Clustering Web Search Results Using Suffix Tree Methods

CS276A Final Project

Steve Branson
sbranson@stanford.edu

Ari Greenberg
arigreen@stanford.edu

ABSTRACT

Most search engines in use today present the user a single ordered list of documents matching the search query. Since most users do not browse past the first page of search results, this method of displaying results often limits the effectiveness of the search to the relevance of the first several documents. An alternative to a single ordered list is to cluster the search results and present a list of clusters to the user. The user selects which cluster appears most relevant, and the search results in that cluster are then displayed in a list. Under the assumption that documents which are similar to each other are likely to be relevant to the same query, we believe that clusters of search results are easier to browse than a single ordered list.

We have built a system that clusters search results from the web. We allow the user to issue a search query and we return the results as a set of coherent clusters. Our algorithm uses suffix trees to efficiently determine documents which share common phrases. The nodes in the suffix tree define the initial clusters. To increase the number of documents in each cluster, clusters which are sufficiently similar are merged. The algorithm itself is implemented in Java, and we use CGI and Python to display the results to the user.

1. RELATED WORK

There has been much work over the past several years in applying document clustering to web search results. Our implementation is based primarily on the work of Zamir and Etzioni.[4] In this paper, the authors describe the suffix tree algorithm and its advantages over previous approaches such as agglomerative hierarchical clustering. In our algorithm extensions, we also incorporated ideas about hiearchical clustering from Koller and Sahami, and Weiss, Veliz, and Sheldon.[1, 2] There are also several existing programs on the web which perform web search clustering. Grouper[3], developed by the authors of [4], is an implementation of Zamir and Etzioni's suffix tree algorithm. Other websites which cluster search results include Vivisimo and Northern Light. Our user interface was influenced by Vivisimo.

2. SUFFIX TREE ALGORITHM

2.1 Introduction

The first algorithm we implemented was a derivative of the Suffix Tree Clustering algorithm described in the paper *Web Document Clustering: A Feasibility Approach* by Zamir and Etzioni.[4] We found that this algorithm yielded better results than other clustering methods, given that our system was restricted to being built on top of an existing search engine such as Google. The suffix tree algorithm is particularly well-suited to clustering problems of this type where the number of words used to cluster each document is very small (i.e. just snippets returned by a search engine).

A **Suffix Tree** is a data structure that keeps track of all n-grams of any length in a set of word strings, while allowing strings to be inserted incrementally in time linear to the number of words in each string. The algorithm is theoretically fast, with a runtime of $O(n)$, where n is the total number of words in all combined document snippets. The algorithm has the important characteristic that the outputted clusters can have overlapping documents. This has the advantage that it ensures that a large number of substantial clusters can be generated, each of which can be labelled fairly accurately. From a user point of view, however, this feature has the disadvantage in that it increases the number of possible document listings that the user may need to look through. This is a drawback that is not easily factored into traditional evaluation techniques. The algorithm we have has several steps:

1. Preparing the Documents
   Retrieving the document snippets from Google and parsing and stemming the results.

2. Suffix Tree Construction
   Inserting the strings associated with each document onto the suffix tree.

3. Merging Clusters
   Combining similar nodes of the suffix tree.

4. Labelling Clusters
   Generating a label for each cluster.

5. Scoring Clusters
   Ranking clusters.

2.2 Preparing the Documents

In this step we retrieve each document snippet from Google and parse, filter, and stem the results. Results are received from Google in a multi-threaded fashion, and the actual construction of the suffix gree can occur concurrently as documents are retrieved over the network. We also have support to cache Google results to the local file system.

Building our clustering system on top of Google restricts the amount of information we have about each document to a snippet and a page title. We clean and add each of these to our suffix tree. Each snippet/title is first tokenized, cleaned of html tags, and converted to lowercase. Next we filter out the 30 most common words in English using a stoplist. We also filtered out various web-related words that were very frequent in webpages and were hurting results, such as 'http' or 'nbsp' using the same stoplist. Next we stem each word using the Porter stemmer.

2.3 Building the Tree

A **Suffix Tree** is a data structure that essentially has one node for every possible phrase in a collection of documents (except that for more efficient space use, consecutive nodes corresponding to words in the same document are combined). A reader should consult Zamir and Etzioni's paper for an exact description of what a suffix tree is. A suffix tree has the nice feature that any string can be inserted onto the tree in linear time by beginning at the root of the tree, and using logic to decide whether or not to add a new branch, or which branch of the tree to traverse or split into multiple branches. At the same time, at each node we maintain a list of documents that contain the phrase corresponding to that node, as well as an index that allows us (in combination with a document id) to lookup the phrase corresponding to that node. Rather than simply insert every possible phrase of any length in our snippets onto our tree, we consider only phrases of 5 or less words. We found that any 2 documents with a string of length more than 5 words in common were probably effectively the same document.

2.4 Merging Clusters

Once the suffix tree has been constructed, each node on the suffix tree that contains multiple documents can feasibly be considered a cluster; i.e. it is a string that is contained in multiple documents. To reduce the number of clusters, we next undergo a cluster merging phase. We begin by adding the N highest scoring nodes in our tree to a list. A score for a node can be determined as a function of the number of documents in the node and an estimate of the probability that the label corresponding to the node occurs on the web. We will talk more about this later. Using just the top N highest scoring nodes ensures a constant upper bound on the runtime of the algorithm. We found that N=500 was sufficient to ensure good performance.

To decide whether or not to merge any 2 given nodes, we use the same merging criteria defined in the Zamir and Etzioni paper, i.e. we merge clusters x and y if:

$$(N_x \cap N_y)/ \mid N_x > k \qquad (1)$$

and

$$(N_x \cap N_y)/ \mid N_y > k \qquad (2)$$

Here N_x is the set of unique documents in cluster x, and k is a constant betwen 0 and 1. Effectively, this criteria requires that each cluster must have at least the fraction k of its documents in common with the other cluster. We found k=.5 to be a good value.

Our algorithm moves iteratively through each cluster C in our list and considers merging it with each cluster D which comes later in the list than C. Nodes that have already been attached to some other node can not be merged again and are ignored. We maintain the data structures to do this fairly efficiently and to determine the documents and labels corresponding to each merged cluster updated.

2.5 Scoring Clusters

After merging suffix tree nodes, we have a list of clusters, each of which corresponds to one or more labels in the original suffix tree and contains a list of documents that contain any of those labels. By our merging criteria with k=.5, we are guaranteed that for each label l in cluster C, at least half of the documents in C contain l. The next step is to sort all resulting clusters by score. Score calculation is something that is not described in the Zamir and Etzioni paper but is nonetheless very important. In designing our scoring functions, we take several ideas into account:

- A cluster with more documents should have a higher score than an equivalent cluster with fewer documents
- The factor that a label l contributes to the score should be proportional to the epxpected probability of that label. For example, a common label such as 'the' is much less valuable than the a rare one such as 'cat Moses integral discrete'.
- Labels that are subsets of one another are likely to be in the same merged cluster (because they likely contain the same documents) so we don't want to double sount them.

Our final score S_C for cluster C is

$$S_C = N_C * \sum_{l_i=l_0}^{l_n} p(l_i) \qquad (3)$$

Here N_C is the number of documents in cluster C. We consider only labels l_0 to l_n that are in C and are not subsets of any other label in C. For each label l, we approximate its probability p(l) as:

$$p(l) = \sum_{w=l_0}^{l_i} p(w) \qquad (4)$$

Here we are saying the probability of seeing string l is simply the product of the probabilities of each word w in l, i.e. we are using an approximation that assumes independence of words in a string. For the probability of each word w, we use
$p(w) = log(1/f_w)$ if $f_w > 0$ and
$p(w) = log(1/.5)$ if $f_w == 0$
where f_w is the frequency of w on the entire web. We constructed a hash table for our expected values of f_w by writing a web crawler that begins at various pages we believed to be

representative of the web, and recording word frequencies while parsing and stemming each document the same way that we do our document cleaning for our suffix tree. We let our web crawler run for a few hours and saved the results to a file. Our reasoning for p(w) is that the probability of a word is inversely proportional to its frequency on the web. However, we use the log of the inverse frequency instead to smooth out bias in our web crawler and in the terms found in pages for some particular query.

We also do further tweaking beyond what is shown in these equations. In particular, rather than simply making S_C proportional N_C, we weigh each unique reference of a document in C by some amount that varies depending on whether or not the node was referenced in the title of a document or the snippet or both. It would also be valuable to weigh a document reference by the rank of the document returned by Google; however, we are currently not doing this.

Once we have sorted each cluster by its score, we present the clusters to the user ranked by highest score. Note that this is not necessarily a good idea. It would be better to attempt to rank the clusters by some measure that also attempts to present clusters that do not overlap much with previous clusters. This is important because clusters using this algorithm have overlapping documents, and therefore it is common for one highly ranked document to contain similar documents to another highly ranked document.

2.6 Labeling Clusters

The suffix tree algorithm has the advantage that clusters are derived from nodes in the suffix tree, which correspond to phrases that are contained in each document in that node. This no longer becomes true after nodes are merged; however, it always remains true that at least half of the nodes (for k=.5) will contain any given label in the merged cluster. Therefore, any label or combination of labels in the merged cluster should be a good general description of the documents in the cluster. However, because it was common for labels that are subsets of one another to be merged into the same cluster, presenting all labels to the user is unnecessary. We simply label each merged cluster by all labels in that cluster that are not subsets of any other label.

3. ALGORITHM EXTENSIONS

3.1 Introduction

Although the suffix tree algorithm works fairly well in practice, we were skeptical of its theoretical value for several reasons. The first reason is that it does not scale well as the number of words per document (snippet) is varied. This is so because the algorithm considers any document that contains a phrase P to be a member of the cluster coresponding to P. As the number of words in document D increase, the probability that D contains phrase P also increases. As a result, the average number of documents for a given cluster increases with document size even though the number of documents remains constant. In the theoretical limit, as the number of words per document goes to ∞, a given cluster will conatin every document.

Another problem with the suffix tree algorithm is that it is difficult to combine it with other sources of information to improve clustering performance. Since clustering is determined by whether or not a document contains a certain phrases, it is difficult to conceive of a way to alter clustering results using other information.

We wanted to get a feel for how important the number of words per document is to clustering performance, and how effect use of document link information is to improving clustering. As a result, we decided to implement a hierarchical clustering algorithm based on the single link method. This algorithm uses document-document similarity scores, and essentially iteratively joins the two most similar pairs of objects together until all objects have been joined. Since our suffix tree data structure allows us to track which n-grams documents have in common, it is still applicable to these calculations.

3.2 Similarity Based Clustering

The algorithm works by allocating an n x n upper-left triangular document-document similarity matrix M where n is the number of documents we are clustering. We construct our suffix tree in the same manner as we did before, except that we no longer bother to merge tree nodes. Next we iterate through each node N in our tree. For each pair of documents (D_i, D_j) in N, we increment M_{ij} by S_N, where S_N is the score of node N as computed in the previous section.

Once the matrix M has been constructed, we create a priority queue which will contain entries of M_{ij} in sorted order. For each document D_i, we add the 10 highest ranking entries of M_{ij} to our queue. The data structures are designed such that for any document D_i, we can locate all corresponding entries in our queue with that document in constant time.

The algorithm incrementally joins joins the highest ranking M_{ij} on the queue. The corresponding clusters C_i and C_j will be merged into new cluster C'_j. This is done by collapsing every entry M_{jk} and M_{ik} into a single entry M'_{jk}, and adjusting each similarity scores by the equations:

$$M'_{jk} = f * M_{jk} + (1 - f) * M_{ik} \tag{5}$$

$$M'_{il} = f * M_{il} + (1 - f) * M_{jl} \tag{6}$$

Here our new similarity score M'_{jk} between merged cluster C'_j and any other cluster C_k is a weighted average between M_{jk} and M_{ij}. We use f=.5. The algorithm iterates until all clusters have been merged into a single cluster. The result is a binary tree that iteratively divides all documents into smaller clusters as you move away from the root (see **figure 1**). Note also that the confidence of any branch of the tree can be measured by looking at the similarity score M_{ij} used to join those two clusters together.

The hierarchical structure and confidence measures allows for greater control in the manner in which the final clusters are chosen, and also allows the possibility for a user interface allowing the user to explore clusters and sub-clusters. The easiest way to extract distinct clusters from the tree hierarchy is to traverse down the tree starting from the root until the confidence measures of a branch reaches above a certain threshold, and take each such branch and its children as a distinct cluster.

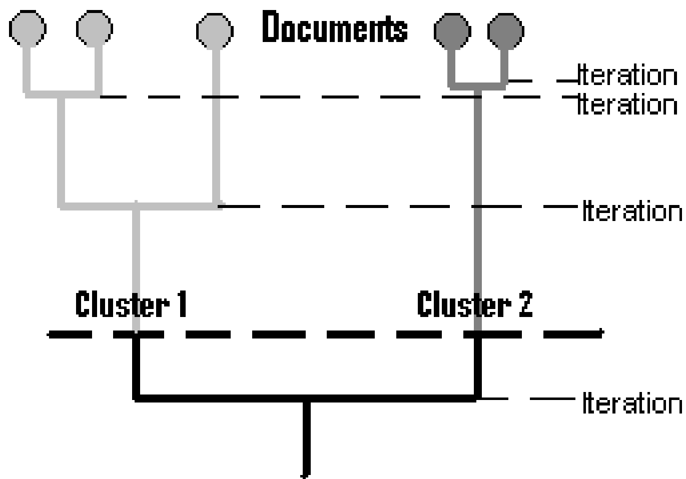


Figure 1: Example of the similarity-based algorithm where circles represent documents and their similarity is represented by the inverse distance between them in the picture. Each branch of the tree represents an iteration of the algorithm where 2 clusters are joined. One way to choose the returned clsuters could be the light-gray nodes as one cluster and dark gray nodes as another.

3.3 Incorporation of Long Documents

Unlike the suffix tree algorithm, the similarity based algorithm is not affected by the number of words per document. Therefore, as a test, we added the ability to download each webpage from a search result from Google, cache it, and insert it onto our suffix tree. Phrases extracted from within the document page can be weighted differently from phrases from the document snippet or document title.

3.4 Incorporation of Link Information

We also experimented with including an extra term in our document-document similarity matrix based on document link structure. In this case we extended our web crawler to search all Google web results for a given query, each down to a certain depth. It builds a link-connectivity graph and saves the graph to a file. The link-connectivity graph encodes information that can be used to extract all incoming and outgoing links from each page down to a certain depth.

Once the connectivity graph has been constructed, we can update each similarity score M_{ij} between document D_i and D_j. We do this by iterating through all documents D_i. We begin with a maximum link-structure score boost S_L set to some constant amount. A single link in document D_i to document D_j would yield the largest possible increase in M_{ij}. Otherwise, we divide S_L by the number of links in D_i, and for each link going into any D_j, increment M_{ij} by S_L. In addition, even if D_i contains links to other documents not in the set of results returned by Google, it is still possible to recursively traverse the graph on those documents up to a certain search depth. It makes some sense to continuously reduce S_L in this fashion by the number of outgoing links, otherwise the total expected contribution of D_i to scores with other documents would grow exponentially. However, a small exponential growth might actually be desired and dividing by the number of outgoing links might be a little severe.

4. USER INTERFACE

Traditionally, the most effective search engines have sported extremely simple user interfaces. Our interface is no different. The user types his or her query in a textbox, and a list of all clusters matching the query is displayed. Clicking a cluster brings up a list of search results in that cluster. The initial search shows all of the search results of the first cluster.

The success of our user interface greatly depends on the precision of the cluster labels. If the labels give a good indication of what documents are actually in that cluster, our user interface becomes very effective because it allows the user to easily choose to browse only those search results that are likely to interest him or her. In this case, we believe that our interface is more effective than a single ordered list because with only one click (choosing the cluster) the user can view many of the search results which are relevant to his or her information need.

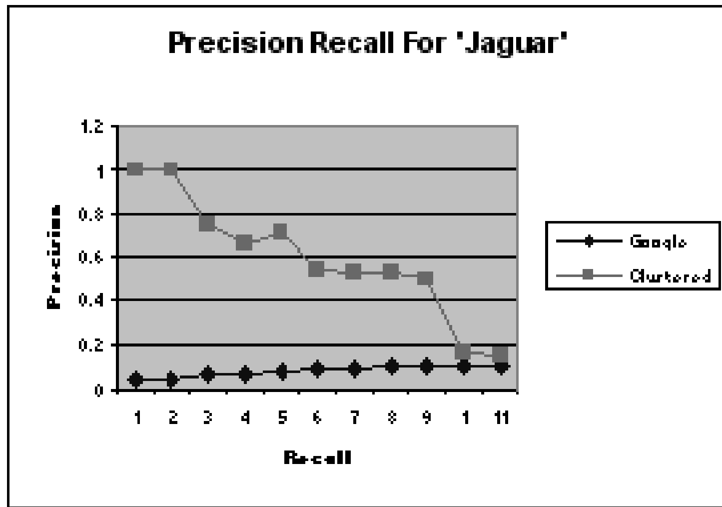
5. EVALUATION OF RESULTS

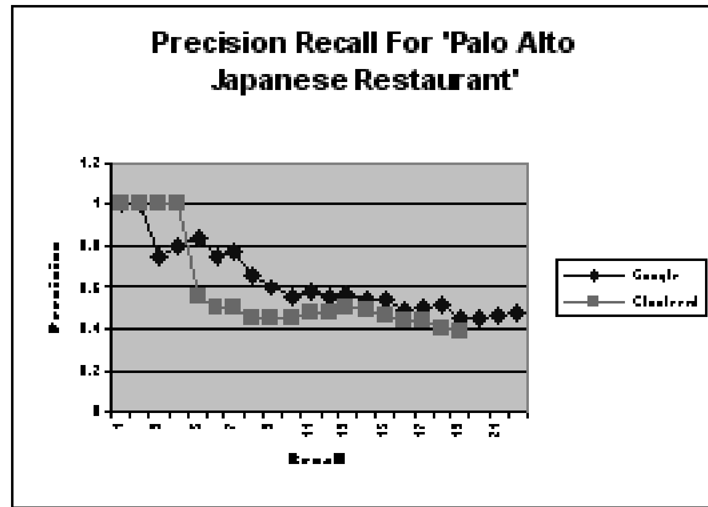
Evaluating the precision and recall of a document clustering system is not a well defined task because there is no single ordering of the results. There are many possible approaches to this evaluation. If the clusters were perfectly labeled so that the user always chose the most relevant cluster, then it would be sufficient to evaluate the precision and recall of the highest scoring clusters in order. Since cluster labelling is one of our more challenging tasks, however, we felt that evaluating our results in this way would give us an unfair advantage. Instead, we ordered the clusters according to which labels seemed most relevant to the information need. For this cluster ordering step, we looked only at the cluster labels, ignoring the contents of each cluster. We then defined the ordering of the results as the ordering of the documents in the highest rank cluster, followed by the documents in the next ranked cluster, and so on through the lowest ranked cluster. Documents appearing in multiple clusters were removed from all but the highest ranked cluster in which they appeared, so as to avoid duplicates.

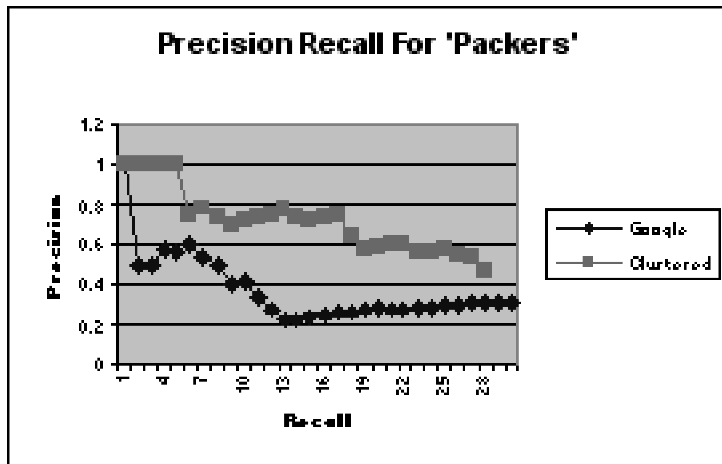
To evaluate the performance of our program, we constructed several fictitious information needs and converted each to a search query of one to four words. Each query was issued to Google, and we marked each of the top 100 search results as relevant or nonrelevant. From this data, we created a precision-recall graph for Google, which we used as a baseline. We next ordered the clusters by label as described above, thus creating an ordering of the documents by our program. With this ordering we were able to create precision-recall curves for our system.

5.1 Suffix Tree Clusterer Results

The performance of the suffix tree clustering algorithm varied greatly depending on the type of search query (see **figure 2**). As expected, it performs very well on searches in which the search query produces groups of pages with distinct, non-overlapping content. It tends to perform best when the







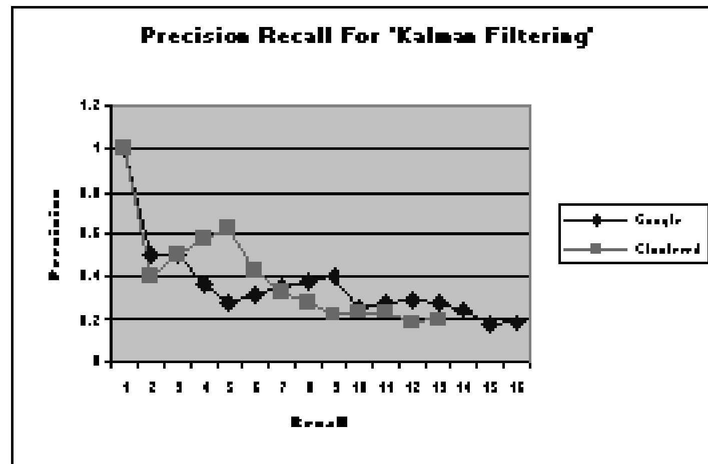


Figure 2: Precision-Recall Curves for 4 different search queries

search query is fairly general and not specified extremely well. When the search query contains many terms, the diversity of page content of the results returned by Google is very low, and therefore our clustering algorithm has little information it can use to segregate relevant and non-relevant documents. As a result, this type of system seems to heavily favor users who are simply exploring the web without knowing exactly what they are looking for before they formulate their search query.

The query 'jaguar' is an obvious example of a query that our algorithm will perform well on. In this case, the user was searching for pages related to the animal jaguar, rather than the car or the operating system. Since pages about the car dominate the results from Google, Google performs extremely poorly, while our cluster search performs very well.

A slightly less contrived example is the query 'Packers', in which the user was searching for pages related to recent news about the NFL football team. In this case, the search query was left a little too vague in that the more adept searcher would know to at least search for 'Green Bay Packers'. Despite this, pages about the NFL team still made up the vast majority of the pages returned by Google, and the ability of the cluster search algorithm to segregrate different types of pages about the 'Green Bay Packers' was responsible for most of its success over Google. In particular, it was able to do a good job filtering out pages related to Packers merchandise and fan sites, while identifying clusters related to Packers news and recent events.

'Kalman filtering' was a much more difficult query. In this case, the user was not searching for any particular genre of Kalman filtering, but instead was searching for general pages which seemed to have reliable, trustworthy information on the subject. The major problem faced here was that many of the pages that seemed to have attractive sounding, scholarly titles and snippets turned out to be links to books for sale. As a result, it was hard even for a human to determine which documents would be relevant just from the snippets. On this query, the clustering algorithm performs about the same or slightly worse than Google. This is not too disappointing because our algorithm is not at all factoring in Google page rank when ordering clusters, while Google is using some kind of ranking procedure that favors more reliable or cited sources.

'Palo Alto Japanese Restaurant' is an example of a query that was specific enough that our clusterer did not perform particularly well. Again this is a case in which Google's use of source reliability to rank documents gives it an advantage over our approach. Adding in a term which partially weights our clusters by document rank would help improve performance. It seems clear, though, that our clusterer is not particularly helpful for well-defined search queries and would be more useful to novice search engine users. Even so, our clusterer seems to extract many good key phrases even for difficult search queries, and we believe that if our system was fine-tuned and ironed out it could be useful even to more advanced searches.

5.2 Use of Documents Vs. Snippets

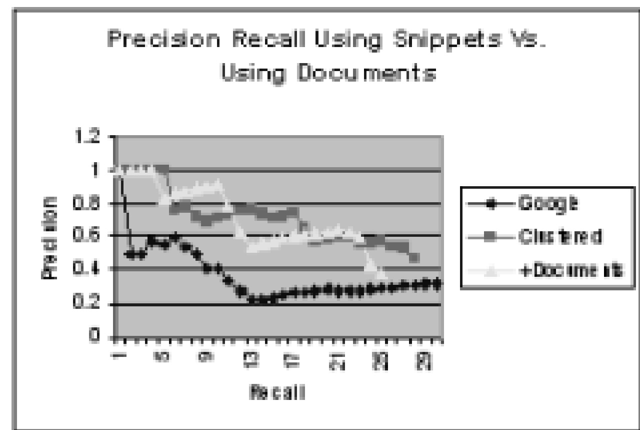


Figure 3: Precision-Recall Curves comparing using just snippets to using full documents with the Suffix Tree Algorithm

In general we found that use of words inside of document pages in addition to snippets slightly reduced the algorithm's performace, or at least made it more erratic (see **figure 3**). The best cluster labels both methods found were more less the same; however, the actual clusters tended contain more documents than the clusters in which only snippets were used. This comes as a result of the suffix tree algorithm, which considers a document to be a member of a cluster if the document mentions any label in that cluster one or more times. The result was that for many of the labels applicable to relevant documents, the corresponding clusters would correctly contain many of the documents pertaining to the relevant labels, but the clusters would also incorrectly contain several pages which happened to mention the labels just a single time.

5.3 Similarity Based Algorithm Analysis

In general we found that the straight Suffix Tree approach performed slightly better than the similarity approach (see **figure 4**). One reason was that we did a little better job tweaking the Suffix Tree algorithm approach. Another major reason was that the straight Suffix Tree approach did a better job labeling clusters. Using a similarity based method, there is no guarantee that all or even most documents in a cluster will contain a given label. Therefore choosing a label is hard and would require more sophistacted approaches than we had time to implement. The last reason is that our evaluation technique is slightly biased toward the Suffix Tree approach. This is so because different clusters can have overlapping sets of documents. When the user orders the clusters, a document list is generated. If a document is repeated in that list, we simply pretend we didn't see it the second time when constructing our precision-recall curve. A better system might incur some penalty in this case, since it leaves more documents for the user to sift through.

Regardless, it seems if the goal is to simply cluster documents in realtime based on short snippets, the Suffix Tree algorithm is hard to beat. For other types of problems, where more time or information is available, we believe that the similarity approach method at least has more potential

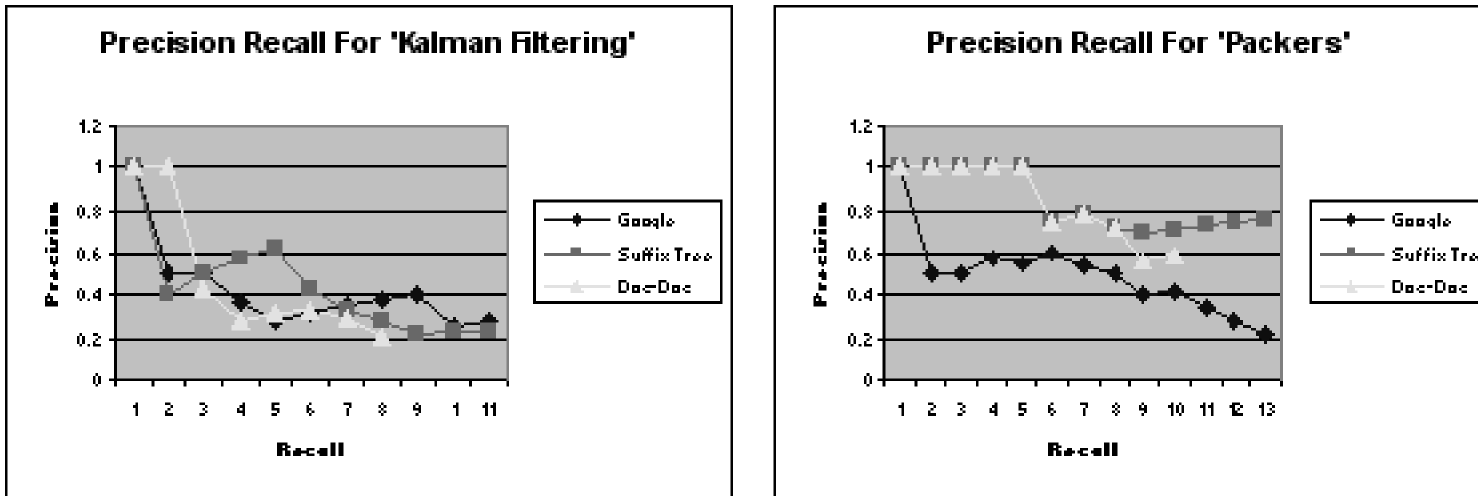


Figure 4: Precision-Recall Curves comparing the straight Suffix Tree algorithm to a Similarity Based Hierarchical Clusterer using the same Suffix Tree data

to do a better clustering job.

5.4 Link and Document Analysis

Comparison of the effects of clustering using documents with more words, as well as the effects of using link information, is more reliably tested using the Similarty Based algorithm than the Suffix Tree algorithm. Here we see that use of documents provides a marginal improvement over using just snippets(see **figure 5**). This seems to imply that it is not terribly necessary to use a high dimensional space to cluster documents, and that snippets are usually nearly sufficient. We believe this partially results from the use of the Suffix Tree data structure, which allows us to track n-grams of any length. Long phrases in common between multiple documents can be extremely informative in evaluating document similarity. It seems clear that if we were using a vector space model considering only frequencies of single words rather than n-grams, varying document size would have a much greater effect.

Use of document-link structure actually hurts performance in our tests. We think this is happening due to shortcomings in what we were able to implement. We did not have time to run our web crawler for a very long time, and as a result, we were only able to construct a link graph with a very limited search depth (3 levels). The result was that link information tended to provide little or no information and we were not able to tweak our scoring functions to make it work. We still believe that link inforation can be helpful, but we believe page content is a more reliable source of information than link structure.

6. IDEAS FOR FURTHER WORK

One way in which our program could be improved is to make the label descriptions more clear. Our current system extracts the labels directly from the suffix tree, which means that words in the stoplist are not displayed. As a result, the label description sometimes appear jarring to the user. If we had had more time, we could have added pointers to the snippet data structures to help remember the exact phrase as found in the snippet.
Another idea which we would be interested in exploring is applying the original suffix tree algorithm to complete web pages instead of snippets. While this might not be practical in terms of speed, it would be interesting to see how this affects the quality of the clusters. Zamir and Etzioni mention in [4] that the clusters based on snippets are almost as good as clusters based on the full text, but it would be interesting to compare our results with theirs.

7. CONCLUSIONS

All things considered, we are impressed with the performance of our system. In many cases, we have found relevant documents inside appropriately labeled clusters which we likely would not have spotted using Google since they were towards the lower end of the list. While our program does not seem to be an improvement over Google for very specific queries, we believe it performs quite well for general queries where the clusters are likely to be significantly different from one another.

8. REFERENCES

[1] D. Koller and M. Sahami. Hierarchically classifying documents using very few words.

[2] M. A. S. Ron Weiss, Bienvenido Velez. Hypursuit: A hierarchical network search engine that exploits content-link hypertext clustering.

[3] O. Zamir and O. Etzioni. Grouper: A dynamic clustering interface to web search results.

[4] O. Zamir and O. Etzioni. Web dcument clustering: A feasibility demonstration.

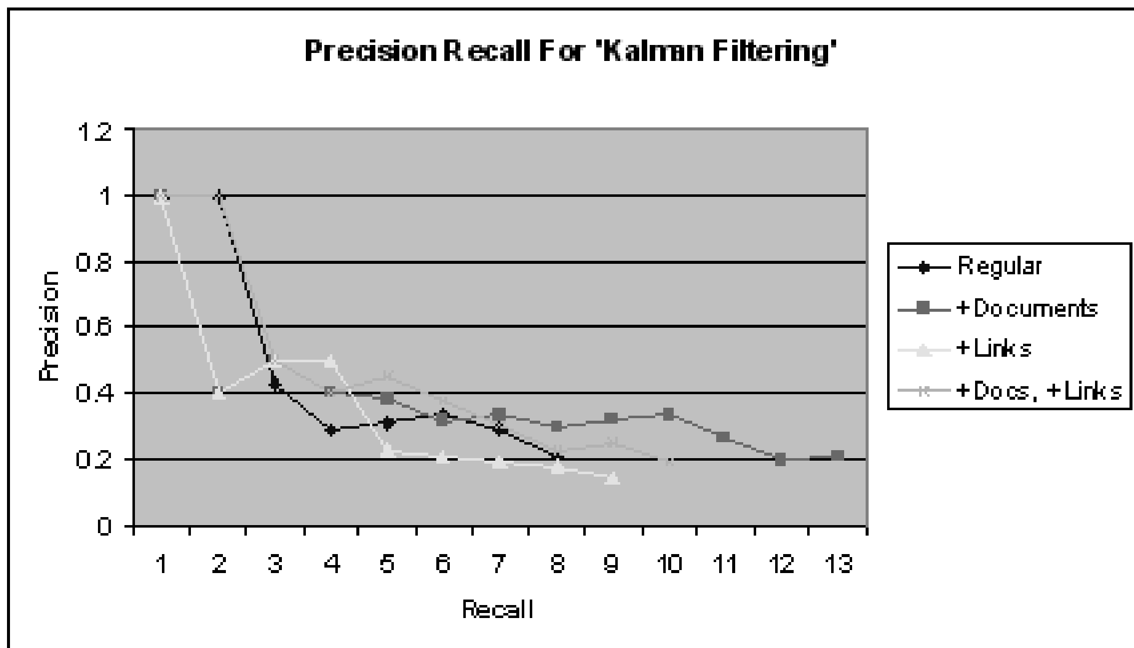


Figure 5: Similarity Based Algorithm Analysis: The curve 'Regular' is the algorithm using just snippets. '+Documents' also incorporates words from the actual document. '+Links' uses snippets and link information. '+Docs, +Links' uses snippets, documents, and links