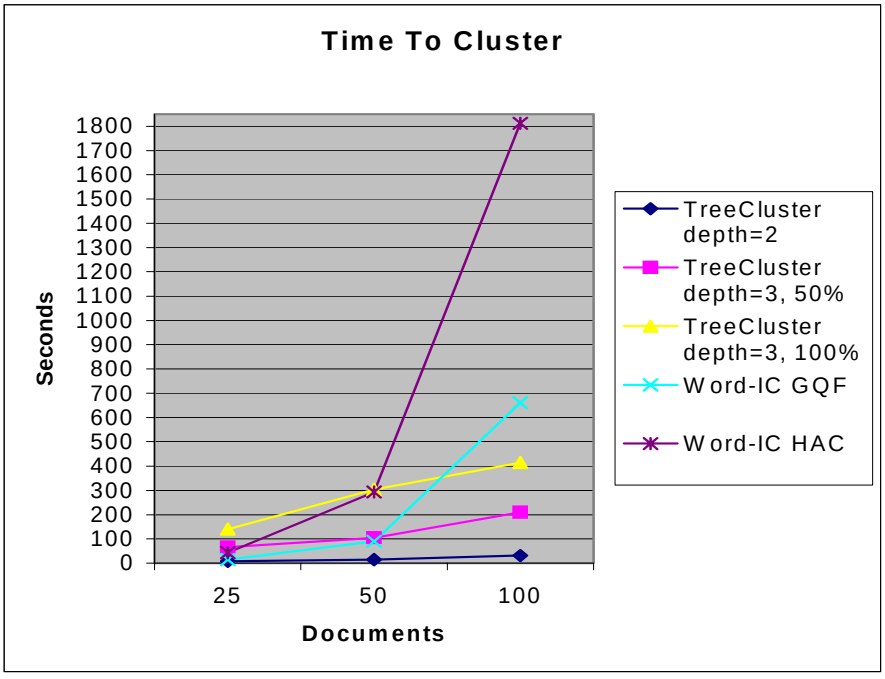

Figure 2. Time to cluster vs. # of documents.

The plot depicts runtime in seconds vs. the number of documents averaged across all runs. The runtime only includes the clustering and term reduction phases, not the time required for parsing, stop listing, and stemming. For the TreeCluster algorithm, three runs were conducted. The first run used a maximum tree depth of 2. The second used a maximum tree depth of 3, but generated only 50% of the possible rules, selected at random. For this setting, results may vary depending upon the random number selections. Consequently, three runs were conducted and the results averaged together. Finally, the third run used all possible rules that can be generated at a depth of 3.

As expected, the TreeCluster algorithm exhibits linear time growth and requires longer time as the tree depth and scope increases. The $O(n^2)$ Word-IC HAC algorithm grew the most quickly and required approximately 30 minutes to cluster 100 documents. The Word-IC GQF algorithm appeared fine for a small number of documents, but the $O(n^2)$ factor causes runtime to increase quickly past 100 documents. Of these algorithms, TreeCluster with a depth of 2 appears to be the only viable algorithm for fast clustering results if desired within a matter of seconds.

Experimental Results - Top Level Cluster Performance

The performance of each clustering algorithm was compared to the expected clusters with respect to the top-most clusters returned by the algorithm. All hierarchical structure within the clusters was ignored, and only the top-level clusters and all documents indexed within that cluster were denoted as belonging to that cluster. Note that this metric is subjective; even though we designated 10 articles to belong to a given cluster, there is overlap among documents and other clusters may certainly be legitimate. Nevertheless, it provides one metric to evaluate performance.

The performance metric used to score the clusters was to compute a score for each document and then average the scores together. The score of a source document was determined by looking at all pairs of documents for each cluster that the source document is in, and dividing the number of true-positive pairs by all possible unique pairs. A score of 1 indicates that documents were clustered with expected documents, while a score of 0 indicates otherwise.

However, this score does not reflect the number of clusters that were created. If a clustering algorithm put each document into its own cluster and then quit, it would receive a perfect score of 1 using this metric. Ideally, the total number of clusters should be combined with the cluster score (as in the GQF function) for a single metric. However, we have instead reported the total number of clusters created as a separate variable.

Results for the algorithms on the top-level clusters are shown in figure 3.

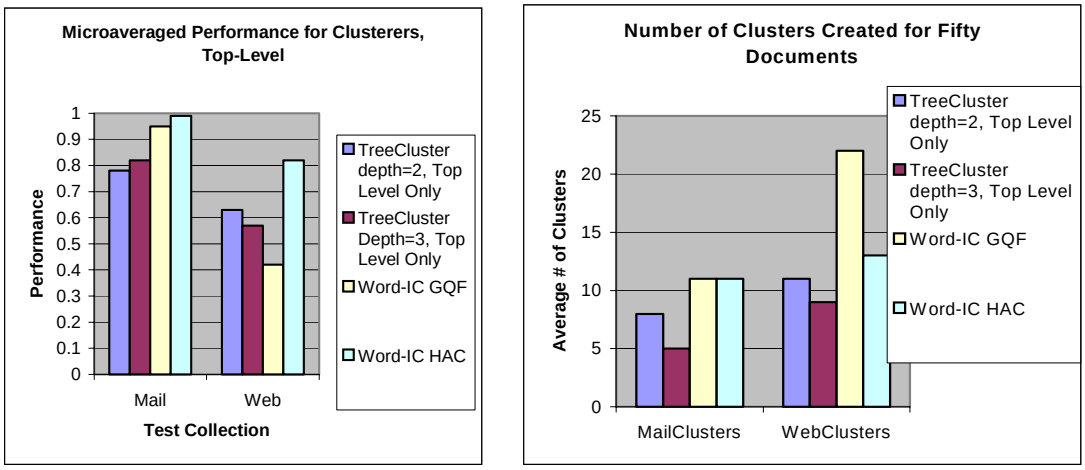

Figure 3. Top-Level Cluster Performance and # of Clusters Created.

Overall, the web domain appeared more difficult to cluster than the email. This may be a result of overlap among the computer-science topics that were selected. The Word-IC HAC algorithm performed the best on both the email and the web domains. It also resulted in a reasonable number of total clusters (10 and 12). The Word-IC GQF algorithm performed well on the mail domain, but extremely poorly on the web domain. The score was lowest (0.4) and the number of clusters high (23 clusters out of 50 documents). As previously noted, a slight modification to the selection of terms or the GQF function could result in different results.

The results for the TreeCluster algorithm are shown for both a depth of 2 and a depth of 3. While the depth 3 algorithm resulted in the fewest overall clusters, the performance score is almost identical. Consequently, the much faster depth 2 version may be preferred over the slower depth 3 version. Compared to Word-IC HAC and GQF, the TreeCluster was slightly worse on the mail domain but better than GQF on the web domain.

Experimental Results - TreeCluster Performance by Individual Cluster

A final experiment examined the clusters created by the TreeCluster method alone as the depth of the tree was increased from 2 to 3. At depth 3, performance and the number of clusters was measured for 33%, 66%, and 100% of all possible rules that can be generated at depth=3.

The score metric in this experiment was computed the same way as the previous experiment, except scores were tallied in individual clusters within the hierarchy, not agglomerated into a single large cluster at the top level.

The results are shown in figure 4. In counting the number of clusters created, the average cluster count is given by tree depth and domain. For example, in the Mail domain at TreeCluster depth=3, 100%, there were 5 clusters at the top level, 9 clusters at the second level, and 2 clusters at the third level.

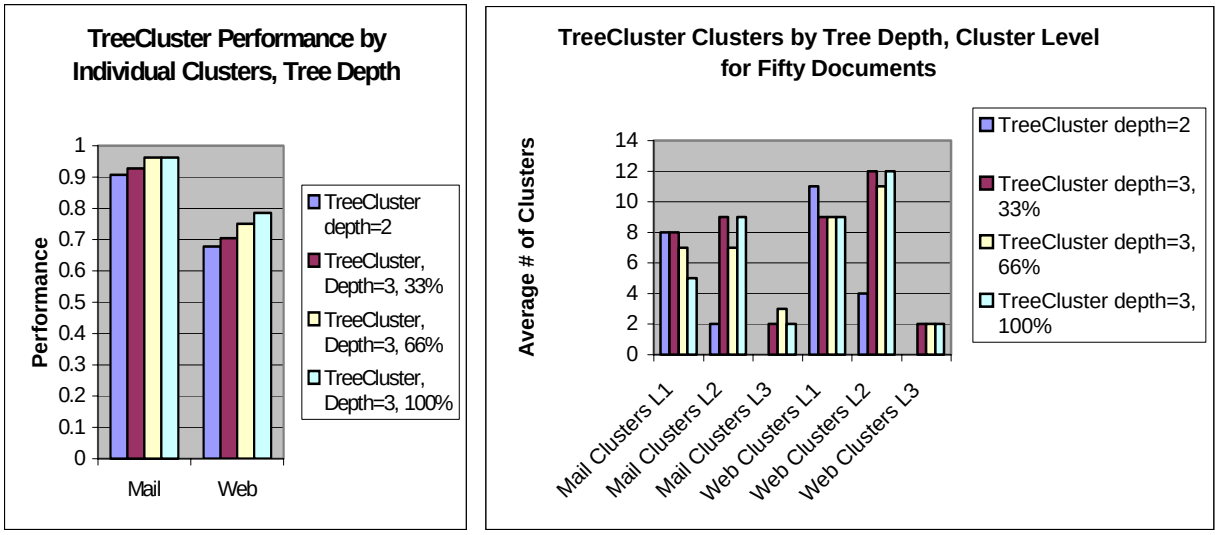

Figure 4. TreeCluster Performance by TreeDepth, Cluster Count.

The leftmost chart indicates that performance using the individual cluster metric increases as the tree depth and number of rules examined increases. This applies to both domains although the email domain appears easier than the web domain. As expected, generating clusters using a deeper tree results in better individual clusters. However, there are more cluster groups that are created compared to shallower trees. The TreeCluster at a depth of 2 averaged a total of 10 clusters across the entire Mail hierarchy, while the TreeCluster at a depth of 3, 100%, averaged a total of 16 clusters across the entire M ail hierarchy. However, at the topmost level, the depth=3 clusterer produced fewer clusters than the depth=2 clusterer, indicating that the tree is weighted towards the middle and not towards the top. This separation will help make the tree easier to browse since the total number of top-level categories is not high.

Interestingly enough, the TreeCluster algorithm did not produce very many clusters at depth 3, but most clusters at depth 2. This suggests that after the term selection process, most documents/clusters share only 2 terms in common.

Future Work

This work has only begun to examine many possibilities in terms of hierarchical clustering. More robust experiments that incorporate different domains need to be examined. Due to time constraints, we were not able to compare the TreeCluster method with the suffix tree clustering method proposed by Zamir, et. al. Additionally, there are many parameters that we could vary to examine the resulting performance. A different threshold setting, different ways to select terms, a larger or smaller term set, and other factors could dramatically alter performance. For example, a larger term set will likely result in more level-3 clusters with the TreeCluster algorithm. Usability studies will also shed light on what methods cater to user preferences. Finally, a better scoring mechanism and well-defined testbeds need to be created so that clustering algorithms may be compared to one another.

While much work remains to be done, this work has reiterated that Word-IC HAC gives excellent results, but results in an exceptionally slow runtime. The Word-IC GQF method appeared to give good results in one domain, but not in another. As a result, the algorithm may require tweaking to the GQF function in order to avoid extended clusters. Finally, the novel TreeCluster algorithm appears to give good results in both domains, scales linearly with the input, and generates a shallow tree hierarchy that may be easily browsed. However, is slow to generate clusters for deep trees and does not scale past a depth of 3.

Other interesting projects reserved for the future involve merging multiple techniques together. For example, a traditional HAC approach might be applied to generate initial clusters, and then applied to the TreeCluster algorithm in an attempt to harness the best qualities of both algorithms.

References

Rorvig, M. (1998). Images of Similarity: A visual exploration of optimal similarity metrics and scaling properties of TREC topic-document sets. Journal of the American Society for Information Science.

Voorhees, E. (1986). Implementing agglomerative hierarchic clustering algorithms for use in document retrieval. Information Processing and Management, 22 (6), 465-476.

Willett, P. (1988). Recent trends in hierarchic document retrieval: A critical review. Information Processing and Management, 24 , 577-597.

Zamir, O., & Etzioni, O., & Madani, O., & Karp, R. (1997). Fast and Intuitive Clustering of Web Documents. Proceedings of the 1997 American Association Conference for Artificial Intelligence.