INTELLIGENT ACCESS TO PUBLIC EMAIL CONVERSATIONS

BY

WILLIAM LEE

B.A., University of California at Berkeley, 2000

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2005

Urbana, Illinois

Abstract

Traditionally, access to newsgroups and mailing lists is limited to two methods: browsing and searching. Browsing effectiveness declines when the number of messages scales up. Searching, on the other hand, does not work for new subscribers, since they can not generate queries that are accurate enough to find the prominent discussions. This thesis has two main contributions that address those shortcomings. First, we propose a novel way to cluster public conversations using agglomerative clustering and a machine-learnable similarity function. Our experiment has shown that this composite similarity function can outperform any single similarity function in our test corpus. Secondly, we introduce a new visualization method called Conversation Map (CM). Extending from the successful Treemap paradigm from HCI (Human-Computer Interaction) research, CM can display conversation clusters and their temporal relationships effectively.

Lastly, this thesis introduces CEES, or Conversation Extraction and Evaluation Service, a reusable and extensible architecture that integrates message management, indexing, querying, clustering, and CM together. CEES effectively combines both browsing and searching and eases the entrance for new subscribers with a logical overview of the forum.

To my Mom and Dad, who exemplify how to love genuinely and live generously.

Acknowledgments

Ideas can be wild, but my adviser ChengXiang Zhai helped me focus on what is more important and necessary. In the last two years, he guided me through this maze of crazy ideas with a high level of professionalism. This work would not be possible without his help.

Hui Fang and Yifan Li have both participated in various parts of this research. I want to thank them for their constant injection of ideas and their significant contribution to this project.

I want to thank Hui Fang especially for her constructive comments on the early drafts of this thesis.

Table of Contents

Chapter

List of Figures

List of Tables

Chapter 1

Introduction and Motivation

> "Come to think of it, there are already a million monkeys on a million typewriters, and the Usenet is NOTHING like Shakespeare!" – Blair Houghton

Even with the growth of the WWW and web-based forums, Usenet and mailing lists still represent a large portion of the internet traffic. According to [24], some experts estimate data in newsgroups doubles every 8 to 12 months. There are more than 90,000 newsgroups in use as of 2002.

Newsgroups and mailing lists can serve as rich knowledge bases since they allow for extensive question and answer exchange among newbies and experts. For example, Lucene, an open-source Java information retrieval library, maintains a very active mailing list. From September 2003 to September 2004, Doug Cutting, the original author of the software package, has posted on the the list 271 times. During the same period, Otis Gospodenetic and Erik Hatcher, the two authors of the only Lucene book[14] and head developers of the project, have posted 406 and 662 times respectively. Since the Lucene mailing list archive contains only around 6200 posts during that period, the three experts have generated around 21.6% of all the mailing list traffic! This means if you are a subscriber to this mailing list, that is a very good chance that you can get your answers straight from the core developers of the project. One can only imagine getting this kind of support elsewhere.

The growing internet traffic, however, makes those forums more difficult to access using traditional browsing methods. As an example, the Apache Tomcat, a popular Java web application server, has on average 13MB of mailing list traffic per month from March 2000 to March 2004. There are 37587 messages exchanged in that mailing list alone in 2003. This means that a subscriber on average receives more than 100 messages a day! For a new subscriber, it is difficult enough to screen through messages received for the day, not to mention going through the complete archive.

Even with search capability, it is difficult for new users to find what they want, since they often lack the necessary background to formulate the relevant queries. This makes picking out important information from the group a challenging task with existing methods.

As a result, duplicated questions are being asked constantly. FAQs, or Frequently-Asked Questions, are created to address this issue. However, FAQs are difficult to maintain, since they requires constant updating from a small group of expert users. Those users do not usually like to continuously monitor the group and mine for repeating questions. They would rather spend their time improving the product or service.

What new subscribers need is a summarizing view of the forum that can show the general trends of discussions. This accomplishes the following:

1. New subscribers can get an overview of the reappearing topics that are being discussed in the group. This can help drive down the number of duplicated questions.

2. New subscribers can learn common terms used in the group in order to better formulate their search queries.

3. FAQ maintainers can use this summarizing view to extract frequently-discussed topics.

In order to create such overview, however, we need two main components:

1. A reliable way to cluster messages into relevant groups so they can be viewed as a whole.

2. A method to visualize those clusters effectively. Since those clusters can sometimes be large, we need a way to "zoom" to the most relevant conversations.

The first part of this thesis discusses methods to address the first challenge. We first propose a clustering algorithm that takes advantage of the threaded nature of the conversation. We then present results from our experiments that show the effectiveness of this clustering method.

The second part of this thesis describes a new visualization method named Conversation Map, or CM, that can generate a logical overview from the conversation clusters. CM is based on Treemap[30], which can effectively show hierarchical structure in a 2-D space. On top of Treemap, CM adds two temporal dimensions to the visual display. Those dimensions allow user to look at the conversation clusters in a sequential order. In addition, CM can dynamically modify the intra-cluster similarity threshold based on user needs. This allows users to "zoom" to the more similar discussions.

The last part of this study introduces CEES, or Conversation Extraction and Evaluation Service. CEES serves as the general framework to integrate the clustering and visualization components.

Chapter 2

Related Works

Much of the recent work on email management concentrates mainly on classifying and extracting data from emails. In the simplest sense, they study how to design systems to automatically sort emails into different categories. Various classification techniques such as Naive Bayesian and rule-learning algorithms have been tried[20, 11, 7]. For example, the initial work on the Enron Corpus[20] gives researchers a comprehensive email data set from a real-world company. However, related works on Enron Corpus try to solve the personalization email management problem, which is different from what we try to do in this work. Nevertheless, techniques such as combining similarity scores from different parts of the message can be used in our domain as well.

Other works on email management try to build intelligent agents that can extract interesting information from daily conversations. There are various attempts to create a comprehensive email management system [5, 26]. Other works put emphasis on extracting and evaluating information from emails. For example, using effective machine learning methods such as SVM, AdaBoost, and VotedPerceptron, Carvalho et. al. [6] have shown that segmenting email messages into different logical parts can be done with fairly high accuracy. Cohen and others [8] have also discovered that one can classify parts of a conversation into different intentions with existing machine learning algorithms.

From the user interface domain, researchers have found alternative ways to visualize newsgroup conversations [12, 32]. In particular, the "piano roll" method described in [32] shows

a ranked list of authors who contribute the most number of posts in the thread. Smith and Fiore believe that this component can help users identify the most relevant posts within the conversation context.

Fundamentally, most information retrieval methods would apply to the mailing list context without much modification. In particular, since email messages in general have more defined structure, we can potentially leverage the work done on structured information retrieval [19, 21, 25, 33] and various smoothing strategies [37, 16].

Furthermore, papers for customer review summarization such as [18, 10] suggest various methods for extracting common features for comparison purposes. Although they may not directly apply to this work, similar ideas of feature extraction are useful for identifying important common topics in the newsgroup.

This thesis does not try to address the more general task of question answering. It does, however, propose a new method to cluster and view those interesting common discussions. Instead of looking at the email management problem as a classification problem, we approach the challenge from the clustering point of view. Unlike classification problems, which require training sets for each of the class, we attempt to learn a similarity function that can work across different newsgroups or mailing lists.

Chapter 3

Email Thread Clustering

> "I can't work without a model. I won't say I turn my back on nature ruthlessly in order to turn a study into a picture, arranging the colors, enlarging and simplifying; but in the matter of form I am too afraid of departing from the possible and the true." – Vincent van Gogh

3.1 Motivation

In order to generate interesting conversation clusters, we first need to estimate what users use the clusters for. In this study, we assume that the users use those clusters as possible FAQ candidates. Base on this assumption, we can use the FAQ's properties as our clustering goals.

First of all, FAQs intend to eliminate duplicated questions from a newsgroup or mailing list. This means a FAQ represents a common issue in a product or service that are being asked multiple times before, possibly in different ways.

Secondly, since a question may not be relevant as time goes by, domain experts may need constant attention to update the FAQ. This prevents the FAQ from becoming outdated and growing larger than what the users are willing to read. In other words, we need to be sensitive to the "age" of a FAQ.

In the following sections, we try to identify several key properties for a FAQ. After specifying those properties, we can create a general model for our clustering algorithm.

3.1.1 Similarity

Similarity is probably the most obvious property of all. Two questions will be assigned to the same FAQ only if they are similar in nature. Although the questions may not be asking or discussing the exact same matter, the intention of the questions or answers should be used to determine this property.

3.1.2 Popularity

Popularity measures the FAQ's usefulness to the general public. The definition of *popularity* may be vague. A question can be consider popular when it has been asked 200 times before at various time points. Or rather, the question may be considered popular if it was asked 10 times yesterday. It is worth mentioning that this property may be in conflict with the *similarity* measure. If you group less similar messages together, then one can say that the topic is more popular since the cluster contains more messages. Thus, one can only apply the *popularity* property on groups of messages that share the same level of *similarity.*

3.1.3 Freshness and Persistence

One common challenge for finding useful information within a mailing list is the *freshness* and the *persistence* factor. This is a common pitfall for traditional IR systems – assuming that each document is static and does not change through time.

For example, for any given question, there may be different answers for such question. One example is:

> "What's the latest stable release for the Tomcat web server?"

The answer to that question in year 2000 is probably different than the answer one gets from year 2005.

The *freshness* of a given message does not necessary correlate to the quality of the FAQ. We also want to look at how questions are asked throughout the period of time. The goal is to show the user both topics that are prevalent through a long period of time (*persistence*) and hot topics that are being discussed recently (*freshness*).

3.1.4 Coverage

The *coverage* measures how well the FAQ covers the portion of the questions that are asked in the mailing list. It is not necessary for a FAQ to cover all the questions, but only the most popular ones that best represent the discussion in the mailing list. In some cases, this may contradict with *freshness,* since you may not want to have antiquated questions to end up in your FAQ.

3.2 Representation of Threads

With those FAQ properties in mind, we can now describe what we attempt to do for the clustering task. We first need a formal model to represent threads in a newsgroup.

We define a newsgroup G as a set of message threads $T_1, T_2, ..., T_n$. Each thread T_i represents a set of messages $m_{i1}, m_{i2}, ..., m_{i|T_i|}$. We assume that one message can only belong to one thread. In other words, $G = \bigcup_i T_i$ and $T_i \cap T_j = \emptyset$ for any i and j. It should be noted that the messages m_{ij} are grouped together in a reply tree within T_i. One can see Appendix A for a description on this message threading algorithm.

We can simplify the retrieval model within a thread by treating each message within T_i as one single message. Thus, in the traditional bag-of-word model, we let T_i and m_{ij} be bag of words. We then set $T_i = m_{i1} \cup ... \cup m_{iq}$.

3.3 Measuring Thread Similarity

3.3.1 Basic Similarity Functions

The similarity between two discussions is measured by $sim_x(T_i, T_j)$, where x is considered a particular "perspective" when comparing the threads. Larger $sim_x(T_i, T_j)$ indicates higher similarity. There can be multiple perspectives when comparing two threads, therefore, we can have multiple similarity functions. Figure 3.1 illustrates the different perspectives that we use in our clustering algorithm. Two threads, Thread 1 and Thread 2, are shown in the figure. The Subject, Authors, and Date are called "thread properties". Thread properties are associated

with the thread, not with a particular message. For example, the Subject property of the thread is set to be the "Subject:" field of the first message, since subsequently replies usually reuse much of the parent's subject line. The Authors property contains a list of email addresses that are extracted from the "From:" headers for all the messages in the thread. The Date property is just the date for the first message of the thread.

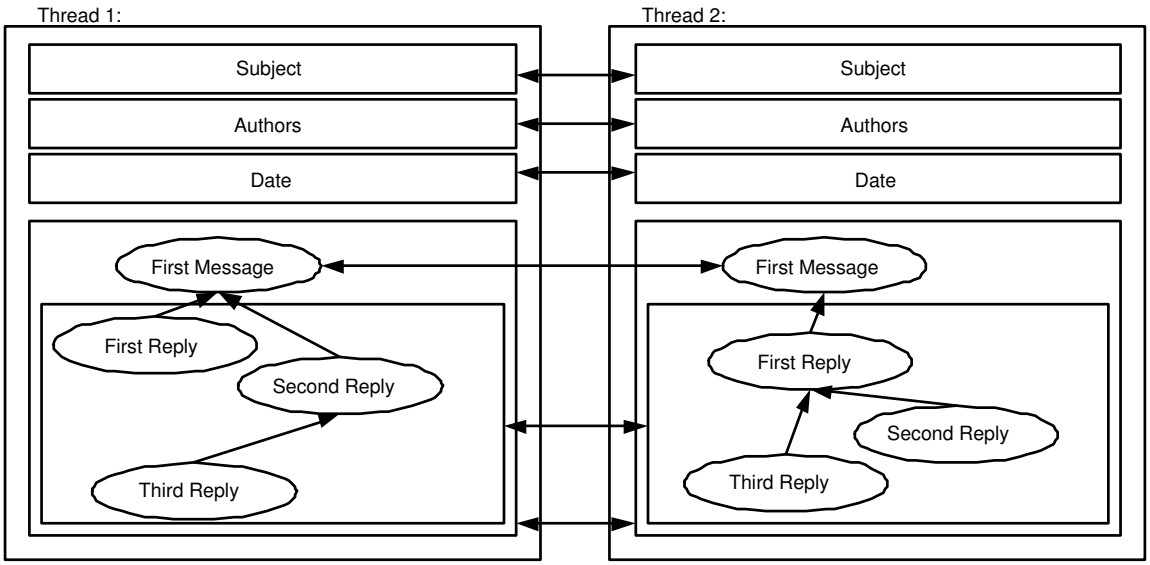

Figure 3.1: Similarity Functions for Two Threads

In particular, for any pair of threads T_i and T_j, we considers the following similarity functions:

1. $sim_m(T_i, T_j)$ measures the similarity between the main contents of two threads. This includes the subject of the thread, quoted text, and unquoted text of all the messages in the thread. It does not include any header field other than the "Subject:" line.

2. $sim_s(T_i, T_j)$ measures the similarity between the subject properties. The subject property, as described before, is just the subject of the first message of the thread.

3. $sim_{mnq}(T_i, T_j)$ is similar to $sim_m(T_i, T_i)$, except that $sim_{mnq}(T_i, T_j)$ excludes quoted text from the content. Quoted text is defined as lines that start with ">". This is done so that

we can safely ignore much of the repeated content in the replies. The quoted paragraphs have usually appeared before in the thread, so logically they are not necessary.

4. $sim_a(T_i, T_j)$ indicates the authorship similarity. As mentioned before, this computes the similarity between the Author properties. This can potentially be used to indicate similarity if the same authors are engaged in a similar conversation across different threads.

5. $sim_f(T_i, T_j)$ is similar to $sim_m(T_i, T_j)$, except we use only the first message for comparison.

6. $sim_r(T_i, T_j)$ is similar to $sim_m(T_i, T_j)$, except we use all messages other than the first one for comparison.

7. $sim_{rnq}(T_i, T_j)$ is similar to $sim_r(T_i, T_j)$, except that $sim_{rnq}(T_i, T_j)$ excludes quoted text.

8. $sim_{date}(T_i, T_j)$ measures the similarity based on the Date property.

9. $sim_{uniform}(T_i, T_j)$ is the baseline similarity function, it always returns 0.5. This is served as the weakest baseline in our experiment.

Each similarity function is normalized to a value between 0 and 1. Suppose the the original similarity function is called $sim'(T_i, T_j)$, then the real similarity function is:

$$sim(T_i, T_j) = \frac{sim'(T_i, T_j) - \min(\forall_{m,n,\, m \neq n} sim'(T_m, T_n))}{(\max(\forall_{m,n,\, m \neq n} sim'(T_m, T_n)) - \min(\forall_{m,n,\, m \neq n} sim'(T_m, T_n)))} \quad (3.1)$$

3.3.2 Computing Similarity Functions

For the similarity functions except for $sim_{date}(T_i, T_j)$ and $sim_{uniform}(T_i, T_j)$, we use a vector space model to compute the similarity between two fields. Let t'_i be the term frequency vector for T_i for the particular perspective. Suppose we have a set of all words $W = \{w_1, w_2, ..., w_n\}$, we can define vector $t'_i = \{v'_{i1}, ..., v'_{in}\}$ where v'_{ij} represents the number of occurrence of w_j in T_i. However, since this term frequency vector t'_i weights each term equally, terms with high frequency, such as common words like "the", "of", etc., would be weighted unreasonably high. Therefore, we want to transform this vector into a weighted term frequency vector, where each

term is weighted according to its frequency and document frequency. We use a variation of the Okapi scoring formula [31, 28] to weight each term. Suppose our final Okapi-weighted vector is $t_i = \{v_{i1}, ..., v_{in}\}$ and we treat each v'_{ij} as term frequency (tf), we can transform t'_i into t_i by setting:

$$v_{ij} = \frac{k_1 \cdot v'_{ij}}{v'_{ij} + k_1((1-b) + b(\frac{dl}{avdl}))} \ln \frac{N - df + 0.5}{df + 0.5}$$

where:

- k_1 and b are constants set to 1.2 and 0.7 accordingly.
- N is the total number of documents in the collection.
- df is the document frequency, or the number of documents that contain the word w_j that has count v'_{ij}.
- dl is the document length. This is the length of the field for the similarity function.
- $avdl$ is the average document length. This is the average field length depending on the similarity function's perspective. For example, the $avdl$ of $sim_s(T_i, T_j)$ is the average length of the subject field.

In order to compute the similarity between two threads, we calculate a simple dot product for the two vectors:

$$sim_x(T_i, T_j) = \sum_{k=1}^{n} v_{ik} \cdot v_{jk} \tag{3.2}$$

where x represents a particular perspective for comparison. For $sim_{date}(T_i, T_j)$, we would like to model the exponential time decay based on the age of the message. Assuming that we have the date for the thread as $date(T_i)$, the similarity function is:

$$sim_{date}(T_i, T_j) = e^{-|date(T_i) - date(T_j)|}$$

It should be noted that the date is measured by day, so two threads posted on the same day would have similarity of 1.

3.3.3 Learning to Combine Similarity Functions

With all the different similarity functions, we would like to combine them in a meaningful way. This can be useful, for example, in a semi-automatic tool for FAQ generation. If a mailing list is in need of a FAQ, a domain expert can first use this system to group some relevant messages together. The system can use this training set to train a new similarity function. This similarity function can then be used to cluster more messages. After new clusters are generated, the expert can verify the clusters, create a new training set, and repeat the process until she is satisfied with the results.

Essentially, given two email threads T_1 and T_2, we would like to find the function $sim(T_1, T_2)$ that can best satisfy our clustering goal. It is worth noting that $sim(T_1, T_2)$ may vary based on user preferences, since no two human beings share the exact same perspectives. We will try to evaluate this important difference in Section 3.5.1 when we construct our test corpus. First, however, we need to generate the training examples for this learning task.

3.3.4 Generating Training Examples

We assume the user has already grouped threads into clusters in $M = \{C_1, C_2, ..., C_{|M|}\}$. For each cluster C_i, there are multiple threads such that $C_i = \{T_{1i}, T_{2i}, ..., T_{|C_i|i}\}$.

In general, we want to maximize the intra-cluster similarity and minimize the inter-cluster similarity. Therefore, if two threads are in the same cluster in the training set, then we should set the ideal similarity to 1 (most similar). Conversely, if two threads are not in the same cluster, we should set the ideal similarity to 0 (not similar).

During training, we want to generate feature matrix X with individual feature vector $X_i = (x_1, x_2, ..., x_n, y_i)$ where $x_i \in [0..1]$ and label $y_i = 0, 1$. We can then create the examples from the training set M using the following algorithm:

We first define the following function:

- $notIn(C_i)$ returns a set of threads that are not in cluster C_i.

1. For each $C_i \in M$:

   (a) For each pair of thread T_{pi}, $T_{qi} \in C_i$ where $T_{pi} \neq T_{qi}$, we do:

      i. If the similarity functions described in Section 3.3.1 are enumerated as sim_1, sim_2, ..., sim_n, we can create the following positive training vector:

$$X_i = (sim_1(T_{pi}, T_{qi}),\ sim_2(T_{pi}, T_{qi}), ...,\ sim_n(T_{pi}, T_{qi}), 1)$$

   (b) For each $T_{pi} \in C_i$:

      i. For each $T_{qi} \in notIn(C_i)$:

         A. We create the negative training example as:

$$X_i = (sim_1(T_{pi}, T_{qi}),\ sim_2(T_{pi}, T_{qi}), ...,\ sim_n(T_{pi}, T_{qi}), 0)$$

The first part of this algorithms leverage the intra-cluster similarity implied by M. It sets the similarity to be 1 for all the pairs of threads within the same cluster. On the other hand, when two threads are not in the same cluster, we assume the similarity is 0.

Given the feature matrix X, we can apply various machine learning algorithms to train for the final similarity function. In our study, we use Linear Regression and Logistic Regression to learn our new composite similarity function. Nevertheless, any kind of distribution-based learning algorithm that takes in numeric values as features can be used to learn this similarity function, as we can see later on in Section 3.3.6.

3.3.5 Combining Similarity Functions using Linear Regression

If we train the feature matrix X using Linear Regression, we would get a weight vector $W = \{w_1, w_2, ..., w_n\}$ that corresponds to the features in feature vector X_i and a constant c. For each pair of threads T_i and T_j, the composite similarity function becomes:

$$sim_{linear}(T_i, T_j) = \left(\sum_{j=1}^{n} w_j sim_j(T_{pi}, T_{qi}) \right) + c$$

This similarity is then normalized using Equation 3.1 to ensure its values be between 0 and 1.

3.3.6 Using Logistic Regression and other Distribution-based Classifier Functions

Given a distribution-based classifier functions such as Logistic Regression, we have to treat the generated examples slightly differently. We can generate the examples as before, but the goal of a distribution-based classifier function is to learn the likelihood of an instance X_i that is classified as label j. This likelihood is written as $P_j(X_i)$.

In this study, we use $P_j(X_i)$ to represent the similarity between two threads. For X_i, there are only two possible values for y_i, 0 and 1. The meaning of the two labels are exactly the same as before. Label 1 indicates that two threads are from the same cluster, and the label is 0 otherwise. The combined similarity is then the value of $P_1(X_i)$, or the likelihood that two threads are from the same cluster.

In Logistic Regression [23], where a parameter matrix B is learned, the likelihood for the instance X_i with label 1 is:

$$sim_{logistic}(T_m, T_n) = P_1(X_i) = \frac{\exp(X_i B_1)}{1 + \exp(X_i B_0) + \exp(X_i B_1)}$$

It is important to note that his framework does not only apply to Logistic Regression, but other learning algorithms that return a distribution function as well. This is true as long as the likelihood $P_1(X_i)$ can be calculated based on the feature vector X_i.

3.4 Agglomerative Clustering

With the similarity functions defined in previous sections, we can perform clustering in order to group similar messages together. Suppose we have a newsgroup G and an empty cluster set M, the original agglomerative clustering method works like the following:

1. For each pair $T_i$and T_j in G, we need to calculate the similarity $s_{ij} = sim(T_i, T_j)$
2. Assign each T_i to a cluster of one node named C_i and add C_i to M.
3. Until $needToStop(M) = 1$, do
   (a) Find pair of cluster C_a and C_b in M where the result of the cluster similarity function $csim(C_a, C_b)$ is maximized. Since we want to have a tight cluster, the cluster similarity function used in our case is the complete link similarity function:

$$csim(C_a, C_b) = \min_{T_i \in C_a, T_j \in C_b} sim(T_i, T_j)$$

   (b) Merge C_a and C_b.

The $needToStop(M)$ function takes in a set of clusters and returns true if the algorithm needs to stop. Traditionally, if we want to have k clusters in an agglomerative algorithm, $needToStop(M)$ is defined as:

$$needToStop(M) = \begin{cases} 1 & \text{if } |M| = k \\ 0 & \text{otherwise} \end{cases} \tag{3.3}$$

However, finding the correct k to stop at is not trivial. Thus, we would like to use a different criteria to stop the clustering process.

Since the task for discovering a frequently discussed topic is to identify very similar questions and their answers, the intuition is to find very tight clusters that gives significant meaning to the complete newsgroup.

An alternatively way to extract different number of clusters is to traverse the agglomerative tree and extract clusters that have exceed a certain intra-cluster similarity threshold. A cluster's

intra-cluster similarity (ICS) is calculated as:

$$ICS(C_a) = \frac{1}{|C_a|^2} \sum_{T_i \in C_a, T_j \in C_a} sim(T_i, T_j) \tag{3.4}$$

Intuitively, we want to find clusters with similar level of similarity. This enables us to present the clusters and show the *popularity* property. Essentially, we want to find clusters with similar "cohesiveness" but with different sizes. For example, given two clusters C_a and C_b with similar ICS where $|C_a| > |C_b|$, we would prefer cluster C_a as a topic with more *popularity*. Since even though C_b has the similar "cohesiveness" as C_a, C_a has more threads. More details on this threshold setting technique can be found in Appendix B.

We use Equation 3.3 in our evaluation of our clustering algorithm and similarity functions. The second thresholding function based on ICS is used in CEES' Conversation Map.

3.5 Evaluation

> "It is easier to perceive error than to find truth, for the former lies on the surface and is easily seen, while the latter lies in the depth, where few are willing to search for it." – Johann Wolfgang von Goethe

3.5.1 Corpus

Traditionally, finding the right corpus for clustering performance evaluation has been difficult. It is even more so in CEES, since we want to evaluate the performance of intra-newsgroup clusters. First of all, the Enron dataset[20] has been designed for personalized email research, so it does not exactly fit what we want to do. We have also considered the original 20 newsgroups dataset[22] as well. However, since the corpus does not contain subtopic groupings within each newsgroup, we can not use that either.

Comparing to newsgroup clustering, subtopic clustering is a more difficult task due to the following reasons:

1. Main topic keywords have similar word distribution among different subtopics within the same newsgroup.

2. Authors in header fields do not contribute much information to the machine learning algorithm, for an author can talk about multiple topics within the newsgroup.

3. The inter-cluster similarity is higher among subtopic clusters, since the subtopics are more closely related to each other.

Due to the lack of appropriate corpus, we need to construct one for our needs. In particular, we have gathered all the messages from the class newsgroups for the Computer Science Department at University of Illinois at Urbana-Champaign (UIUC) for the 2004 Fall semester. For each semesters, a class newsgroup is created for each Computer Science class at UIUC. Professors, TAs, and students can freely post messages to any group. Previous messages are deleted in the beginning of a new semester. Thus, we have a complete set of messages from the beginning to the end of semester. This allows us to see the discussion trends more easily.

In particular, we have chosen the following three classes in Table 3.1 for our experiment.

Newsgroup	Title of Class
class.cs225	Data Structure & Software Principles
class.cs473	Advanced Algorithms
class.cs475	Formal Models of Computation

Table 3.1: CEES Corpus

More specifically, the CS225 is a lower-division data structure course. CS473 and CS475 are two upper-division/graduate-level algorithm classes. Some topics, such as NP-complete problems, are taught in both CS473 and CS475. There should be no obvious overlapping topics between CS225 and the upper-division algorithm classes. Table 3.2 and 3.3 show the statistics for the three newsgroups collected for the 2004 Fall semester.

Newsgroup	Size (KB)	Messages	Threads	Messages/Thread	Size (KB)/Message
class.cs225	4211.5	2534	806	3.14	1.66
class.cs473	975.8	349	146	2.39	2.80
class.cs475	1765.7	1019	314	3.24	1.73

Table 3.2: Message Statistics

Newsgroup	# of Unique Authors	Messages/Author
class.cs225	242	10.47
class.cs473	49	7.12
class.cs475	94	10.84

Table 3.3: Author Statistics

As one can see, the newsgroups vary in their sizes. CS473 is selected as the newsgroup with relatively low traffic, CS475 with medium traffic, and CS225 with high traffic.

The number of unique authors is a count of unique email addresses in the "From:" field for all the messages in the group. On average, an author who posted on the newsgroup would make close to 10 posts per semester.

3.5.1.1 Identifying Subtopics

Three taggers have participated in creating the corpus for our evaluation. In order to simulate the FAQ creation experience, we assign one tagger per newsgroup as the domain expert. All taggers are students who have taken the class before or have extensive knowledge in the class subject. Each tagger then separately create subtopics within the newsgroup. By design, there is no definite criteria for creating a subtopic. A tagger is free to choose the appropriate subtopic coverage. There is no restrictions on the number of subtopics or how narrow a subtopic should be.

Each message is presented in its threaded form. A tagger can only assign a complete thread to a subtopic. In addition, taggers must assign all the threads in their respective group to the subtopics. For the sake of simplicity, one thread can only be assigned to one subtopic. Table 3.4 shows the number of subtopics created by the taggers.

Newsgroup	# of Subtopics	Avg. # of thread/topic
class.cs225	26	31.0
class.cs473	16	5.6
class.cs475	15	20.9

Table 3.4: Number of Subtopics

As one can see, subtopics in different groups have different level of granularity. This is expected due to differences in tagging behavior.

3.5.2 Evaluation Measures

In order to evaluate our clustering performance, we use the entropy-based class conformation measure as described in [15]. There are two quantitative measures in this approach: Cluster Entropy and Class Entropy. Cluster Entropy measures the entropy (or the degree of mixture) of the resulting clusters from the clustering algorithm. Class Entropy, on the other hand, compares the actual clusters from a golden set and measures how many of the elements in the golden set cluster are assigned to different resulting clusters.

More precisely, assuming Ec_i represents the entropy for a cluster in the results, it can be expressed as

$$Ec_i = -\sum_j \frac{n(l_j, c_i)}{n(c_i)} \log \frac{n(l_j, c_i)}{n(c_i)}$$

where $n(l_j, c_i)$ is the number of samples in cluster $c_i$that are assigned to label l_j and $n(c_i)$ represents the number of samples in c_i. In other words, $\frac{n(l_j,c_i)}{n(c_i)}$ can be considered as the probability of picking of sample with label l_j from cluster c_i.

Given Ec_i, the Cluster Entropy measure is the weighted average of Ec_i for all the clusters:

$$Ec = \frac{1}{\sum_i n(c_i)} \sum_i n(c_i) Ec_i$$

As for Class Entropy, we compute the entropy for each label l_j:

$$El_j = -\sum_i \frac{n(l_j, c_i)}{n(l_j)} \log \frac{n(l_j, c_i)}{n(l_j)}$$

where $n(l_j, c_i)$ represents the number of samples in cluster c_i with label l_j and $n(l_j)$ is the number of samples with label l_j in the actual clusters. Similar to Cluster Entropy, the overall Class Entropy is the weighted average of El_i for all the labels:

$$El = \frac{1}{\sum_i n(l_i)} \sum_i n(l_i) El_i$$

In general, Cluster Entropy decreases when the number of clusters increases. Conversely, Class Entropy tends to increase when the number of cluster increases. This is true intuitively since one can imagine that the "mixture" in a smaller cluster is probably going to be less than a larger cluster. For the extreme case, Cluster Entropy would be 0 when the number of resulting clusters equals the number of samples in the set. This is easily seen since there is one sample per cluster, thus $\frac{n(l_j,c_i)}{n(c_i)} = 1$ and so $\forall i$, $Ec_i = 0$. On the other extreme, Class Entropy would be 0 when there is only one result cluster, since $\frac{n(l_j,c_i)}{n(l_i)} = 1$ and $\forall i$, $El_i = 0$. In [15], a β parameter is used to weight El and Ec. Thus, the Combined Entropy becomes:

$$Ecl(\beta) = \beta \cdot Ec + (1 - \beta) El \tag{3.5}$$

where $\beta \in [0, 1]$. However, [15] only mention that β is a weighted parameter that can be set by the experiment. It does not mention how to set β specifically. We will discuss how to set this parameter in our experiment in the next section.

3.5.2.1 All-in-All v.s. All-in-One

During our experiment with CEES, we discovered that in order to perform a fair measure, we need to set β depending on the actual clusters. Essentially, we want to set β such that the two extreme clustering cases have the same penalty. In other words, if we have n samples in the set

that we want to cluster, and our clustering results also have n results (all-in-all), we want the overall entropy measure to be the same as if we have 1 cluster for the n samples (all-in-one).

Therefore, given the actual clusters, we can measure the two extreme cases. Assuming that we have the all-in-all case, where the $Ec_{all-in-all} = 0$, we want to measure $El_{all-in-all}$ given the actual clustering result. Conversely, for the all-in-one case, where $El_{all-in-one} = 0$, we want to measure $Ec_{all-in-one}$. We can then set β to:

$$\beta = \frac{El_{all-in-all}}{El_{all-in-all} + Ec_{all-in-one}}$$

If we use this form to calculate β, the two extreme cases will share the same entropy. The β values for each group in our corpus is given in Table 3.5.

Newsgroup	β
class.cs255	0.6072
class.cs473	0.6254
class.cs475	0.5323

Table 3.5: β Values for the CEES Corpus

3.5.3 Clustering Results

Since setting different number of clusters yields different entropy scores, we want to compute the clustering performance based on the average of the entropy scores across different number of clusters. Since we use agglomerative clustering, we can evaluate based on any number of clusters from 1 to n, where n equals the number of threads in the group.

For each group that we would like to cluster, we take 21 snapshots (0 to 20) during the agglomerative clustering process at even intervals. We then average the Cluster Entropy, Class Entropy, and Combined Entropy based on the 21 snapshots. The β values used in the Combined Entropy calculation for the group are given in Table 3.5. Figure 3.2 shows how the Cluster Sampling Iteration affects the number of resulting clusters for all three groups. Essentially, the higher the Cluster Sampling Iteration, the lower the number of clusters. The number of cluster decreases linearly in proportion to the maximum number of the threads in the group.

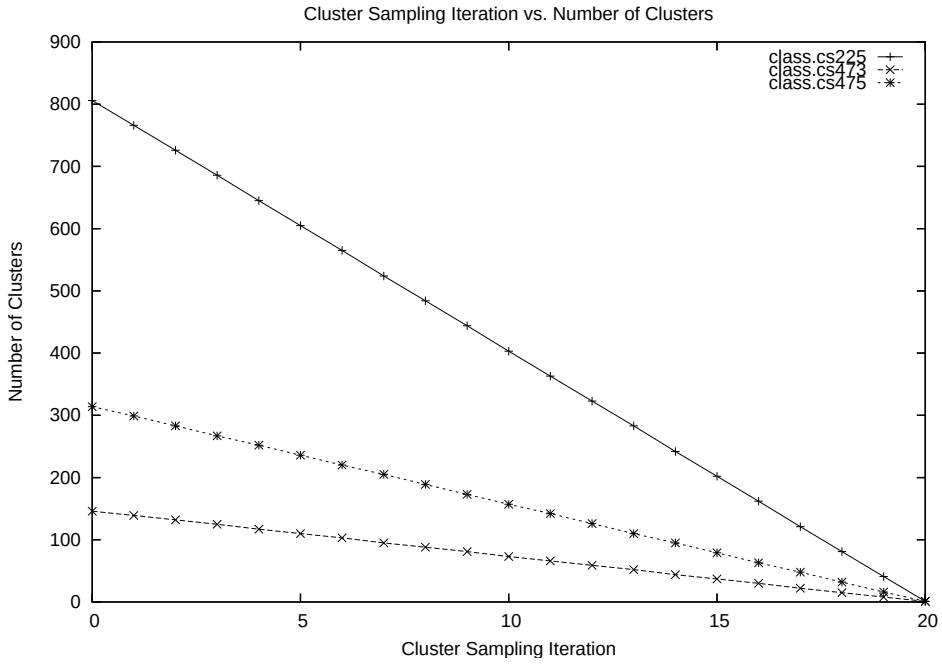

Figure 3.2: Cluster Sampling Iteration vs. Number of Clusters

Table 3.6 presents the clustering results for each of the similarity function specified in Section 3.3.1. In particular, "[newsgroup] LinearReg" or "[newsgroup] Logistic" represents the a combined similarity functions using all the other similarity functions except for "uniform" (authors, contents, content no quote, date, first mesg, rest mesg, rest mesg no quote, and subject). The [newsgroup] prefix implies that the similarity function has been learned from the [newsgroup]'s training set. For example, "class.cs225 Logistic" is a similarity function that uses Logistic Regression to learn from the actual class.cs225 clusters using the algorithm in Section 3.3.6. The columns in the table describe the Combined Entropy and the Average Combined Entropy. The Average Combined Entropy is the average value for the two groups indicated in the column headings. This is similar to performing a 3-way cross validation for the CS225, CS473, and CS475 newsgroups.

	Combined Entropy			Combined Entropy Average		
Similarity Function	225	473	475	225 + 473	225 + 475	473 + 475
class.cs225 LinearReg	*1.589*	1.159	1.427	*1.374*	*1.508*	1.293
class.cs225 Logistic	*1.568*	1.163	**1.397**	*1.366*	*1.483*	**1.280**
class.cs473 LinearReg	1.634	*1.136*	1.477	*1.385*	1.556	*1.307*
class.cs473 Logistic	1.657	*1.154*	1.481	*1.406*	1.569	*1.318*
class.cs475 LinearReg	1.613	1.179	*1.415*	1.396	*1.514*	*1.297*
class.cs475 Logistic	**1.592**	1.195	*1.403*	**1.394**	*1.498*	*1.299*
authors	1.944	1.204	1.738	1.574	1.841	1.471
contents	1.662	1.141	1.486	1.402	1.574	1.314
contents no quote	1.666	**1.131**	1.462	1.399	1.564	1.297
date	1.855	1.400	1.677	1.628	1.766	1.539
first mesg	1.629	1.177	1.431	1.403	**1.530**	1.304
rest mesg	1.694	1.137	1.517	1.416	1.606	1.327
rest mesg no quote	1.785	1.188	1.533	1.487	1.659	1.361
subject	1.672	1.155	1.483	1.414	1.578	1.319
uniform	1.961	1.491	1.765	1.726	1.863	1.628

Table 3.6: Clustering Performance

One can see from Table 3.6 that using either Linear or Logistic Regression would exceed or match the best performance of any single similarity function. We can also see that Logistic and Linear Regression have very similar performance. The numbers in italic are generated when the group is testing and training on itself, either entirely or partially. The bold numbers are the best run excluding the numbers in italic. Unsurprisingly, due to overfitting, training and testing on the same group yields the best performance. There is no single similarity function that performs the best across all the groups.

Interestingly, even with three different newsgroups and three different taggers, Table 3.6 shows that training the similarity function on one group can be used to learn the parameters of the combined similarity function for another group. This shows that our learning algorithm can be effective in learning the parameters of the general similarity function. Furthermore, since the tagging behavior for a single group is consistent, one would imagine that the learning algorithm can perform even better when training on a subset of the newsgroup and apply the parameters on the similarity function to cluster the other messages in the same group.

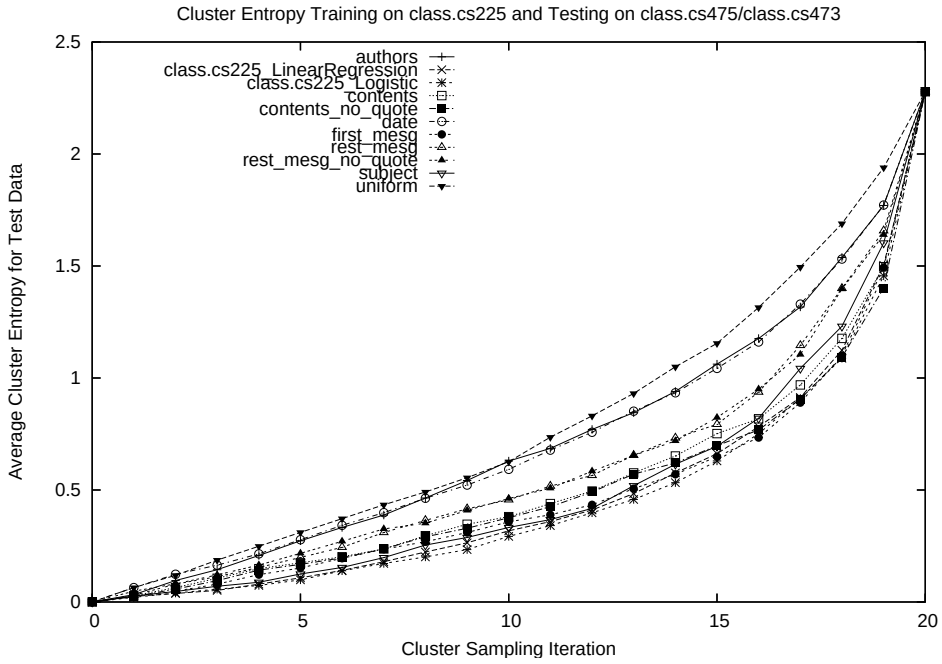

Figure 3.3: Cluster Entropy for Training on class.cs225.

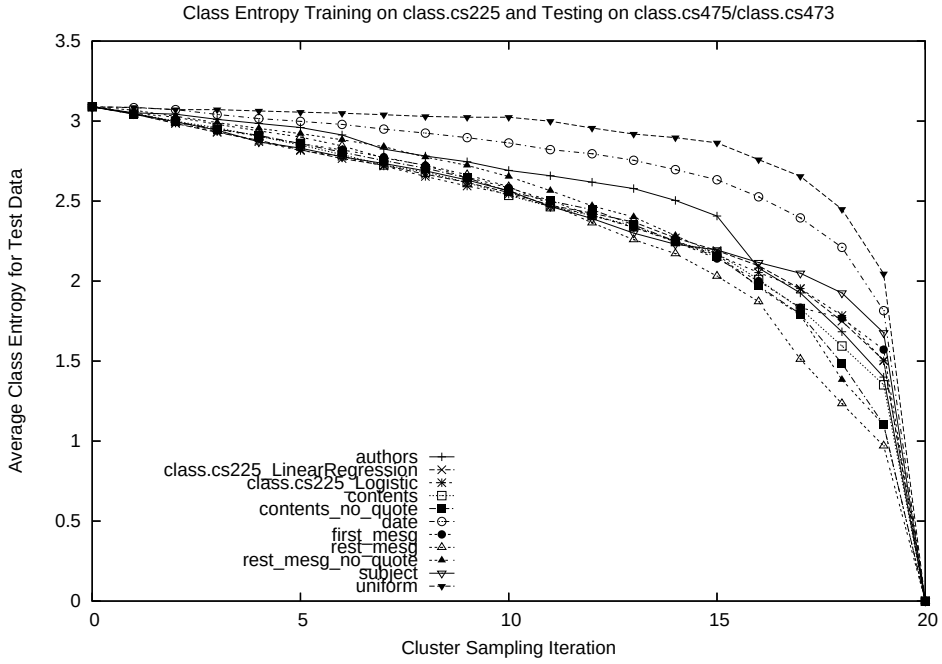

Figure 3.4: Class Entropy for Training on class.cs225.

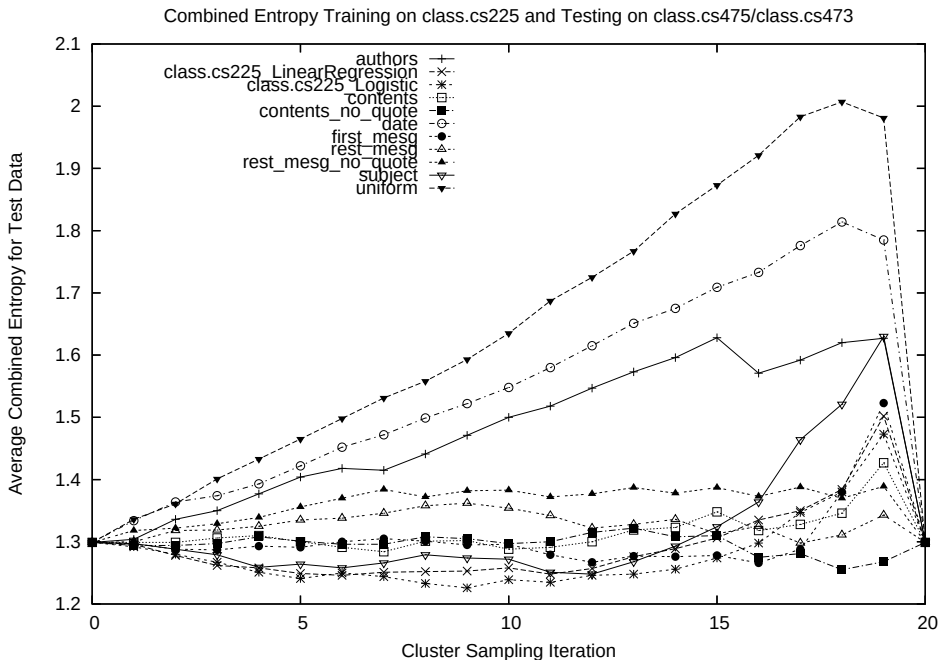

Figure 3.5: Combined Entropy for Training on class.cs225.

Figure 3.3, 3.4, and 3.5 illustrate the clustering performance when training the similarity function on class.cs225 and testing on the other two groups. Using the algorithm described in Section 3.3.4, more than 610,000 examples are generated from the tagged class.cs225 group. The large number of examples tends to overfit the class.cs225 group, as we can see in Table 3.6 that the performance when applying the similarity function on itself produces much better results (Combined Entropy is 1.589 for Linear and 1.568 for Logistic Regression). Nevertheless, one can see that Logistic and Linear Regression perform reasonably well on the test set (class.cs475 and class.cs473) on both Cluster Entropy and Class Entropy (Figure 3.3 and 3.4).

Figure 3.6, 3.7, and 3.8 show the result for training on class.cs473, similarly for what Figure 3.9, 3.10, and 3.11 do for class.cs475. Due to the small size of class.cs473, there are only a little over 10,000 training examples. The performance in this case may suffer due to the small number of examples. The medium-size group class.cs475 produces around 90,000 examples and produces the best performance on group class.cs473 and class.cs225.

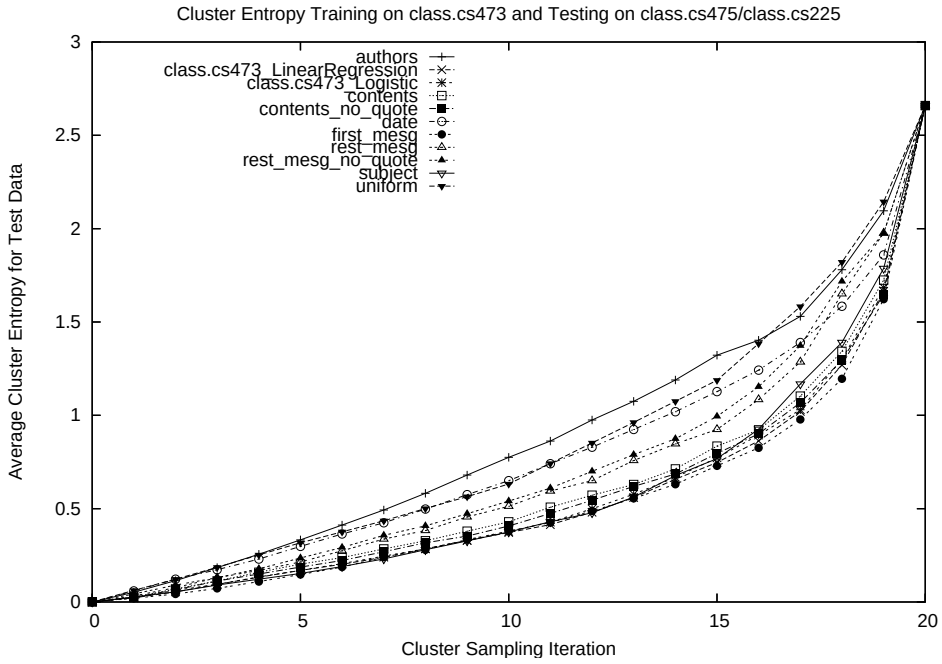

Figure 3.6: Cluster Entropy for Training on class.cs473.

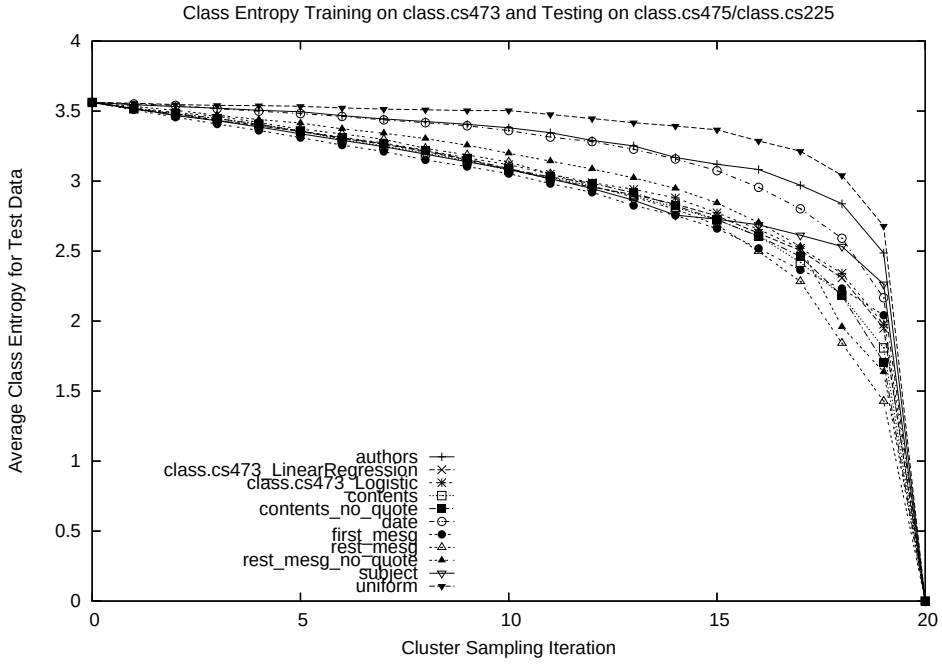

Figure 3.7: Class Entropy for Training on class.cs473

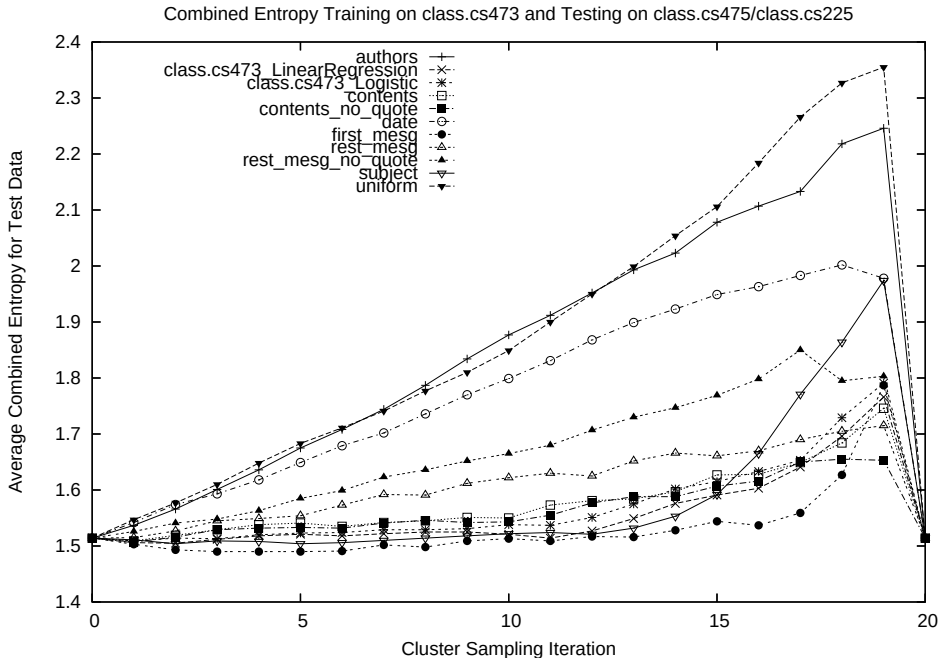

Figure 3.8: Combined Entropy for Training on class.cs473.

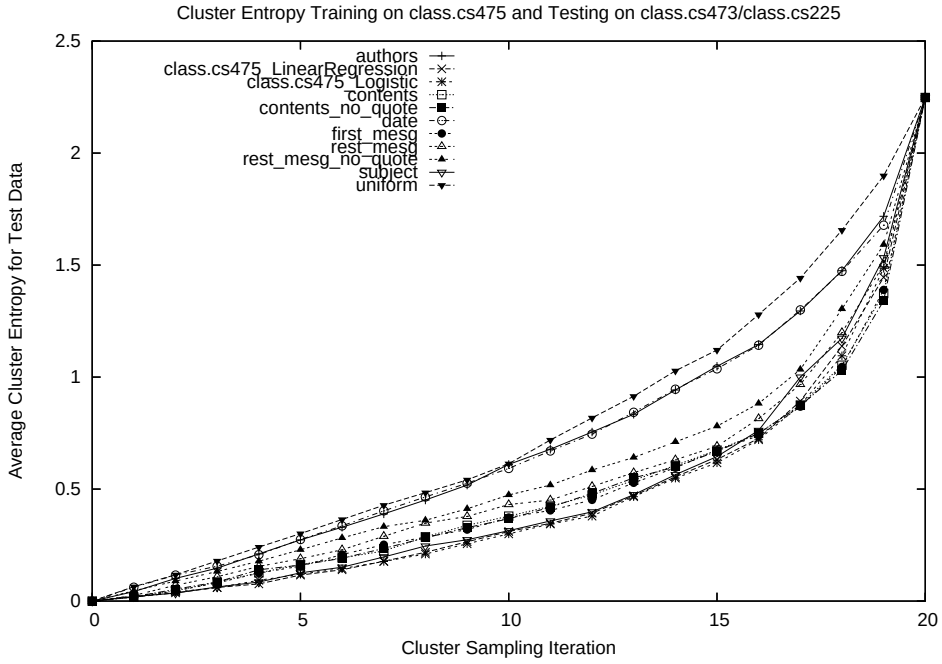

Figure 3.9: Cluster Entropy for Training on class.cs475.

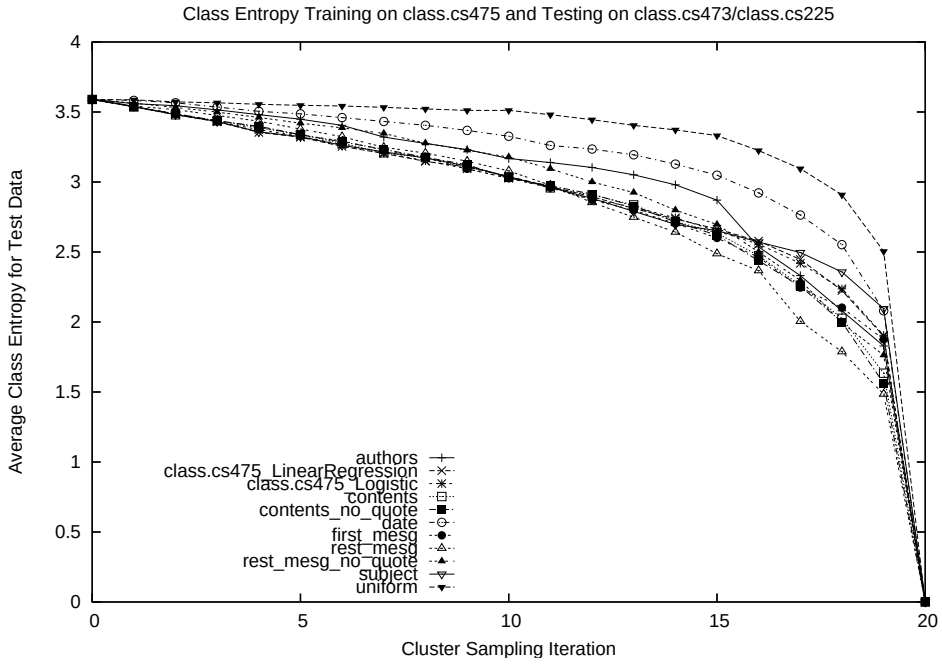

Figure 3.10: Class Entropy for Training on class.cs475

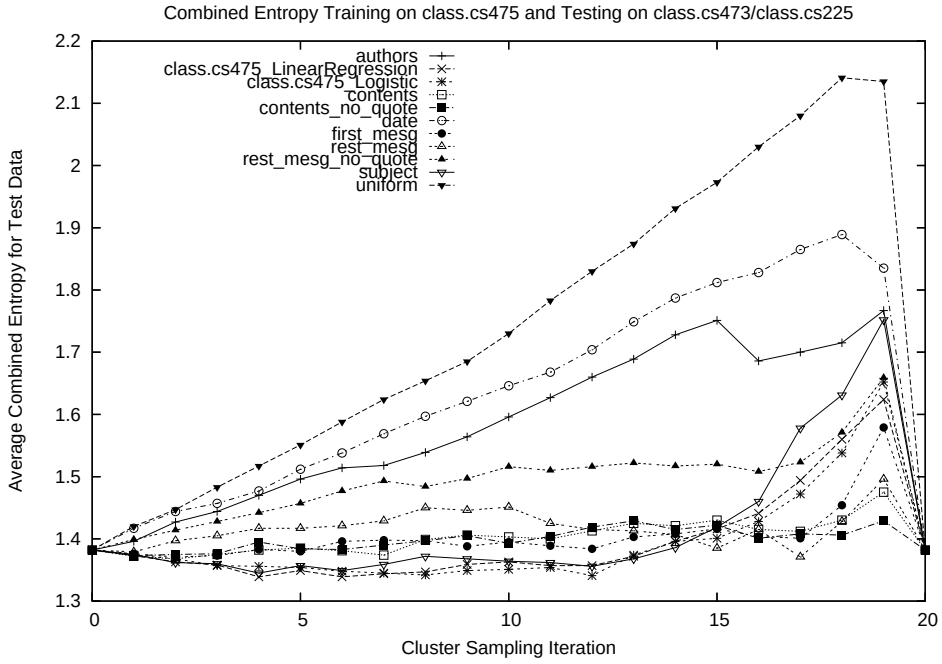

Figure 3.11: Combined Entropy for Training on class.cs475.

Among the unsupervised similarity functions, "first mesg", "subject", and "contents no quote" are the best functions for this clustering task. One should note that "subject" tends to produce poor performance when the number of clusters becomes small. This may be due to the relatively short content in the "Subject:" field. Using complete link and small weighted vectors, there are probably more outliers when there are more messages in a cluster.

3.6 Summary of Findings

Our experiment has shown that the trained similarity function can be effective in the subtopic clustering task. In particular, the composite similarity function exceeds or matches the performance of any single unsupervised similarity function on average. Also, we can see this performance gap gets larger when the the number of cluster increases. This is desirable for our task, since we would like to focus on smaller subtopics in order to find tight conversation clusters.

It is worth emphasizing that the three newsgroups that we use in our corpus have very different subtopics. The newsgroups are also tagged with different individuals without any criteria. Nevertheless, our experiment has shown that the learned weights can be carried over to another group as well.

The size of the training examples may also play a row in the learning performance. As we can see, the performance for training on class.cs225 and class.cs475 outperform training on class.cs473.

We have shown in this chapter that we have a way to cluster similar conversations, based on the trainable similarity function. However, this is only one piece of the puzzle since it can only be used to address the *similarity* property. We will try to fulfill the remaining properties (*coverage, popularity*, *freshness*, and *persistence*) in the following sections.

Chapter 4

Conversation Map

4.1 Motivation

Given a set of conversation clusters, our goal is to display them to the users in a coherent way. Traditionally, IR applications show their clustering results through a flat list[2, 1], where each item in the list represents a cluster. Users can then browse through the clusters by clicking on the titles.

This visualization technique has many limitations. First, it is hard to see the importance of each cluster. Usually, the number of items in the cluster can serve as an indication of importance. Existing techniques usually put the size of a cluster next to the cluster title. The more important clusters are not usually made to be more obvious.

Existing techniques also has no way to show the logical and sequential order of the clusters. Conversation clusters are unique in the sense that conversations are recorded in a particular order. Conventional listing do not show user the temporal dimension that makes the clusters easier to read.

4.2 Treemap and Conversation Map

Treemap[30], originated from HCI research, can be a potential solution to those limitations. Treemap is an effective way to visualize hierarchical information in a 2-D space. In the simplest

sense, Treemap provides visualization for a tree node using two dimensions, the size of the rectangle and the color of the space. A larger space usually indicates higher importance for the node. The color dimension can also be used to indicate the relative values of another property. The values for the two properties (size and color) are configurable for different applications.

If we consider each conversation cluster as a tree node and the threads in it are the leaves, we can use Treemap to display our conversation clusters easily. Since users may be more interested in active threads than quiet ones, we use the number of messages in each thread as the value for the size dimension. This makes an assumption that the number of messages in the thread can somewhat correlate with the *coverage* property of the FAQ.

On the other hand, we use a gradient of two colors to represent the age for a given thread. In our case, we use red and green. Red indicates the thread is relatively older, and green shows that the conversation has occurred recently.

We call this cluster representation Conversation Map (CM). CM leverages the Treemap rendering engine to draw appropriately-sized rectangles for the conversation clusters. Using this scheme, the users can easily detect the more prominent conversations clusters, since they take up more area on the map. Using only two levels of the tree hierarchy, each cluster is displayed in a column, where the agglomerative cluster hierarchy is flattened for the cluster. A thread is represented by a row, or rectangle, in the column. Figure 4.1 shows an example of a CM. If a user moves her mouse on top of a rectangle, she can see the subject, author, date, and the number of messages in the thread.

Treemap, however, can not solve all of our problems. In particular, Treemap has no way to represent the sequential nature of the conversation clusters. Although we display the threads dated at different time periods with different colors, it is often difficult to follow the temporal order of the conversations.

One important contribution that we make to this visualization technique is to add two temporal dimensions to Treemap: the *Intra-Cluster Time* and *Inter-Cluster Time*.

Since each conversation within a cluster has a time stamp, we can sort this date so the threads appear from the oldest to the newest. This represents the *Intra-Cluster Time* dimension. In CM, this dimension is represented vertically for each column. The top-most thread

for a cluster indicates the oldest within the cluster, and the bottom-most represents the newest message in the cluster. This enables the interface to show both the *persistence* and *freshness* property of the cluster at the same time. For example, if a column has the same color throughout, then the threads within the conversation clusters have been discussed at roughly the same time. Conversely, if the column has many different colors, this means the cluster has the *persistence* property, where the topic has been discussed at different time points. In Figure 4.1, one can get an idea that "Pumping Lemma" is a persistent topic for CS475. After all, the topic has been discussed extensively from the beginning to the middle of the semester, and then once again at the very end, probably due to the final examination.

The *Inter-Cluster Time* property compares the the time dimension among clusters. This is represented by the horizontal dimension on the map. Given a cluster C with n threads where $C = \{T_1, T_2, ..., T_n\}$, the *Inter-Cluster Time* for the cluster is just the average time for all the threads within C:

$$time(C) = \sum_{i=1}^{n} \frac{1}{n} time(T_i)$$

where $time(T_i)$ equals the time for the first message of the thread. This dimension is represented by the horizontal axis of Treemap, sorted by $time(C)$ from oldest to newest. One can see from Figure 4.1 how CM represents the *Inter-Cluster* dimensions.

The slider below the map let user set the similarity threshold as in Equation B.1. The more the user move the slider to the right, the more similar the clusters would be. One should notice that CM filters out clusters with just one thread, for we are not interested in stand-alone threads, but conversations that are frequently discussed. Figure 4.2 and 4.3 show a map with the slider moved further to the right for the same newsgroup. Now, one can see the tighter clusters more prominently. Since we have removed all the stand-alone clusters, fewer messages in the group are shown in the map. This allows users to "zoom" into the more similar conversations. For example, one can now identify the discussions about TA office hours, which happen in the beginning of the semesters (all in red).

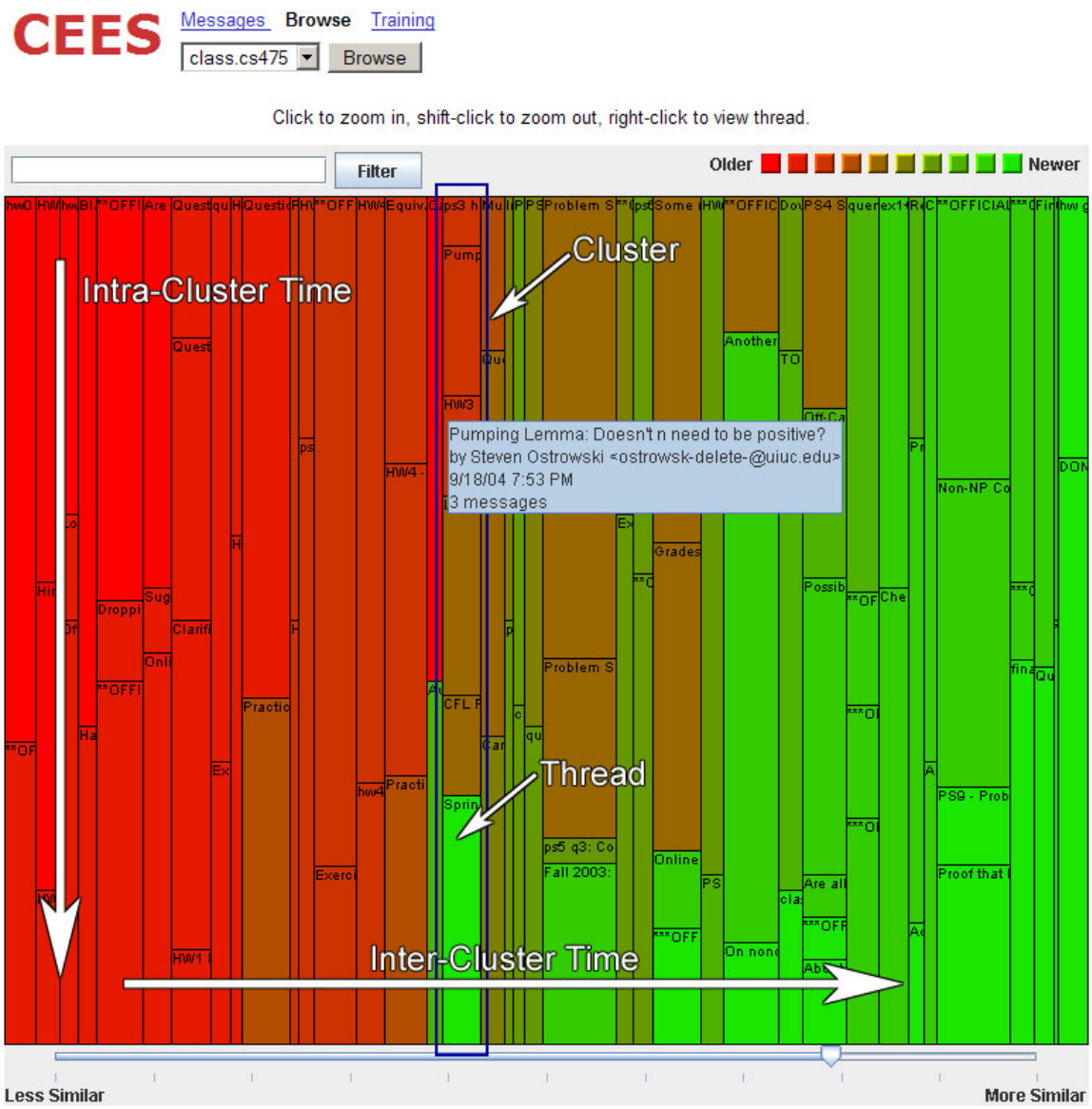

Figure 4.1: Example of a Conversation Map

4.3 Message Filtering

We also support on-the-fly filtered clustering. Given a query, we can first perform traditional information retrieval to find the relevant threads. Then, we can cluster the subset like what we do to the complete group. Figure 4.4 shows an example for this feature when the user typed in "final exam" as the query. The returning results are grouped into two clusters, based on the

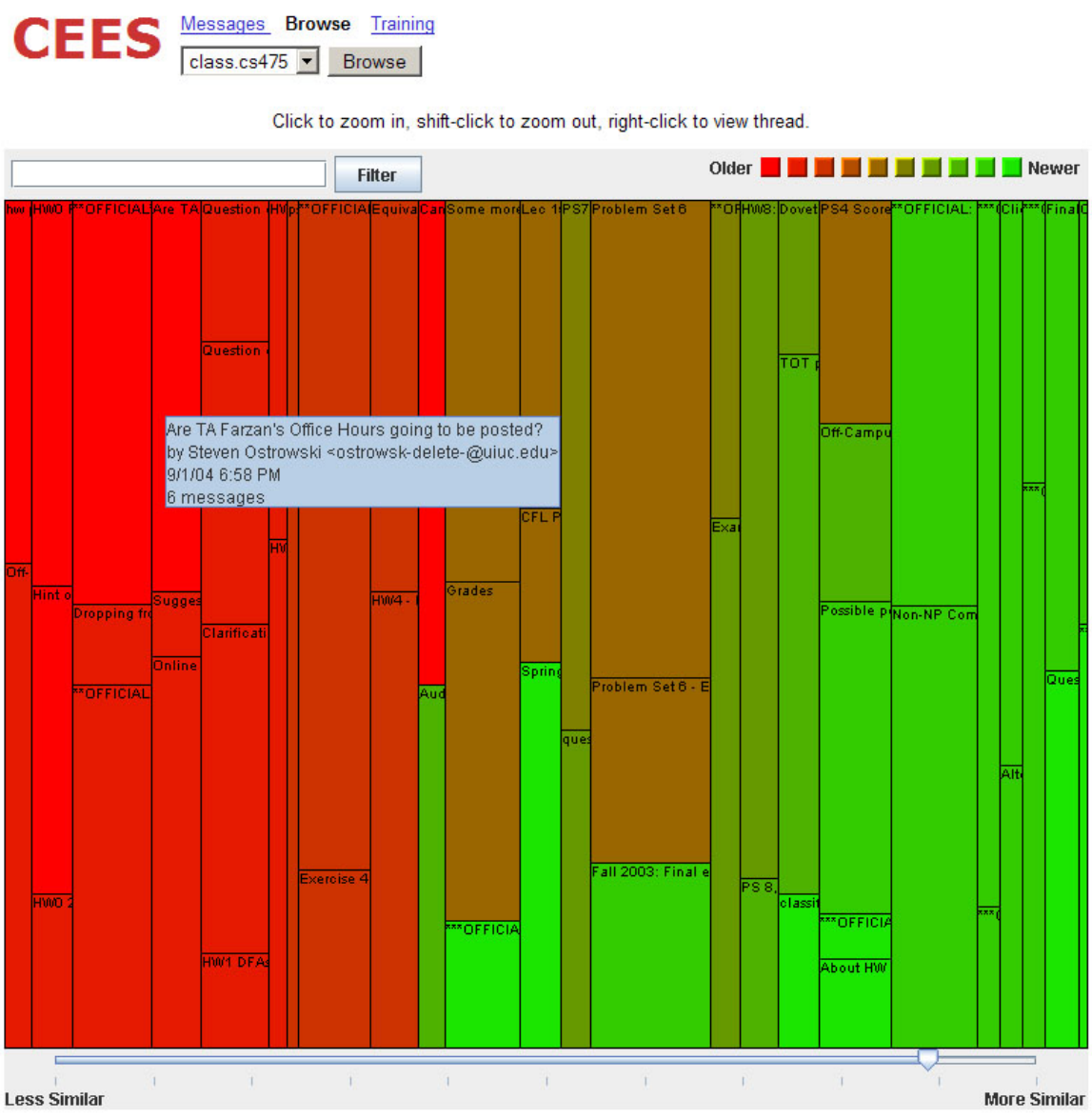

Figure 4.2: More Similar Clusters

threshold that the user set from the slider. The sizes of the rectangles are now determined by the relevance score for the query "final exam".

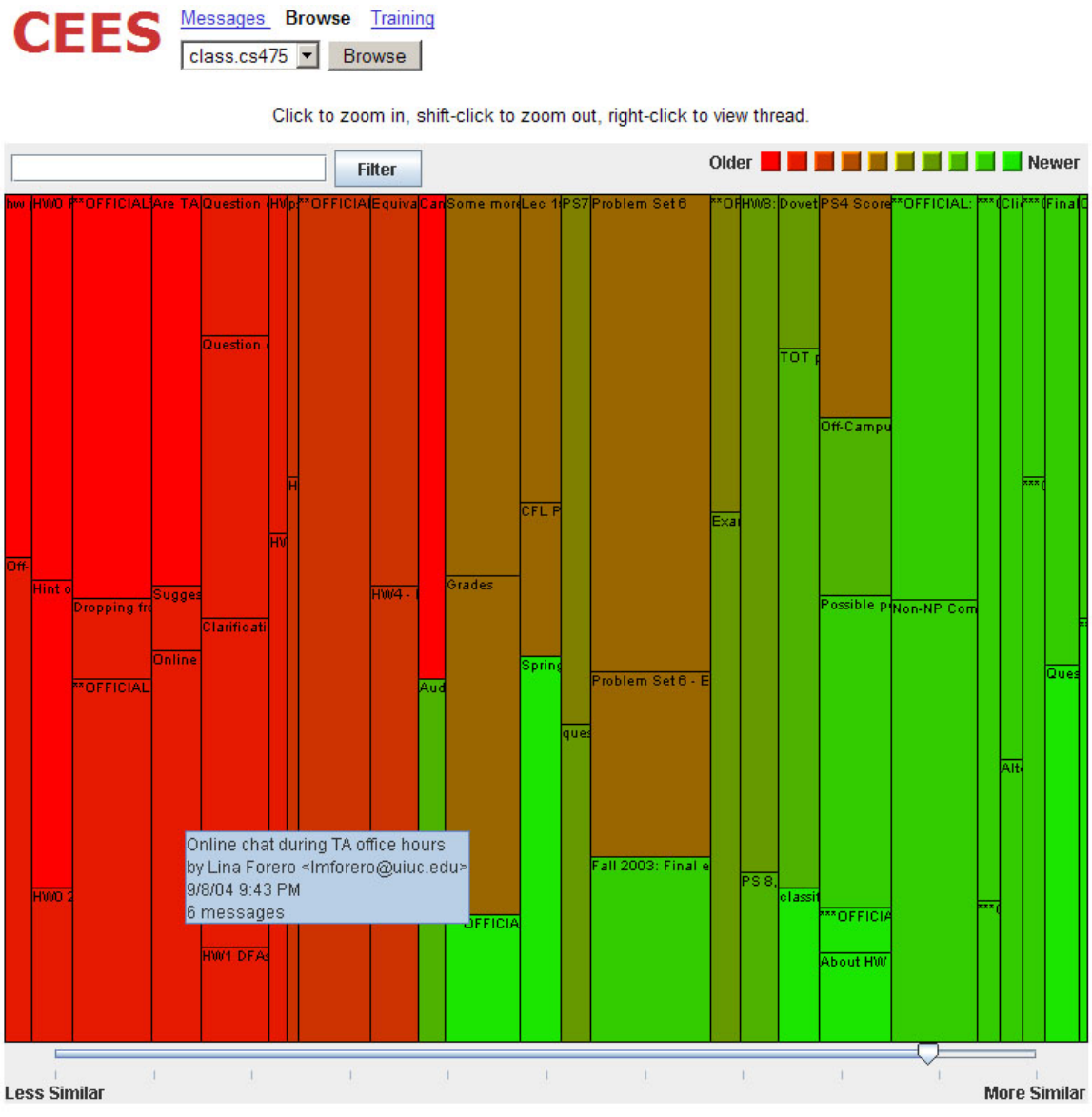

Figure 4.3: More Similar Clusters 2

4.4 Summary of Findings

Conversation Map is based on the fundamental principles of the Treemap interface, which has been widely used in many interesting visualization and mining applications. In addition, we have added our own unique temporal dimensions that has made CM even more useful for summarizing conversations. The similarity threshold, which let user zoom into more similar

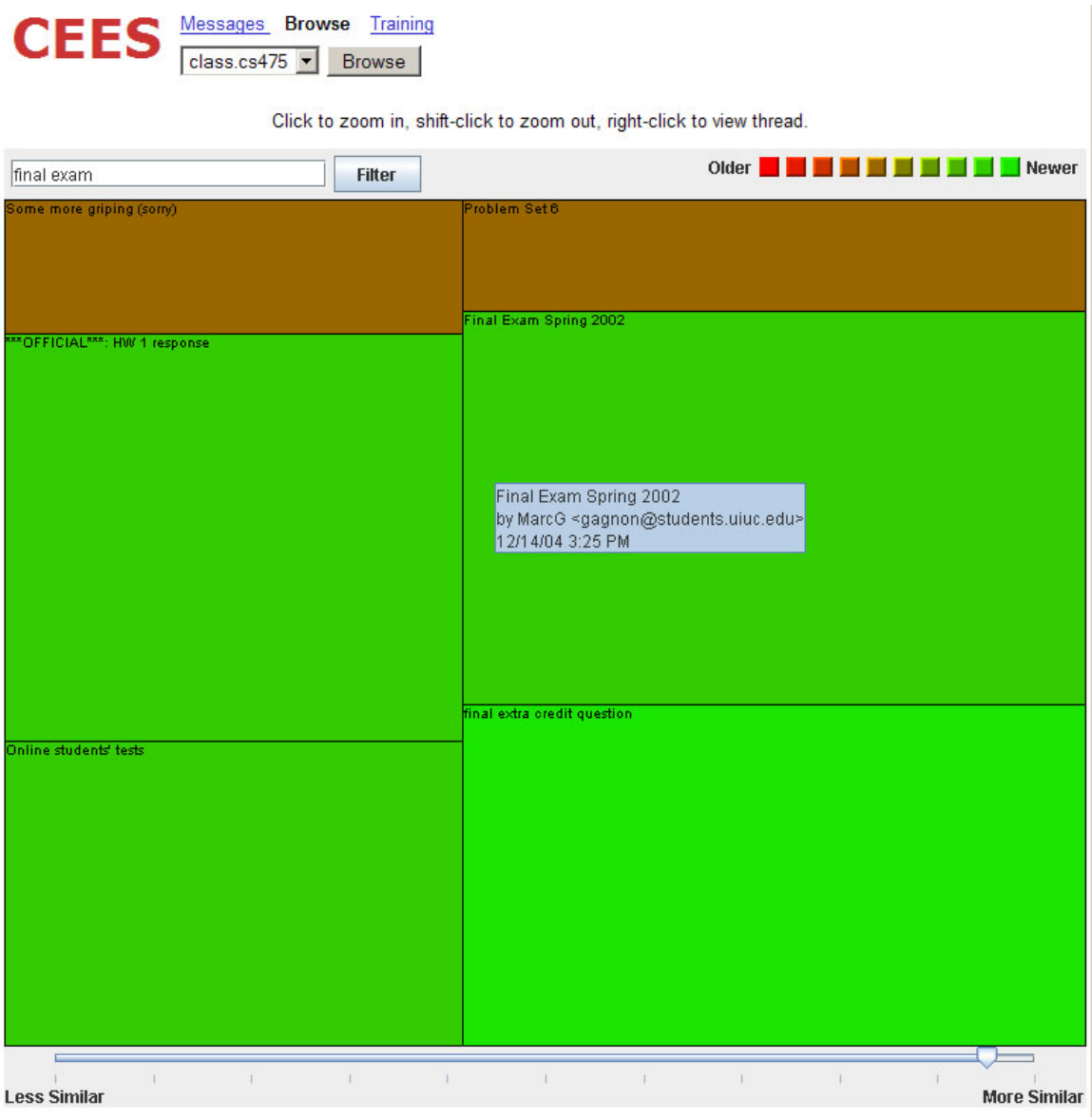

Figure 4.4: Clustering Results for a Query

messages, can be set by simply moving a slider. This simple interface gives maximum flexibility to the users who like to "scale" the similarity of the discussions.

Given we have a way to cluster conversations and a way to display them visually, we need to mention the underlying system that tie everything under one umbrella: CEES, or Conversation Extraction and Evaluation Service.

Chapter 5

Conversation Extraction and Evaluation Service

> "The mother art is architecture. Without an architecture of our own, we have no soul of our own civilization." – Frank Lloyd Wright

5.1 Overview

Due to the lack of an integrated toolkit for processing, indexing, and clustering email data, CEES, or Conversation Extraction and Evaluation Service, was being constructed to fulfill such role. One of CEES' goals is to construct a framework that can help researchers do more innovative things and less redundant grunt work.

CEES aims to provide tools that are capable of solving the common challenges and nuances in email-related research. For example, message threading is a well-established problem and has been addressed in [36]. The implementation of this algorithm, however, is not trivial due to the difficulties in parsing email messages and messy details in the mail transfer protocol. CEES has implemented this algorithm and provided APIs to process messages in tree-like structures.

More specifically, CEES is designed to support the following requirements:

1. *Complex Message Processing*

Since email messages are somewhat difficult to process in a robust way, a framework can vastly simplify storing and parsing messages.

2. *Simple Message Querying*

   Due to the complexity of indexing, querying, and processing, we want to have a simple yet powerful way to query for a message or a group of messages.

3. *Flexible Clustering and Evaluation Framework*

   CEES similarity-learning algorithms and clustering methods require the infrastructure to be flexible enough to accommodate for different experiments.

4. *User Interface that Integrates Tagging, Searching, and Browsing*

   Since we need to tag our corpus quickly, we need a user interface that can speed up this process. CEES also needs to provide the underlying GUI framework in order to integrate searching and browsing. This includes showing the clusters through the Conversation Map.

With those requirements in mind, we now are ready to describe an architecture that can satisfy those requirements.

5.2 Architecture

Since we do not want to reinvent the wheel, CEES is built on a foundation of many popular open source projects such as Tapestry[29], Lucene[14], Hibernate[4], and Weka[34]. Those libraries can save researchers time when they find the need for a rich web-based user interface framework, a fast index implementation, a simplified access abstraction to the relational database, or an implementation of a machine learning algorithm.

In addition, CEES attempts to integrate with those components seamlessly and provides a useful library for indexing, querying, and clustering email messages. Figure 5.1 illustrates the high-level CEES architecture. In the center, the CEES IR library provides the main clustering,

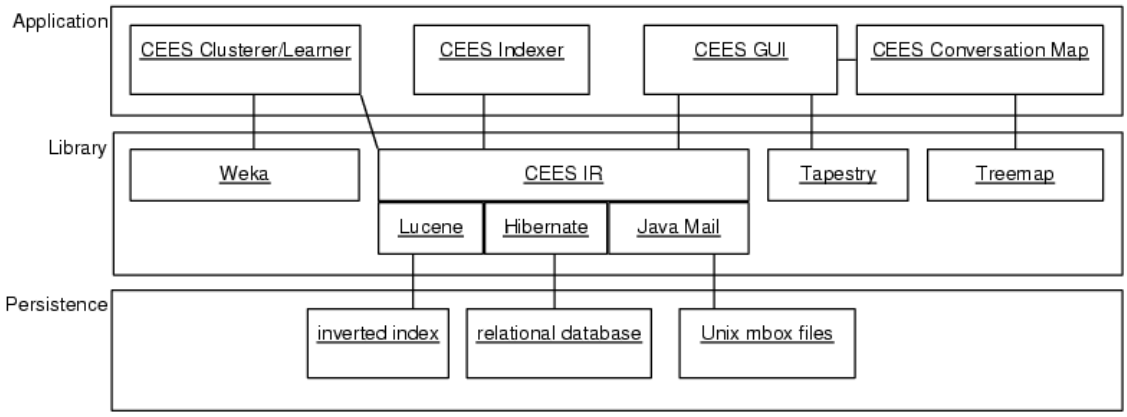

Figure 5.1: CEES Architecture

indexing, and querying services that are shared among the upper-level applications.The upper-level applications, such as CEES Indexer, Clusterer, and the GUI components, all use the CEES IR library layer to access the persistence layer. We will describe some of these components in more details. In order to show the interactions among these components, however, we need first to describe how messages flow through the CEES system.

5.3 Message Processing

Figure 5.2 shows how CEES processes incoming messages. First of all, messages are collected from a newsgroup through NNTP or from a mailing list archive. This process can be done in a manual or automated way by the application that uses CEES. Secondly, if the archive is not in Unix Mbox format already, it needs to be converted to be so. It is worth mentioning that CEES provides an option to implement a custom archive file parser, thus this step depends entirely on the application as well.

CEES then populates the relational database in an object-oriented way. This process is described in Section 5.4 in more details. Essentially, the meta information of a message such as the subject, author, message ids, and references, is stored in the RDBMS. The message body is kept in the Unix Mbox files.

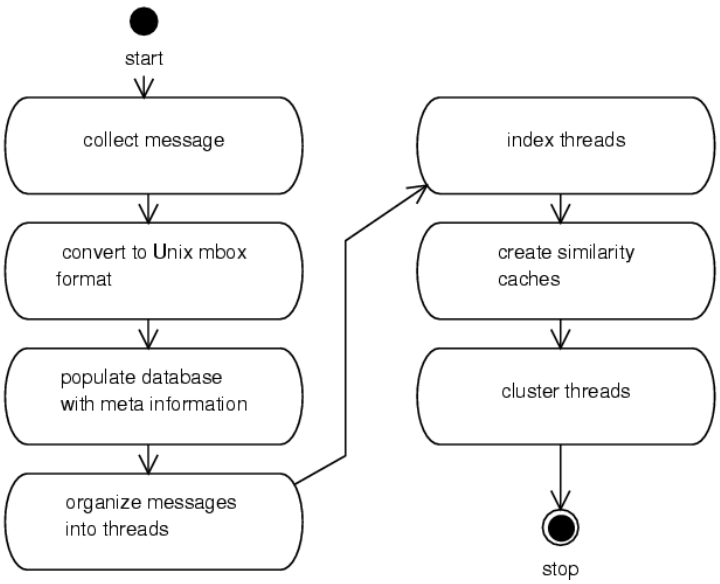

Figure 5.2: CEES Message Processing

The CEES threading library then organizes messages into threads using the meta information stored in the database. The threading library then stores the thread structure in the RDBMS as well. After all thread structures are constructed, CEES' Indexer indexes those threads accordingly.

CEES provides a flexible way to index its documents. By default, the basic indexing unit is a thread, thus CEES indexes all the messages in a thread as a single document. CEES also has an option to index individual fields of a document. Section 3.3.1 depends on this flexible structure in order to calculate the results for different similarity functions.

Finally, the CEES Clusterer calculates the similarity functions and creates the according similarity caches. Similarity caches stores the similarities between any two threads. There can be multiple similarity caches since there are multiple similarity functions, as described in Section 3.3.1. The clustering code in CEES uses those caches during training and testing in order to create the combined similarity functions.

5.4 Domain Objects

Due to the complex indexing requirements, CEES needs a flexible framework to query messages creatively. As seen from the Section 5.3, CEES uses a relational database as the backend to manage the metadata for the email messages. This enables querying and extracting mail metadata using well-known SQL.

Domain objects are objects that map to rows in a relational table. The general domain object patterns are widely used in enterprise software designs [13, 3]. A relational-to-object mapping tool enables the programmer to completely separate business logics from database access code. We use the Hibernate[4] framework to constructs domain objects for CEES. Using Hibernate, domain objects can be written in POJO (Plain Old Java Objects) with very little overhead. Within the Java Virtual Machine (JVM), Hibernate uses Java reflection and a configuration file to create special objects that can access the database.

One advantage of using such framework is the simplicity of coding. Throughout the CEES framework, no SQL statement is written to access the metadata from the database. Mail metadata are retrieved in an object-oriented way using simple Java iterators and factory methods. Using such framework can greatly simplify coding, especially when we perform sophisticated indexing and clustering. The main CEES domain classes are described in Figure 5.3.

5.5 Clustering

CEES clustering architecture also includes an extensible framework for different clustering algorithms. By default, CEES uses an agglomerative clustering algorithm. This allows for more flexibility in the design of the user interface, such as the zooming slider for the Conversation Map. However, CEES' clustering architecture allows for different clustering algorithm such as K-Mean and K-Medoid. In addition, CEES separates the similarity calculation code from the algorithm implementation. This allows CEES to support different implementations of similarity function used for the clustering algorithm.

Figure 5.4 shows an overall design for the clustering classes.

There are five main interfaces in the clustering API. They are:

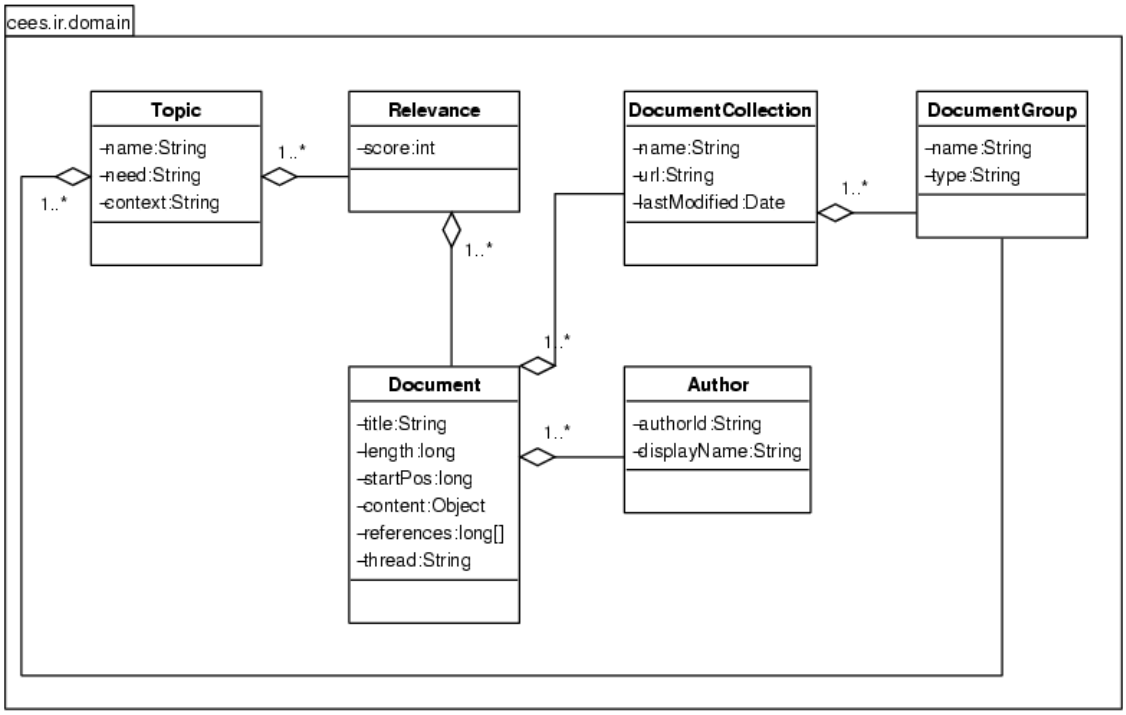

Figure 5.3: Domain Objects

- *ClusterAlgorithm*
- *Cluster*
- *Clusterable*
- *ClusterSimilarityFunction*
- *EntitySimilarityFunction*

The *ClusterAlgorithm* denotes the main algorithm to perform the clustering. Starting from the source entities to cluster (*Clusterable*). *ClusterAlgorithm* uses *EntitySimilarityFunction*, which calculates the similarity between two entities, and optionally the *ClusterSimilarityFunction*, which computes the similarity between two clusters, in order to produce *Cluster.*

The *CachingSimilarityFunction* is used to cache the similarity calculated by its children. It is enabled so that each similarity score between two entities do not need to be recalculated each time.

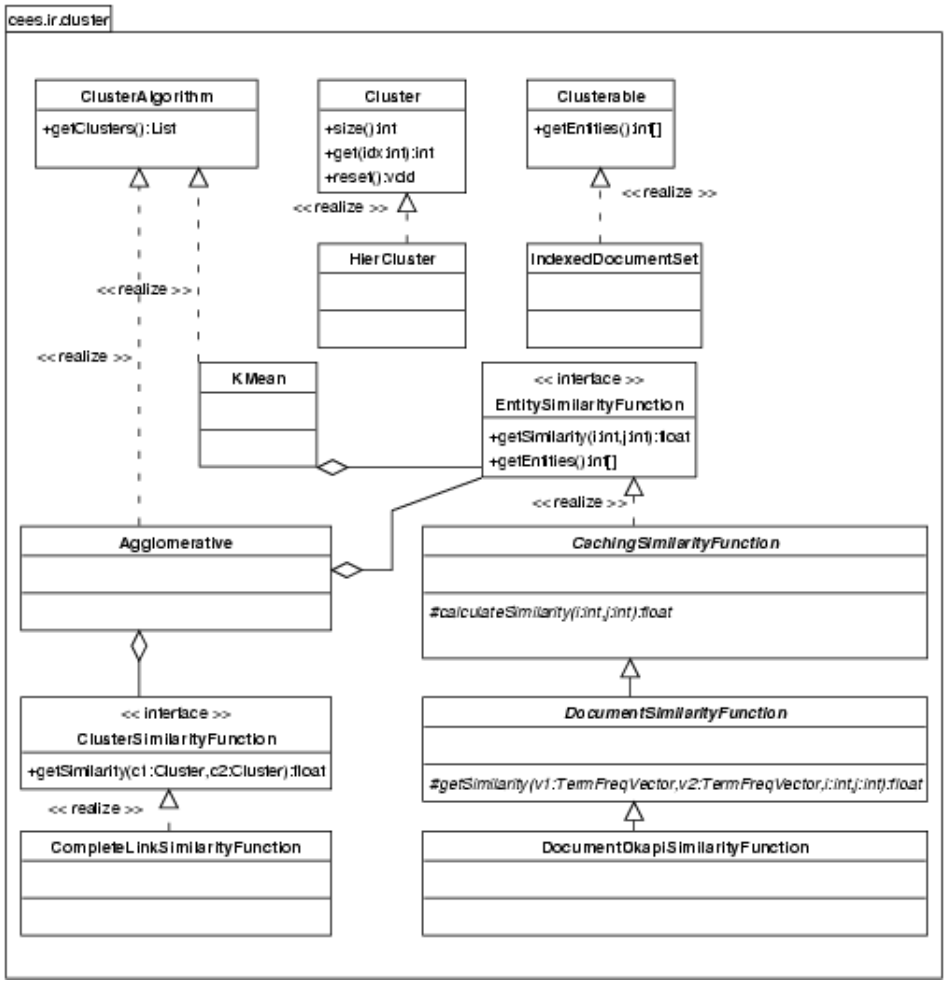

Figure 5.4: Cluster classes

The main similarity function that we use is the *DocumentOkapiSimilarityFunction*, it essentially implements what is described in Section 3.3.2. There is also a *DocumentDateSimilarityFunction* (not shown), which implements the exponential date similarity function. More functions can be added in the future.

In order to find the tight clusters, we use the a complete link algorithm, which is implemented in *CompleteLinkSimilarityFunction*. An average link or single link algorithm can also be implemented with just a few lines of code.

5.6 User Interface

"As far as the customer is concerned, the interface is the product." – Jef Raskin

CEES provides a web-based infrastructure to create feature-rich IR applications. In addition, it provides an innovative way to tag and view threads in a newsgroup or mailing list. CEES' GUI components are based on standard Java servlet technologies and the open source GUI framework Tapestry[29] from the Jakarta project.

The CEES user interface includes:

1. A basic interface that can provide basic search functionality and viewing of threads
2. An advanced tagging interface for generating the golden set for clustering and IR tasks
3. A CM-based searching and browsing interface that can show users an overview of the conversations within a newsgroup. We describe this in Section 5.7.

5.6.1 Basic Thread Information Retrieval

CEES provides basic search capability based on Lucene[14], an open source information retrieval engine. On top of Lucene, CEES builds an abstract layer that integrates Lucene with Tapestry and Hibernate.

Figure 5.5 shows the main search screen. One can perform basic queries such as boolean or phrase queries using the simple Lucene query language.

One can also view the complete thread by either clicking on the [+] symbol or open the thread in a new window by clicking on the link, as shown in Figure 5.6. It is important to note that although the thread structure is constructed internally in the database, it is flattened when displaying in this view.

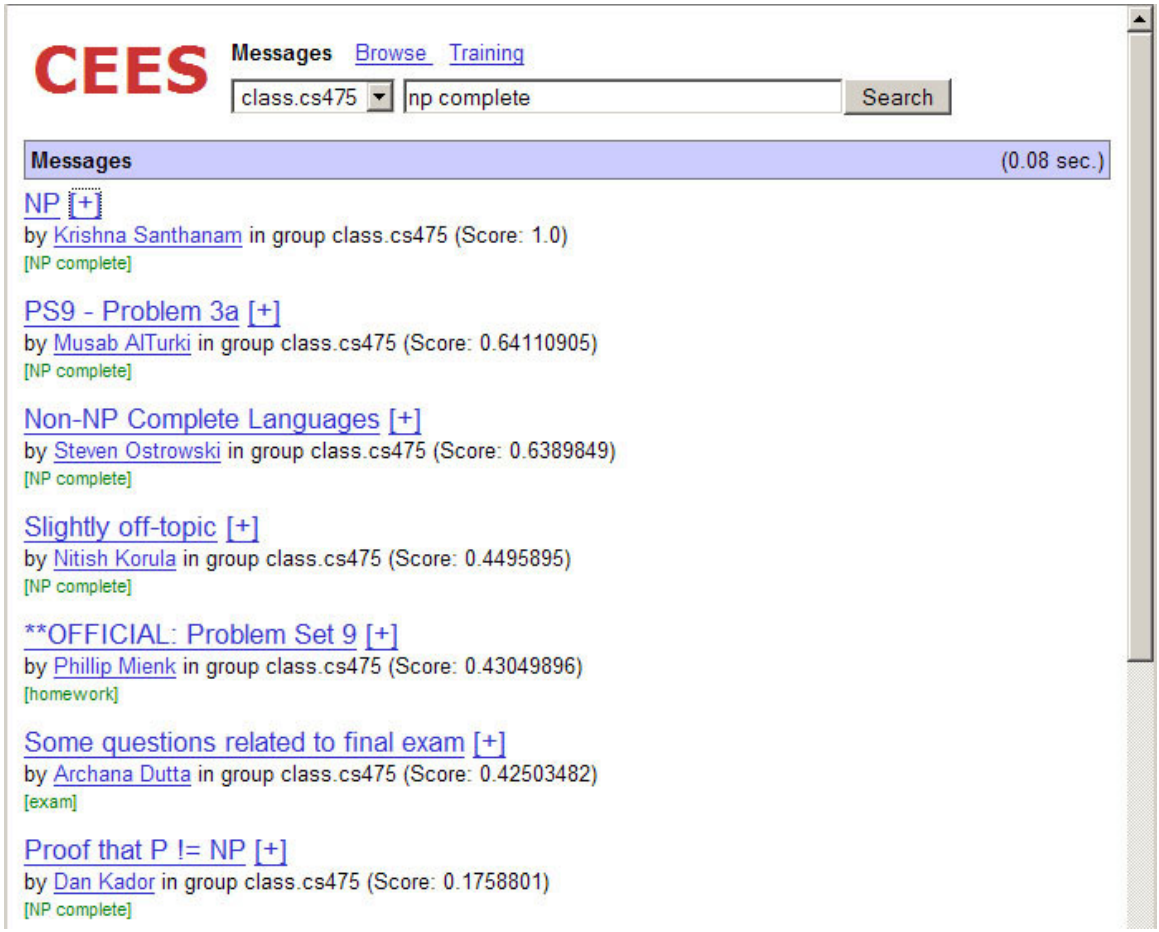

Figure 5.5: Basic Thread Search

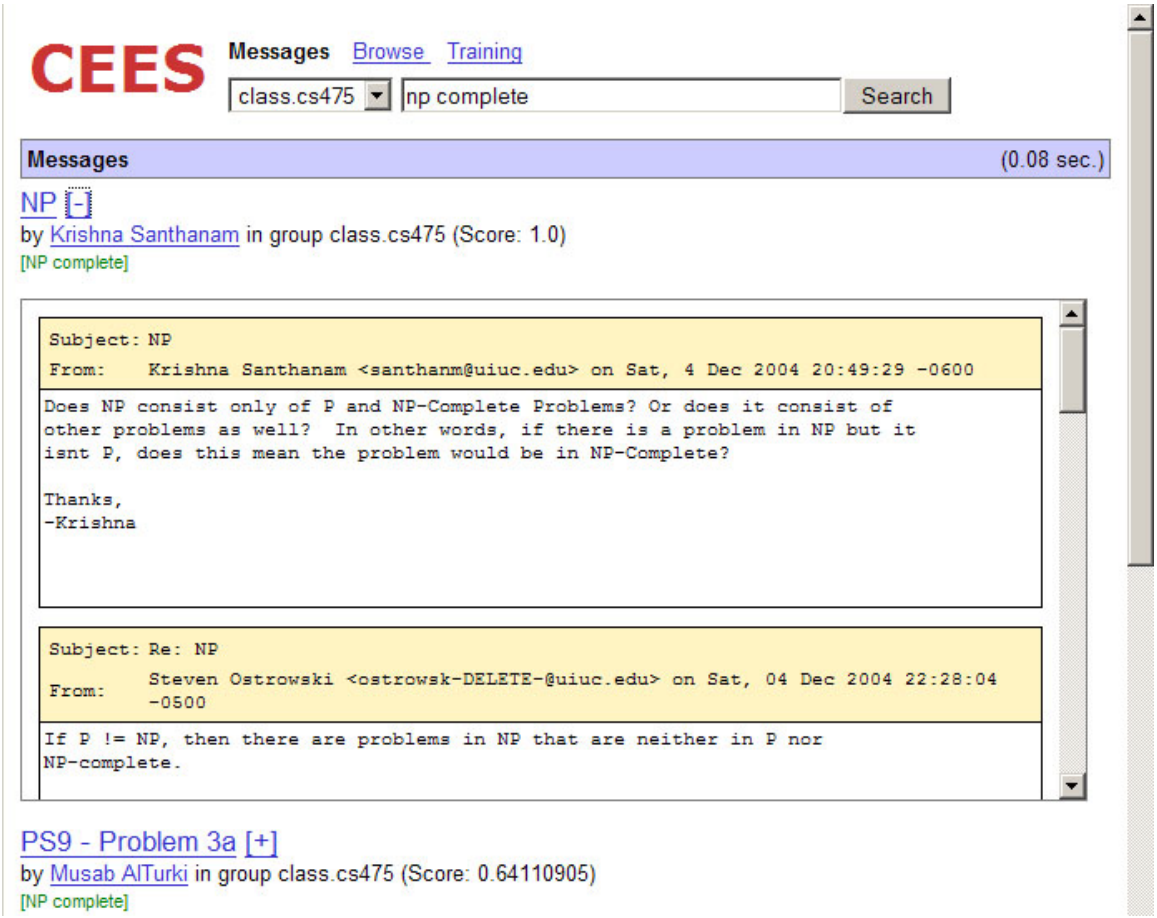

Figure 5.6: Viewing Resulting Thread

This user interface is designed for searching and viewing messages listed as threads. It serves as the building block for the training interface.

5.6.2 Training Interface

CEES also features an unique training interface for grouping threads into groups. This interface can be used to:

1. create the training sets for learning algorithms.

2. construct golden sets for evaluation.

3. build a knowledge base for a newsgroup or mailing list by grouping relevant messages together.

Figure 5.7 shows the main training interface for assigning threads into different subtopics. During our experiment, the taggers have used this interface extensively to group threads into subtopics.

The panel on the left is similar to the basic search interface. A thread can be viewed by clicking on the [+] symbol or clicking on the subject link. The panel on the right indicates the list of subtopics that one can construct by hand.

CEES supports a m-to-n relationship from threads to subtopics. In other words, a subtopic can hold multiple threads and a thread can be associated with multiple subtopics. CEES also supports tagging with different degree of relevance (Figure 5.8). For example, CEES can be used to construct clusters as used in fuzzy clustering algorithms. In our experiment, the tagger is only allowed to assign a thread to one subtopic. The score for the assignment is always the same ("Relevant").

Figure 5.7: CEES Training Interface

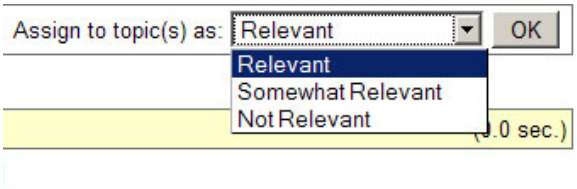

Figure 5.8: Different Levels of Relevance

Figure 5.9 shows the screen when a user click on the "Add" or "Edit" button. Each cluster in CEES can be associated with a Name, Need, or Context. Name is a short subtopic description. The Need attribute stores queries or questions related to this cluster. The context describes the "background" text for the Need.

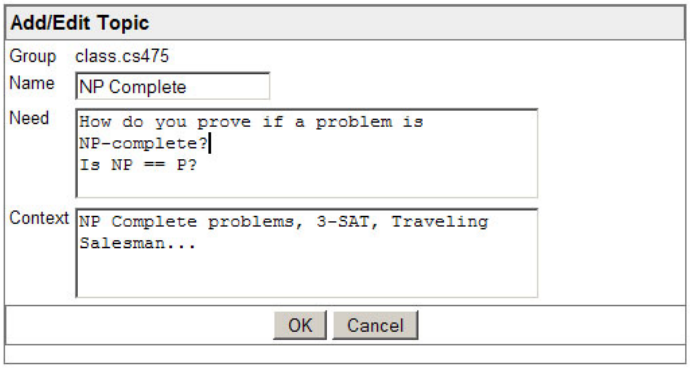

Figure 5.9: Add/Edit a Topic

5.7 Integrating Searching and Browsing with Conversation Map

Although the Conversation Map (CM) is just a visualization mechanism, it can be used in an application with searching and browsing capabilities. In fact, CEES has integrated CM with the underlying web application and information retrieval framework. Figure 4.1 shows a screenshot of the web-based Conversation Map in addition to a search and browse tool.

Due to CM's user interface and portability requirements, the CM module is written as a Java applet. After loading the CM applet on the client side, the applet requests for a cluster definition file from the server. The cluster definition file, written in XML, describes the agglomerative

clustering structure and the metadata for all the threads within the clusters. After the cluster definition file is transferred to the client, the CM applet is responsible for drawing the map on the screen, zooming to the more similar clusters, and make request to the server when the user wants to view the content of a thread.

For the filtering to work, a query is submitted by the CM applet to the CEES GUI server. The server then uses the query to filter out the irrelevant documents, performs agglomerative clustering on the fly, generates a new cluster definition document, and transfers the document to the client. The CM applet then draws from the new definition file in the exact same way.

5.8 Other Possible CEES Applications

As we have seen some of CEES' capabilities, we can think of some possible applications that can utilize its flexible architecture:

- *Author-oriented Search Engine*

  Extending the indexing framework in CEES, one can create a document model based on what the author has posted in the newsgroup or mailing list. One can possibly do interesting language model work based on this author model.

- *Social Network Discovery*

  Since CEES treats each author as an entity, one can create complex queries using CEES domain objects. This may help researchers who are interested in social network mining.

- *Conversation Cluster Summarization*

  Using CEES' message processing and querying code, one can extract the content of each thread and perform summarization on the conversation cluster.

Chapter 6

Conclusions and Future Work

"Believe those who are seeking the truth; doubt those who find it." – André Gide

Due to the rapidly increasing volume of electronic information exchange, the need for managing on-line conversations will nevertheless gain importance in the future. The work we have done during the CEES project has demonstrated that machine learning can be effective in constructing the similarity function between two threads in our clustering task. We have also shown that different newsgroups have different optimal similarity functions due to user preferences and the nature of such group. No single unsupervised similarity function can work best for all groups. We have shown, however, that the similarity functions that we learn from training yield optimal or near optimal results for all three groups in our corpus.

CEES' training and testing framework can be used to combine more sophisticated similarity functions between two threads. For example, future work can leverage natural language processing techniques and define new similarity functions for sentence-level similarity or part-of-speech similarity. Structural similarity in a conversation may also be of interest, since many conversations may follow a particular query and response pattern.

Furthermore, the CEES system may be used to develop and test new clustering techniques that can yield better performance over the existing methods. For example, if we just want to mine for similar messages or questions asked, we may not want to cluster all the messages in

the group. Essentially, we want to focus more on precision and less on recall. One may be able to use this property in order to dramatically improve performance.

CEES also creates a new model for viewing conversations. The Conversation Map is based on the Treemap paradigm and has shown some interesting patterns in our corpus. More efforts should be put in the usability studies in order to test the effectiveness of such model. Providing summarization of the cluster may also make the map more useful.

After all, CEES has established a foundation for further conversation-centric research. Providing a flexible architecture that integrates message management, indexing, parsing, querying, and clustering, CEES has built a framework that can facilitate other research directions. Such research can range from cluster summarization, conversation-centric user interface design, entity extraction, author-oriented information retrieval models, and much more.

Appendix A

Message Threading Algorithm

This threading algorithm is based on the web article[36] posted by Jamie Zawinski. Zawinski was a core developer and the main author for the message threading code used by the original Netscape email client (before version 4.0). According to Zawinski, this algorithm is used in many popular email clients such as Evolution and Balsa. There are libraries that implement this algorithm in Perl and C. CEES has implemented a version of this in Java.

In a RFC 822[9] or 2822[27] compliant email or newsgroup message, the algorithm mainly depends on three headers:

1. *Message-ID*, originally defined in [9], contains an unique identifier for the message. Although the format for the header can be a free-form string, one can be reasonably sure that it contains a unique string that is different than another message's *Message-ID*.

2. If this message is a reply to another, the *In-Reply-To* header, originally defined in [9], specifies the *Message-ID* of the parent message.

3. The *References* header is defined in [9] and later on described in [17] for the USENET messages. By the definition in [17], it should contains the *Message-ID* strings for the message's ancestors, starting from the oldest. However, this header can be somewhat "noisy" since this behavior is not guaranteed. It is possible for two messages to have contradictory headers.

Based on those three headers, one can construct the reply tree structure from a collection of messages. First of all, we defined two objects:

```
class Container
{
  Message message;
  Container parent;
  Container child;
  Container next;
}
class Message
{
  String subject;
  long messageId;
  long[] references;
}
```

The class *Container* would store the tree structure of the thread. The *Message* class contains the subject line, the messageId stored as a Java long type, and a list of references for the message.

As suggested by Zawinski in [35] and modified slightly in CEES, messageId are converted into a 64-bit Java long type by the following algorithm:

```
// Given the Email type representing a valid RFC2822 message
long getLongId(Email message)
{
  String id = message.getHeaderValue(■Message-ID■));
  if (id is not empty)
  {
    return first64BitsToLong(MD5.hash(id));
```

```
    }
    else
    {
      return first64BitsToLong(MD5.hash(message.getContent()));
    }
  }
```

The *MD5* class contains a method named *hash* that would return a 128-bit MD5 hash from a *String*. The function *first64BitsToLong()* takes the first (left-most) 64 bits of the 128-bit hash and convert that to the Java long type. If the message does not have a *Message-ID* header, we simply hash the message itself and transform that into a long type.

This optimization enables us to make comparison of different message types much more easily. We can simply compare the long value instead of storing the *Message-ID* that can be more than 25 characters long.

There is an *idTable* hash table that maps a *messageID* to the appropriate *Container*. Given such table, Zawinski suggests the following algorithm for message threading:

1. For each of the message in the collection that you want to thread:
   (a) If *idTable* contains a container with no message (the container is empty) with the message id, we put the message in the given container, otherwise, we create a new container holding the message and put the container in *idTable*.
   (b) For each element in the message's *References* field:
      i. If the element's *messageId* is in *idTable*, we add the message in the container.
      ii. Otherwise, we make a new container and put it in *idTable*.
   (c) We then link the references field in the order implied in the *References:*
      i. If the container is already linked, then do not change.
      ii. If adding a link will create a loop, do not add the link.
   (d) Sets the parent of this message to be the last element in the *References*.

2. Gather the elements in *idTable* that has no parent.

3. Prune the empty containers. Recursively walk down all the containers:

   (a) If a container has no message and no children, we discard it.

   (b) If a container does not have a message but has children, we remove the current container and promote the child to take its place. One should not promote the child to the root set, unless there is only one child.

4. Group root set by subject. This part is considered somewhat optional. CEES has implemented this for completeness. Threads are usually grouped together if they share the same subject, even if they have not specified the *References* field.

   (a) Create a subject table *subjectTable* that maps subject strings to the container objects.

   (b) For all the top level containers:

      i. Find the subject of the subtree

         A. If container contains a message, the subject is the subject of the message

         B. Otherwise, use the subject of the child's message as the subject

         C. Strip the "Re:", "RE:", etc. from the subject

         D. If the subject is now an empty string, then we skip this container

         E. Add this container to *subjectTable*:

         F. If there is no entry in *subjectTable* with the same subject

         G. If this one is an empty container and the old one in *subjectTable* is not, then we use replace the entry in the table with the empty one instead.

         H. If the container has a "Re:" version of this subject in *subjectTable*, and this container does not, replace the entry in *subjectTable* with this one.

   (c) After we populate the *subjectTable*, we need to gather the same subjects together and put them under one thread. Essentially, for each container in the root set:

      i. Find the subject of this container

ii. If *subjectTable* does not have this subject as a key, or if the lookup is the current container, we continue.

iii. Otherwise, we want to group together this container and the one in the table. Here are the different cases:

A. If both container are empty, append one's children to the other and remove the empty container.

B. If one is empty and the other is not, then make the non-empty one the child of the empty one, with its siblings as the previous children of the empty container.

C. If this container's subject begins with a "Re:", and the one in the table does not, make this the child of the other.

D. If this container's subject does not begin with a "Re:", and the one in the table does, then make this the parent of the other.

E. Otherwise, make an empty container and make both container the child of the empty one.

5. You can now sort the top-level containers based on the date, sender, subject, or others. In CEES, they are sorted by date. You are done!

Appendix B

Efficiently Returning Different Numbers of Clusters

For a Conversation Map, in order to quickly return different number of clusters, we want to maintain the complete agglomerative tree hierarchy. Each intermediate agglomerative clustering node is then represented by a tree node of type *HierCluster*. *HierCluster* can be defined as:

```
class HierCluster
{
  int threadId;
  float intraSimilarity;
  HierCluster child;
  HierCluster next;
}
```

In particular, *child* references the first child of this tree node, and *next* points to the next sibling cluster in the tree. For example, there are three clusters, {D, E}, {C}, and {A, B}, in Figure B.1.

The *intraSimilarity* field is only set in a non-leaf node. This variable represents the ICS for all non-leaf nodes under this tree node. In Figure B.1, Cluster 1, 2, and 3 are non-leaf nodes and contains a non-zero *intraSimilarity* field. The field is calculated as in Equation 3.4.

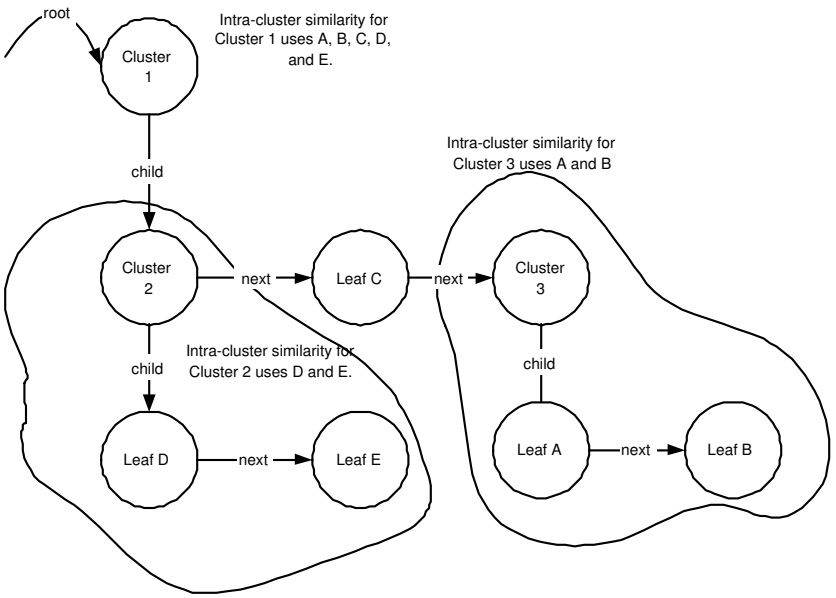

Figure B.1: An example cluster

We can sort the all the ICS (Intra-Cluster Similarity) of all the non-leaf nodes of the tree into an list $(x_0, x_1, x_2, ..., x_{n-1})$ where $x_{i+1} > x_i$ and n is the number of non-leaf nodes in the tree. We then would like to let user define a threshold $\alpha = [0..1]$ that determines the "cohesiveness" of the clusters. Given α, we would pick x_i as the ICS threshold by setting i as:

$$i \quad = \quad \begin{cases} n-1 & \text{if } \alpha = 1 \\ \lfloor n \times \alpha \rfloor & \text{otherwise} \end{cases} \tag{B.1}$$

where$\lfloor y \rfloor$ stands for the flooring function.

In order to the use x_i as a cut-off measure to separate our cluster, we need to walk down the *HierCluster* tree recursively and collect the clusters where the *intraSimilarity* is larger than x_i. We can perform this cluster splitting quickly if we store the ICS for each non-leaf node of the tree, and we run the following recursive function:

```
collectClusters(List finalClusters, HierCluster clusterN-
ode, float xi)
```

```
{
  if (clusterNode is a leaf or clusterNode.intraSimilarity >= x_i)
    add clusterNode to finalClusters;
    return;
  else
    for each children c of clusterNode
      collectClusters(finalClusters, c, x_i)
}
```

This essentially "pulls up" the clusters in the tree to the highest level. The recursion would stop when it reaches a node where the *intraSimilarity* is more than the threshold x_i or when it reaches a leaf. In such case, we just add the leaf to the cluster.

For example in Figure B.1, we can set *intraSimilarity* for Cluster 1, 2, 3 to 0.001, 0.2, and 0.1 accordingly. One should notice that in most cases, the parent ICS is going to be less than the children's, since grouping more nodes in the cluster is most likely going to make the elements in the cluster less alike.

Therefore, if x_i is set to be 0.1 , then the final cluster would become $\{(D, E), (A, B), (C)\}$. If we set $x_i = 0.2$, then the cluster would become $\{(D, E), (A), (B), (C)\}$. In the second case, Leaf A and B are broken into two separate clusters, since the Intra-Cluster Similarity for Cluster 3 is less than 0.2.

The slider for the Conversation Map implements this thresholding scheme in order to zoom to the more relevant clusters.

Bibliography

[1] Carrot 2. http://www.cs.put.poznan.pl/dweiss/carrot/.

[2] Visvismo. http://visvismo.com.

[3] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 2003.

[4] Christian Bauer and Gavin King. *Hibernate In Action*. Manning Publications Co., 2004.

[5] Ruth Bergman, Martin Griss, and Carl Staelin. A personal email assistant. 2002.

[6] Vitor R. Carvalho and William W. Cohen. Learning to extract signature and reply lines from email. *Conference on Email and Anti-Spam (CEAS 2004)*, 2004.

[7] William Cohen. Learning rules that classify e-mail. In *Advances in Inductive Logic Programming*. 1996.

[8] William Cohen, Vitor R. Carvalho, and Tom Mitchell. Learning to classify email into speech acts. *HLT/EMNLP*, 2004.

[9] D. Crocker. Standard for the format of ARPA Internet text messages. RFC 822 (Standard), August 1982. Obsoleted by RFC 2822, updated by RFCs 1123, 1138, 1148, 1327, 2156.

[10] Kushal Dave, Steve Lawrence, and David M. Pennock. Mining the peanut gallery: opinion extraction and semantic classification of product reviews. In *Proceedings of the twelfth international conference on World Wide Web*, pages 519–528. ACM Press, 2003.

[11] Yanlei Diao, Hongjun Lu, , and Dekai Wu. A comparative study of classification based personal e-mail filtering. 2000.

[12] Peter Eklund and Richard Cole. Structured Ontology and Information Retrieval for Email Search and Discovery. 2002.

[13] Martin Fowler. *Patterns of Enterprise Application Aarchitecture*. 2003.

[14] Otis Gospodnetic and Erik Hatcher. *Lucene in Action*. Manning Publications Co., 2005.

[15] Ji He, Ah-Hwee Tan, Chew-Lim Tan, and Sam-Yuan Sung. On quantitative evaluation of clustering systems. In Weili Wu and Hui Xiong, editors, *Information Retrieval and Clustering*. Kluwer Academic Publishers, 2002. In press.

[16] Djoerd Hiemstra. Term-Specific Smoothing for the Language Modeling Approach to Information Retrieval: The Importance of a Query Term. 2002.

[17] M.R. Horton and R. Adams. Standard for interchange of USENET messages. RFC 1036, December 1987.

[18] Minqing Hu and Bing Liu. Mining and summarizing customer reviews. In *Proceedings of the 2004 ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 168–177. ACM Press, 2004.

[19] Rong Jin, Alex G. Hauptmann, and ChengXiang Zhai. Title Language Model for Information Retrieval. 2002.

[20] Bryan Klimt and Yiming Yang. The enron corpus: A new dataset for email classification research. 2004.

[21] John Lafferty and ChengXiang Zhai. Document Language Models, Query Models, and Risk Minization for Information Retrieval. 2001.

[22] Ken Lang. Newsweeder: Learning to filter netnews. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 331–339, 1995.

[23] S. Le Cessie and J.C. Van Houwelingen. Ridge estimators in logistic regression. 41(1):191–201, 1992.

[24] Tom Mainelli. Newsgroups get a new life. http://www.pcworld.com/news/article/0,aid,102081,00.asp. 2002.

[25] Paul Ogilvie and Jamie Callan. Combining document representations for known-item search. In *SIGIR '03: Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*, pages 143–150, New York, NY, USA, 2003. ACM Press.

[26] Terry Payne. *Learning Email Filtering Rules with Magi, A Mail Agent Interface.* PhD thesis, Dept. of Computer Science, University of Aberdeen, 1994.

[27] P. Resnick. Internet Message Format. RFC 2822 (Proposed Standard), April 2001.

[28] S E Robertson and S Walker. Okapi/keenbow at trec-8.

[29] Howard M. Lewis Ship. *Tapestry In Action.* Manning Publications Co., 2004.

[30] Ben Shneiderman. Tree visualization with tree-maps: 2-d space-filling approach. *ACM Trans. Graph.*, 11(1):92–99, 1992.

[31] Amit Singhal. Modern information retrieval: A brief overview. 2001.

[32] Marc A. Smith and Andrew T. Fiore. Visualization components for persistent conversations. In *CHI '01: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 136–143, New York, NY, USA, 2001. ACM Press.

[33] Fei Song and W. Bruce Croft. A general language model for information retrieval. In *CIKM '99: Proceedings of the eighth international conference on Information and knowledge management*, pages 316–321, New York, NY, USA, 1999. ACM Press.

[34] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations.* Morgan Kaufmann, 1999.

[35] Jamie Zawinski. Mbox summarization. http://www.jwz.org/doc/mailsum.html. 2002.

[36] Jamie Zawinski. Message threading. http://www.jwz.org/doc/threading.html. 2002.

[37] Chengxiang Zhai and John Lafferty. A Study of Smoothing Methods for Language Models Applied to Ad Hoc Information Retrieval. In *Research and Development in Information Retrieval*, pages 334–342, 2001.