SANDIA REPORT

SAND2004-4905
Unlimited Release
Printed October 2004

Analysis of Multichannel Internet Communication

Eric R. Robinson and Lisa A. Torrey
University of Wisconsin-Madison
Madison, WI 53706

Jeremy L. Newland and Andrew K. Theuninck
University of Minnesota-Duluth
Duluth, MN 55812

Richard F. Maclin and Charles E. Nove
Systems Technology Department
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1397

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94AL85000.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865)576-8401
Facsimile: (865)576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800)553-6847
Facsimile: (703)605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2004-4905
Unlimited Release
Printed October 2004

Analysis of Multichannel Internet Communication

Eric R. Robinson and Lisa A. Torrey
University of Wisconsin-Madison
Madison, WI 53706

Jeremy L. Newland and Andrew K. Theuninck
University of Minnesota-Duluth
Duluth, MN 55812

Richard F. Maclin and Charles E. Nove
Systems Technology Department
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1397

Abstract

A novel method employing machine-based learning to identify messages related to other messages is described and evaluated. This technique may enable an analyst to identify and correlate a small number of related messages from a large sample of individual messages. The classic machine learning techniques of decision trees and naïve Bayes classification are seeded with few (or no) messages of interest and "learn" to identify other related messages. The performance of this approach and these specific learning techniques are evaluated and generalized.

Contents

1. Introduction 7
2. Message Simulation....7
3. Communication Capture 10
Overview 11
The Link Layer (Sniffer) 11
The Network Layer (TCP Assembler) 12
The Transport Layer (Stream Analyzer) 12
The Application Layer (Stream Parser) 13
4. Graphical Interface....13
Machine Learning in the Analyzer 15
5. Decision Tree Learning....16
6. Naïve Bayes Learning....22
7. Future Work 25
8. Conclusions 26
9. References....27

List of Figures

Figure 1. Virtual network structure....8
Figure 2. Client communication. 8
Figure 3. Communication capture software. 11
Figure 4. Using the Conversation Analyzer....15
Figure 5. Decision tree example....18
Figure 6. Decision tree cross-validation. 20
Figure 7. Decision tree performance. 22
Figure 8. Naïve Bayes performance with seeds. 24
Figure 9. Naïve Bayes performance without seeds. 25

List of Tables

Table 1. Client-to-server messages....9
Table 2. Server-to-client messages. 9
Table 3. Identifying elements of IM messages. 14
Table 4. Identifying elements of chat messages....14

1. Introduction

The goal of LDRD 04-0442 was to help an analyst sort through large amounts of captured message data to find a few interesting conversations. Building off previous work from 2003, we completed a framework to simulate a network of communicating users and capture their communications. We then went on to design a convenient graphical interface to allow an analyst to examine captured conversations and applied several machine learning methods to the problem of finding conversations of interest.

This report describes the software we produced and the machine learning techniques we applied, and presents results of the tests we performed to evaluate those techniques. Section 2 explains the simulation, Section 3 the communication capturing program, Section 4 the graphical interface, and Sections 5-7 the machine learning techniques, their experimental results, and suggestions for future work. Section 8 closes with our conclusions.

2. Message Simulation

We simulated a network of communicating users in order to show that we could reliably capture conversation data and reconstruct conversations. We also used the simulation to generate realistic message data to use for analysis. Each instance of the simulation program imitates the behavior of an AOL Instant Messenger client, signing onto the AIM service, and exchanging messages with other copies of the program. We chose to work with AIM because it is one of the most popular messaging programs available. Client communications are drawn from a database of topic-oriented messages.

Each client program is intended to run on a different computer or virtual machine. Using a program called VMware, we ran 11 virtual machines on each of two computers and put a client on each one, creating a network of 22 clients. Using virtual machines instead of physical ones was just a convenience that allowed us to use a small number of machines to simulate a larger network. The network's layout is summarized in Figure 1. Note that by capturing packets where we do, we will not see direct communications between machines like 16 and 17 who are on the same sub-network; however, since all AIM messages go out to the server and then come back to the recipient, we will capture any AIM messages between conversants on those two machines.

Client programs get information about how to behave from configuration files. These files determine when the program signs on to AIM, when it signs off, who it talks to in between, and where it gets the messages it sends. Each of these properties has probabilistic parameters, so that each time a client runs it will behave slightly differently but within a characteristic range.

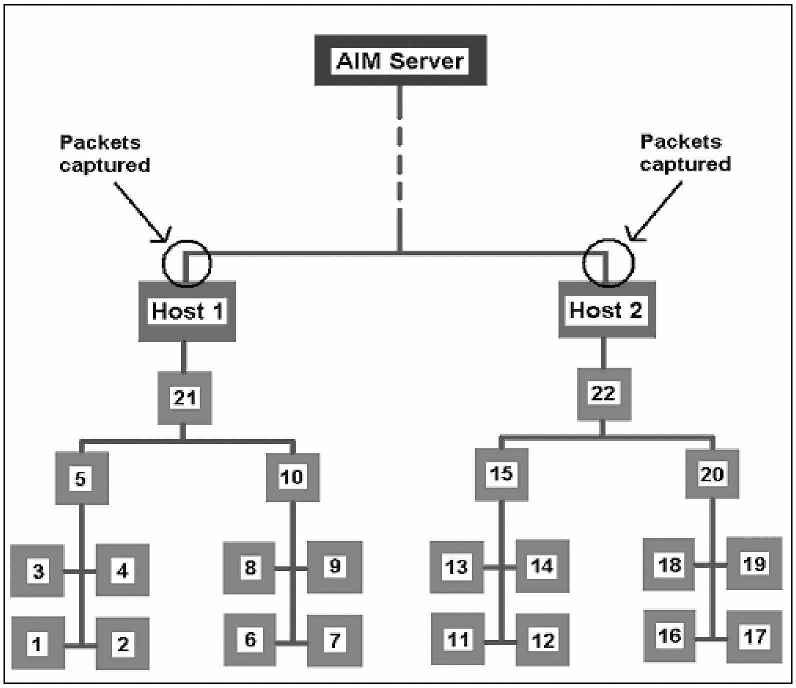


Figure 1. Virtual network structure.

We configured our 22 clients so that they would each talk to several others in two different formats. One is the instant message, in which two people send messages through the AIM server to each other. The other is the group chat, in which any number of people send messages through the server to the whole group. Figure 2 illustrates the general scheme.

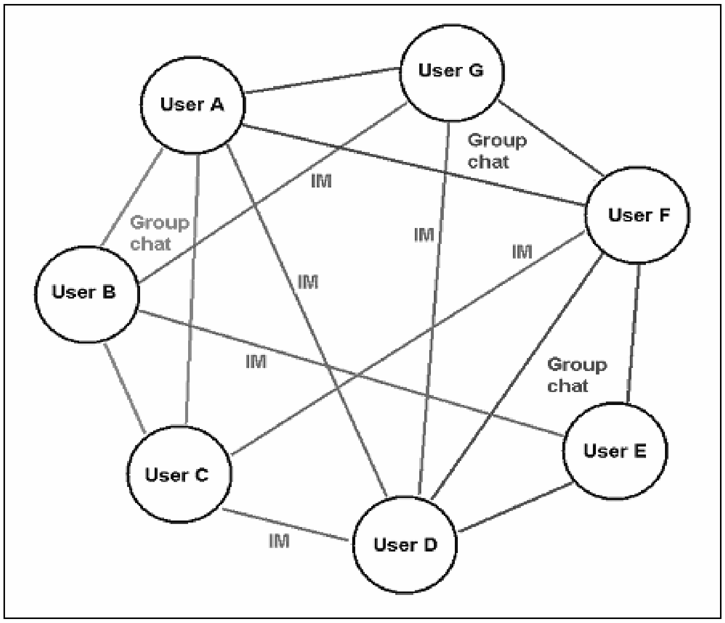


Figure 2. Client communication.

In order to produce somewhat realistic conversations, we drew messages from a database that was organized into different topics. Previous work on this project included a program to pull posts from Usenet newsgroups, and we used that program to acquire 50 conversation topics with hundreds of messages each.

For each instant message or chat exchange, the client programs decide on a topic and take messages for that conversation from a single section of the database. The messages are cut short to obey the length limits of the TOC protocol, consecutive messages in an exchange may not be directly related to one another, and messages sometimes do not stick to the supposed topic. However, they are valid natural-language communications that connect loosely, and we can almost always identify the source of a conversation just by glancing through it.

The client programs communicate with the AIM server using one of the two existing AIM protocols: Oscar and TOC. Oscar is the official protocol, used by the AIM software, but it is not publicly available and is only used in non-AIM products through back-engineering. TOC is an unofficial protocol, but it is publicly available, and therefore we chose to use it.

Most of the client program is original Java code. To use the TOC protocol, we imported code from the open-source project "Java AIMBot." The project was designed to allow a program to answer instant messages, and with a few modifications and additions we adapted it to our purpose. The AIMBot code handles all the messages that a client sends to or receives from the AIM server. Tables 1 and 2 show some of these messages.

Table 1. Client-to-server messages.

toc_send_im <Destination User> <Message>
toc_chat_join <Exchange> <Chat Room Name>
toc_chat_invite <Chat Room ID> <Invite Msg> <buddy1> [<buddy2> [<buddy3> [...]]]
toc_chat_leave <Chat Room ID>

Table 2. Server-to-client messages.

IM_IN:<Source User>:<Auto Response T/F?>:<Message>
UPDATE_BUDDY:<Buddy User>:<Online? T/F>:<Evil Amount>:<Signon Time>:<IdleTime>:<UC>
CHAT_INVITE:<Chat Room Name>:<Chat Room Id>:<Invite Sender>:<Message>
CHAT_JOIN:<Chat Room Id>:<Chat Room Name>
CHAT_IN:<Chat Room Id>:<Source User>:<Whisper? T/F>:<Message>
CHAT_UPDATE_BUDDY:<Chat Room Id>:<Inside? T/F>:<User 1>:<User 2>...
ERROR:<Error Code>:Var args

The clients log all the messages they send. We used these logs to verify our message capturing process, which is described in the following section.

3. Communication Capture

Communication capture software was written in a previous stage of this project. We restructured and rewrote this software in order to achieve our goal of 100% accurate conversation reconstruction from captured network traffic. The original version of the "sniffer" processed most network traffic under the assumption of a one-to-one correspondence between packets and messages. While this was sufficient for the majority of the messages in a conversation, we noticed some discrepancies between the transcripts recorded in the client logs and the transcripts reconstructed from the captured network traffic. There were four major sources for these discrepancies:

1. Packets may contain multiple complete and/or partial messages.
2. Messages may span multiple packets.
3. The "protocol flag" (i.e., the '*' indicating AIM messages) may not always be the first character in the application layer of a packet.
4. Duplicate copies of a single message/packet may be sent and/or received.

Problems one, two and three generally lead to un-recovered messages. Additionally, problem two can also lead to partially recovered messages while problem four leads to duplicate messages.

Problem one, that packets may contain multiple complete and/or partial messages, we were able to resolve to a large extent by extending the original sniffer. Problems two, three and four arose at approximately the same time. We initially considered adapting the code in the original sniffer that partially handles packet-spanning messages for the Yahoo Messenger protocol to work with the TOC protocol. It soon became clear, however, that this was not a desirable approach. The issue underlying all four problems was TCP/IP fragmentation. As such, extending the Yahoo Messenger code was clearly not only inappropriate (why handle the general issue of TCP/IP fragmentation in a chat-protocol specific manner?), but also would not provide a solution for the "protocol flag" problem or the duplicate message problem.

We chose instead to redesign and rewrite the communication capture software. This had the dual benefits of addressing all four problems inherent in the design/implementation of the original sniffer, while also isolating and reducing the amount of protocol-specific processing by doing conversation reconstruction in a more general and robust framework. The final result was a suite of programs illustrated in Figure 3.

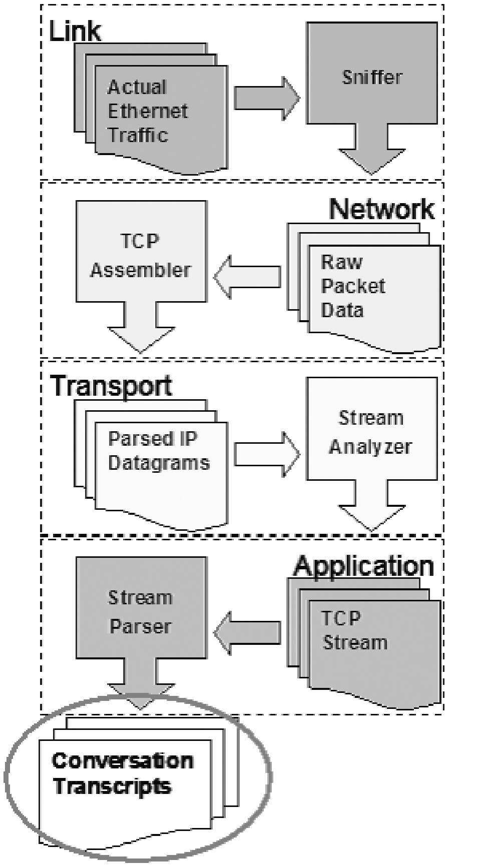


Figure 3. Communication capture software.

Overview

There are four programs corresponding to the four layers of encapsulation (Link, Network, Transport, Application) in the exchange of IM/Chat messages. The ultimate goal of the four programs is the accurate reconstruction of the IM/Chat conversations contained in the observed network traffic. We have already seen that attempting to reconstruct conversations from packet data alone, without consideration for which communication stream that packet is a part of and where in that stream the packet belongs, is bound to fail. Briefly, the new "Sniffer" is responsible for filtering out and recording TCP/IP packets. The "TCP Assembler" is responsible for extracting source/destination information and IP fragment reassembly. The "Stream Analyzer" is responsible for reconstructing (including duplicate detection and out-of-order packet receipt) the TCP stream data exchanged (i.e., the socket of communication). The "Stream Parser" is responsible for recognizing whether the information exchanged over a socket conforms to a known chat/IM protocol and, if so, extracting the messages from that communication stream and producing the accurately reconstructed conversation transcripts.

The Link Layer (Sniffer)

Packets are captured from the primary network device using the winpcap library. As with the original sniffer, non-IP (v4) and non-TCP packets are ignored. Other than this

TCP/IP protocol filtering, the new sniffer does not do any byte-level analysis or processing of the packets. The raw IP datagrams (i.e., the packets stripped of the Link layer) are passed to a binary dump file for processing by the "TCP Assembler."

The Network Layer (TCP Assembler)

The raw IP datagrams are read sequentially from the binary dump file generated by the Sniffer. The IP headers are then parsed for their relevant fragmentation, source, and destination information. Due to the nature of the IP protocol, once parsed (and reassembled, if necessary), the only information required for future use for our purposes (in addition to the TCP/Application payload) is the source and destination IP addresses. Thus, immediately after parsing, the addressing information and the TCP/Application payload are stored in a MySQL database and the datagram is forgotten.

The IP fragment reassembly is not currently implemented in the TCP Assembler. No fragmented IP datagrams arose in any of our simulations, so reconstructed conversation accuracy was not compromised. In the absence of any actual fragmented IP datagrams, testing of fragment reassembly code could be done with specially (manually) generated IP packet data. We chose not to pursue this due to time constraints.

The Transport Layer (Stream Analyzer)

The Stream Analyzer is the most complicated piece of the conversation reconstruction puzzle. The Stream Analyzer examines the TCP/Application payload from each IP datagram stored in the database by the TCP Assembler. It then parses the TCP header to extract the information necessary to reconstruct the data exchanged over the socket. With this information, it attempts to simulate the processing done at each end of the socket that led to the observed exchange of packets. By utilizing the information contained in the TCP header (e.g., SYN, FIN, ACK control information, Acknowledgement Number and Sequence Number information) the Stream Analyzer is able to do a byte-level reconstruction of the data exchanged over the socket in question, reconstruction that is accurate with respect to both the content exchanged and the order in which it was exchanged. For each reconstructed socket, the Stream Analyzer stores the stream of application data exchanged in a series of BLOBs (Binary Large OBjects, which are database structures for binary data) in sequence number order in the MySQL database. The Stream Analyzer is written to appropriately handle partial socket observation, meaning that extraneous traffic detected need not be purged from the database for analysis and will not lead to any erroneous results or program crashes.

It would probably be worth the effort to store the Application data stream in one single BLOB in one single socket tuple rather than in a series of BLOBs indexed on sequence number. This was not pursued due to time constraints. Also, the issue of partial acknowledgements of packets is still not being handled 100% correctly. Based on the traffic generated by our simulations, it seems to be an extremely rare event (only occurred once and did not affect any of our transcript analysis) but should still be addressed.

As with the original sniffer, the Stream Analyzer will ignore Sockets on ports 80 or 8080. We ignored these ports because they are used almost exclusively for HTTP traffic and increased running time for packet processing without contributing to our research goals. These ports need not be ignored, and including them will not cause any problems for the Stream Parser.

The Application Layer (Stream Parser)

The Stream Parser is the last stage in the processing and is responsible for producing the final conversation transcripts. The Stream Parser examines each reconstructed socket to determine if the communication exchange corresponds to a known IM/Chat protocol. If so, the Stream Parser works its way through the stream, parsing and recording messages as encountered until it reaches the end of the stream. Since duplicate detection, reordering, etc. are handled by the Stream Analyzer, the parsing itself is quite straightforward — simply a matter of following the protocol in question. Also, since the Parser deals with a stream of data and not an atomic unit like a packet, the "protocol flag" issue is completely sidestepped. Upon completion of this step, the MySQL database contains the reconstructed conversations (100% accuracy on our simulations this summer).

Additional protocols can be added to the Stream Parser by adding appropriate methods and calls to the StreamParser class. Stream Parsing could be greatly simplified by implementing the single BLOB per socket optimization, as described above. Also, conversation transcripts come in identical pairs, one transcript from the server perspective and one transcript from the client perspective.

4. Graphical Interface

To examine captured conversations conveniently, we developed a GUI, or graphical user interface, called the Conversation Analyzer. It is written in Java, and it essentially serves as a visual front-end to the database of captured data. Once the packets have been stored in the table format specified by the Stream Parser, we can use the Analyzer to group messages into conversations, display the conversations, and manipulate selections of conversations. Note that conversations are not discarded by the tool, but are merely re-ordered in how they are presented to the analyst.

To prepare a set of data for use, an Analyzer menu item can be selected that will group together messages in the database that belong to the same exchange. The process is necessarily specific to the TOC protocol. It starts by putting together all the messages that are from one person's "side" of a conversation — everything that person sent to or received from a single instant message or chat channel. As shown in Tables 3 and 4, the bold elements distinguish sides from each other; these elements must match in all messages belonging to one side of one conversation. A client's outgoing chat messages are reflected back as incoming chat messages, so outgoing chats are ignored.

Table 3. Identifying elements of IM messages.

Outgoing:	*<source IP address>* <server IP address> *<socket ID>* *<recipient>* <message>
Incoming:	<server IP address> *<destination IP address>* *<socket ID>* *<sender>* <message>

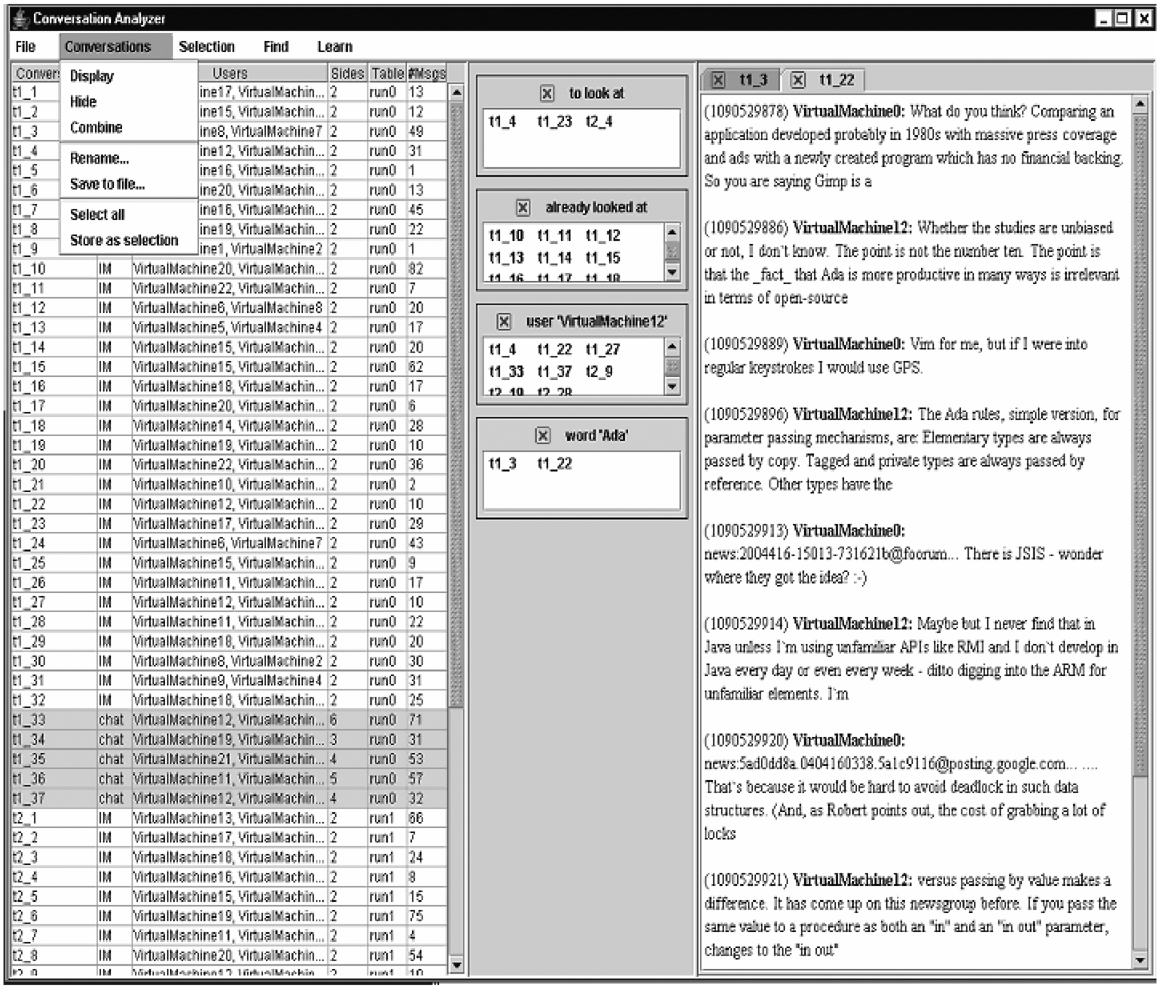
Table 4. Identifying elements of chat messages.

Incoming:	*<chat room ID>* <server IP address> *<destination IP address>* *<socket ID>* <sender> <message>

Depending on where the data were sniffed, one or more sides of each conversation may have been recorded. With the TOC protocol, the only way to determine whether two sides belong together is to look at the messages they have in common. The Analyzer program attempts to put sides together using an arbitrary threshold: if two sides have at least five consecutive messages in common, they are combined. This has only a very small chance of combining sides inappropriately. It occasionally leaves sides separate when it could combine them, but then these are simply treated as separate conversations.

Once conversations have been assembled, the Analyzer will display a list of the conversations. We can then select from menu options to manipulate them: display the text, remove some from the display, combine sides manually, give them labels, print them to a file, or put selected ones into a temporary group or "selection." We can make a selection out of all conversations that involve a certain user or contain a certain word, and we can take the union and intersection of selections. These basic functions allow us to examine and organize conversations easily. Figure 4 shows the Analyzer in the process of doing several of these things.

While this interface is useful for basic functions, it does not solve the motivating problem of our research, which is that there will be too many conversations to search exhaustively. Note also that while we can use simple word searches, not all conversations of interest may use the words we think to search for. It may be difficult to characterize exactly what makes a conversation interesting. We might find a few interesting ones and want to find others similar to it, or we might not have anything to start from at all and want to sift through the list in a more efficient way than top-to-bottom. This is where machine learning comes in: the Analyzer has the capability to learn our interests and bring to our attention, more quickly than we could alone, conversations that are likely to be relevant to us.



Conver		Users	Sides	Table	#Msgs
t1_1		ine17, VirtualMachin...	2	run0	13
t1_2		ine15, VirtualMachin...	2	run0	12
t1_3		ine8, VirtualMachine7	2	run0	49
t1_4		ine12, VirtualMachin...	2	run0	31
t1_5		ine16, VirtualMachin...	2	run0	1
t1_6		ine20, VirtualMachin...	2	run0	13
t1_7		ine16, VirtualMachin...	2	run0	45
t1_8		ine19, VirtualMachin...	2	run0	22
t1_9		ine1, VirtualMachine2	2	run0	1
t1_10	IM	VirtualMachine20, VirtualMachin...	2	run0	82
t1_11	IM	VirtualMachine22, VirtualMachin...	2	run0	7
t1_12	IM	VirtualMachine6, VirtualMachine8	2	run0	20
t1_13	IM	VirtualMachine5, VirtualMachine4	2	run0	17
t1_14	IM	VirtualMachine15, VirtualMachin...	2	run0	20
t1_15	IM	VirtualMachine15, VirtualMachin...	2	run0	62
t1_16	IM	VirtualMachine18, VirtualMachin...	2	run0	17
t1_17	IM	VirtualMachine20, VirtualMachin...	2	run0	6
t1_18	IM	VirtualMachine14, VirtualMachin...	2	run0	28
t1_19	IM	VirtualMachine19, VirtualMachin...	2	run0	10
t1_20	IM	VirtualMachine22, VirtualMachin...	2	run0	36
t1_21	IM	VirtualMachine10, VirtualMachin...	2	run0	2
t1_22	IM	VirtualMachine12, VirtualMachin...	2	run0	10
t1_23	IM	VirtualMachine17, VirtualMachin...	2	run0	29
t1_24	IM	VirtualMachine6, VirtualMachine7	2	run0	43
t1_25	IM	VirtualMachine15, VirtualMachin...	2	run0	9
t1_26	IM	VirtualMachine11, VirtualMachin...	2	run0	17
t1_27	IM	VirtualMachine12, VirtualMachin...	2	run0	10
t1_28	IM	VirtualMachine11, VirtualMachin...	2	run0	22
t1_29	IM	VirtualMachine18, VirtualMachin...	2	run0	20
t1_30	IM	VirtualMachine8, VirtualMachine2	2	run0	30
t1_31	IM	VirtualMachine9, VirtualMachine4	2	run0	31
t1_32	IM	VirtualMachine18, VirtualMachin...	2	run0	25
t1_33	chat	VirtualMachine12, VirtualMachin...	6	run0	71
t1_34	chat	VirtualMachine19, VirtualMachin...	3	run0	31
t1_35	chat	VirtualMachine21, VirtualMachin...	4	run0	53
t1_36	chat	VirtualMachine11, VirtualMachin...	5	run0	57
t1_37	chat	VirtualMachine12, VirtualMachin...	4	run0	32
t2_1	IM	VirtualMachine13, VirtualMachin...	2	run1	66
t2_2	IM	VirtualMachine17, VirtualMachin...	2	run1	7
t2_3	IM	VirtualMachine18, VirtualMachin...	2	run1	24
t2_4	IM	VirtualMachine16, VirtualMachin...	2	run1	8
t2_5	IM	VirtualMachine15, VirtualMachin...	2	run1	15
t2_6	IM	VirtualMachine19, VirtualMachin...	2	run1	75
t2_7	IM	VirtualMachine11, VirtualMachin...	2	run1	4
t2_8	IM	VirtualMachine20, VirtualMachin...	2	run1	54
t2_9	IM	VirtualMachine12, VirtualMachin...	2	run1	10

☒ to look at

t1_4 t1_23 t2_4

☒ already looked at

t1_10 t1_11 t1_12
t1_13 t1_14 t1_15
t1_16 t1_17 t1_18

☒ user 'VirtualMachine12'

t1_4 t1_22 t1_27
t1_33 t1_37 t2_9
t2_10 t2_28

☒ word 'Ada'

t1_3 t1_22

(1090529878) **VirtualMachine0:** What do you think? Comparing an application developed probably in 1980s with massive press coverage and ads with a newly created program which has no financial backing. So you are saying Gimp is a

(1090529886) **VirtualMachine12:** Whether the studies are unbiased or not, I don't know. The point is not the number ten. The point is that the _fact_ that Ada is more productive in many ways is irrelevant in terms of open-source

(1090529889) **VirtualMachine0:** Vim for me, but if I were into regular keystrokes I would use GPS.

(1090529896) **VirtualMachine12:** The Ada rules, simple version, for parameter passing mechanisms, are: Elementary types are always passed by copy. Tagged and private types are always passed by reference. Other types have the

(1090529913) **VirtualMachine0:** news:2004416-15013-731621b@foorum... There is JSIS - wonder where they got the idea? :-)

(1090529914) **VirtualMachine12:** Maybe but I never find that in Java unless I`m using unfamiliar APIs like RMI and I don`t develop in Java every day or even every week - ditto digging into the ARM for unfamiliar elements. I`m

(1090529920) **VirtualMachine0:** news:5ad0dd8a.0404160338.5a1c9116@posting.google.com... .... That`s because it would be hard to avoid deadlock in such data structures. (And, as Robert points out, the cost of grabbing a lot of locks

(1090529921) **VirtualMachine12:** versus passing by value makes a difference. It has come up on this newsgroup before. If you pass the same value to a procedure as both an "in" and an "in out" parameter, changes to the "in out"

Figure 4. Using the Conversation Analyzer.

Machine Learning in the Analyzer

Machine learning techniques typically use a large set of training examples to learn a concept. In this case, the training examples would be conversations, labeled by a human as interesting or uninteresting, and the concept would be whether a new conversation that we have not seen yet is likely to be interesting. Each learning algorithm would look at the examples and decide, to the best of its ability, what attributes of an example make it interesting or not. It would then use those assumptions to guess how to label the new conversation. There is one problem with this usual approach: we would have to read and label many conversations to use as examples, which would defeat the purpose of using machine learning to save us work.

Instead, we decided to use an approach similar to one called *active learning*. The goal of active learning is to learn a concept using as few training examples as possible. Instead of accepting a set of labeled examples from the user, the active learning algorithm itself chooses the examples that would help it the most and asks the user to label them. This way, the user ends up labeling only as many as the algorithm needs to learn the concept.

In the spirit of active learning, the Analyzer's learning algorithms choose the examples that they think are most likely to be interesting to the user, and the user labels these.

We can designate several conversations as interesting at first by placing them in a selection in the Analyzer. Then we choose a menu option that starts the learning process (using one of the algorithms that we have investigated). The algorithm uses whatever initial information we have given it to learn a simple model, evaluates all the remaining conversations, and displays to us the one (or more) that is most likely to be interesting according to that model. We inspect it and hit a button to say that it is interesting or uninteresting (or that we cannot decide, but that isn't very useful). If it is interesting, the Analyzer adds that conversation to the selection. If not, the conversation will be re-ordered and no longer appear on the display. Either way, we have given the program more information, and it uses that information to repeat the process until we wish to stop.

The type of model that the algorithms learn and the way that they order conversations is described in the next section. First, though, we had to decide what the *attributes* of a conversation would be, because all classification algorithms need to know the properties that describe an example. The most important aspect of a conversation seems to be the words it contains. The simplest approach would be to say that conversation A has attribute B if it contains word B. More sophisticated approaches might take into account word frequency or order, or even grammatical structures. We could also make use of the names chosen by the users for their chat sessions and information about the location of the connection endpoint. But for now we have used a simple version, and made a "bag of words" — or a list of all the words a conversation contains — to use as attributes. We used the Porter stemmer to remove prefixes and suffixes, and ignored very common words.

The Analyzer code could easily be extended to incorporate other learning algorithms. For now, we have investigated two: decision trees and naïve Bayes. The following sections describe how these methods work and how well they performed in our tests.

For each of the experiments that we describe, we chose beforehand one of the database topics to be the "interesting" one. We arbitrarily chose six topics, and ran each experiment six times, using a different "interesting" topic each time, to study the variation that can occur.

5. Decision Tree Learning

Decision trees are very good at classifying items by finding a few important attributes. They start with the entire set of training examples, look at all the possible attributes, and choose the one that best separates the examples. In our case, the best attribute would be a word that all the interesting conversations have and all the uninteresting conversations do not have (or vice versa). There may not be an attribute that splits perfectly that way, but the decision tree algorithm will choose the one that comes closest, based on one of several possible statistical measures. That attribute will be the first node of the tree, and the training examples will be split accordingly into two parts. For each part, the algorithm repeats this process unless the examples in the part are uniformly interesting or uniformly uninteresting.

The most common statistical measure for the value of an attribute is *information gain.* It uses a value called *entropy* to measure how uniform a set of examples is. The entropy of a set of examples *X* is calculated in the following way:

$$Entropy(X) = -\sum_{j=1}^{m} p_j \log_2 p_j$$

The different values of *j* represent the labels an example can have. In our case we could use *j=1* means "interesting" and *j=2* means "uninteresting." The variable p_j is the fraction of examples in *X* that have the label *j.* The information gain of an attribute *X* over a set of examples *Y* is then:

$$InformationGain(Y / X) = Entropy(Y) - Entropy(Y / X)$$

In our case, *Entropy(Y)* would be the entropy of the original set of examples, and *Entropy(Y|X)* would be the sum of the entropies of the two parts produced by splitting on the word *X.* That is, the value of using the word *X* to split the examples is the amount that it decreases the overall entropy. This is a greedy strategy, which means that it might not find the best possible way to build the tree, but at each step it will do the best it can.

Another possible measure for the uniformity of a set of examples is the *Gini index.* The Gini index of the set *S* is:

$$Gini(S) = 1 - \sum_{j=1}^{c} {p_j}^2$$

Again, the values of *j* are the labels an example can have, and p_j is the fraction of examples in *S* that have the label *j.* The Gini index of *S* when it is split into two pieces (*S1* and *S2*) by attribute *sc* is:

$$Gini^D(S, sc) = \frac{n_1}{n} Gini(S_1) + \frac{n_2}{n} Gini(S_2)$$

The ratios n_1/n and n_2/n are the fractions of examples that were placed into the two subsets. Unlike information gain, for which larger values correspond to better attributes, we would choose the attribute whose split caused the smallest Gini index.

Once a decision tree has been built from training examples, it can be used to label new items. Starting at the top node, it tests whether the new conversation contains the split word, and follows the corresponding branch. It continues this way until it reaches a leaf, where all the training examples are uniform. It then gives the new conversation the same label as those examples.

Using a decision tree algorithm like the one described above, we performed an initial experiment to see whether the presence and absence of words could be used to classify conversations. We simulated ten sets of conversation data using the system described in section 2, gathered them using the programs described in section 3, and used them to do a ten-fold cross-validation of the decision tree algorithm. That is, we divided the data randomly into ten equal sized sets, trained on nine sets to produce a tree, measured the tree's accuracy on the tenth set, and repeated ten times using a different test set each time.

We performed this entire cross-validation process six times, each time labeling a different topic as interesting. One topic was "rec.arts.books.tolkien," which came from a newsgroup that discussed the works of J.R.R. Tolkien, and Figure 5 is an example of a tree from that experiment. The implication of this tree is that if a conversation contains the word "tolkien," it is interesting. If not, but it contains the word "faramir," it is still interesting. These words were chosen to maximize the Gini index, as described above. If it contains neither of those, it is not interesting. This particular tree was completely successful in classifying the conversations in the test set.

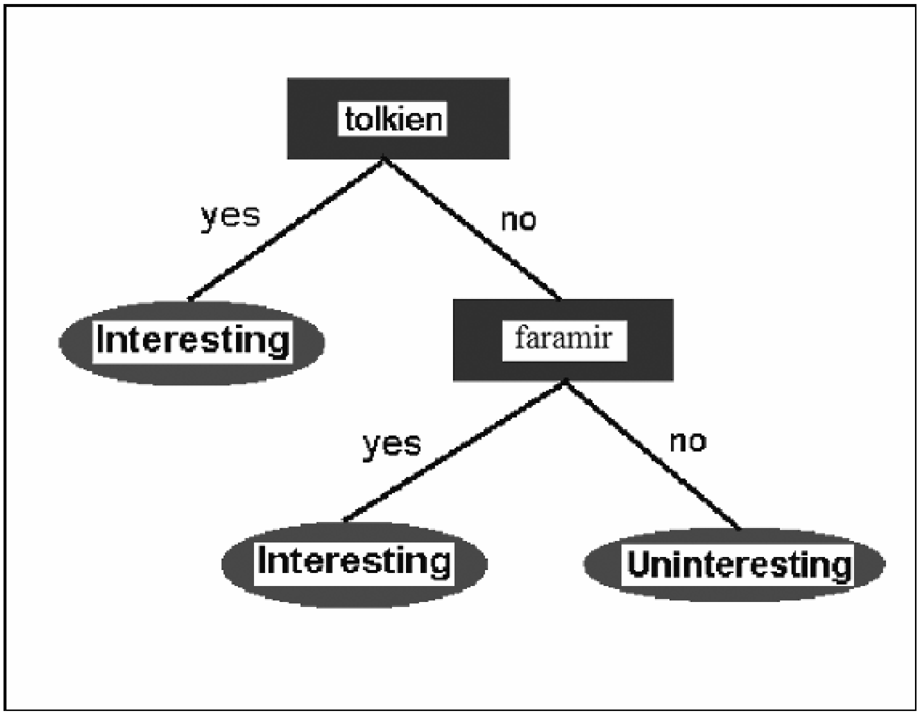


Figure 5. Decision tree example.

The test sets contained 1–2 interesting conversations (those belonging to the topic that we selected beforehand) and approximately 50 uninteresting conversations. The decision tree labeled them according to what it had learned from the training set, and we compared those labels to the true ones. We judged the performance of a decision tree by the following values:

True positive rate — fraction of *interesting* conversations in test set that the tree labeled as *interesting*

False positive rate — fraction of *interesting* conversations in test set that the tree labeled as *uninteresting*

True negative rate — fraction of *uninteresting* conversations in test set that the tree labeled as *uninteresting*

False negative rate	fraction of *uninteresting* conversations in test set that the tree labeled as *interesting*

Performance varied depending on the topic, but in general it was good. The vast majority of conversations were correctly classified. Some were misclassified, but few enough that the approach of using words as attributes seemed likely to have some success. Figure 6 summarizes the results of the ten-fold cross-validation test.

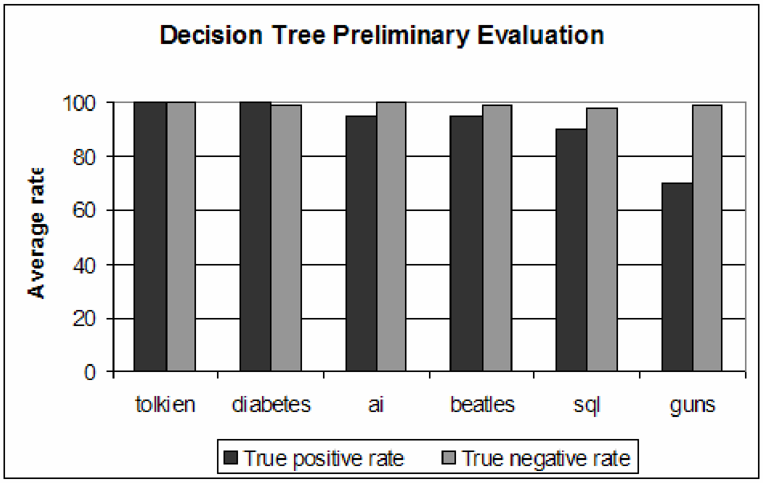


Figure 6. Decision tree cross-validation.

To use a decision tree in the Analyzer, we had to adapt the algorithm to the active learning framework. Instead of starting with a large set of positive and negative examples, we would only have a small set of positive examples that the user had found. The majority of them, which the user would not have labeled yet, we could not use in training. We also had to take time efficiency into account. Using Java, which does not run quickly, the cross-validation tests took around ten minutes to build a tree. That would not be acceptable in the Analyzer, with the user waiting for results. Finally, we needed the tree to give not just a label for a new conversation, but a probability that the new conversation is interesting, so that we could choose the highest-probability one to present to the user.

When we start the learning process in the Analyzer, the algorithm builds a tree with the small amount of information that our initial selection provides. As it does so, it attaches all the examples (labeled and unlabeled) to a branch of the tree. It cares only about splitting the labeled examples, so it will stop when the positives are separated from the negatives without trying to differentiate those from unlabeled ones. This is a key feature, because what we want is for a small number of unlabeled conversations to be grouped with interesting conversations — those are precisely the ones that would be most likely to be interesting.

Formally, our system calculates a probability for each branch of the finished tree. The probability that an unlabeled conversation in a branch is positive is the proportion of positive examples to total examples in that branch. (We chose this definition because examples that are grouped together share some attributes, and unlabeled conversations that share attributes with positively labeled conversations seem most likely to be positive.) It finds the branch with the highest probability and starts offering the unlabeled ones in that branch to the user. It will continue to offer them until the branch runs out of unlabeled conversations or it has gotten three “uninteresting” judgments in a row. At that point it takes the new information (the new labels) and repeats the process.

We tried many modifications to this basic procedure to find which worked the best. Fortunately, it was often possible to see what we needed to do by looking at the trees that it constructed — we could see clearly what policies had led to what structures and change them as necessary. We settled on the following modifications:

- Instead of using all words as potential attributes, we only use words from interesting conversations. This speeds up the tree-building process, and focuses on the most relevant attributes.
- Attributes (words) are selected using the Gini index instead of information gain. Informal observation showed the two measures performing similarly, but the Gini index version seemed a little better.
- An attribute (word) that gets used to create the highest-probability branch is removed from consideration, so that same tree cannot be built again. This forces the algorithm to try new directions instead of using the same attributes repeatedly.

We tested the procedure on all six topics, using two randomly selected conversations from each topic as the initial selection. In general, it tended to present uninteresting conversations for a while, and then start finding the interesting ones fairly quickly. Figure 7 summarizes the performance. Each experiment is labeled by the topic that we used as "interesting" in that experiment.

The performance of a learner on one topic is represented by a stair-step line that starts at the bottom left. Each time the learner offers a conversation, our judgment of whether the conversation is interesting or not determines whether the line moves horizontally or vertically. If it is uninteresting, the next point is to the right along the x axis, and if it is interesting, the next point is up along the y axis. Thus the steeper the line, the better the performance.

The decision tree performance differs on different topics. Each line moves horizontally for a while, which we expected since the learner offers some uninteresting conversations at first before it narrows in on what the user wants. The ideal line is the "Diabetes" dataset, where there were only four uninteresting conversations, and then it found all the interesting ones in succession. With the "Tolkien" and "SQL Server" topics it took longer to start finding interesting ones, but also found them quickly once it began. It alternated between a couple hits and a couple misses on the "Guns" topic, overall doing quite well. The "AI" and "Beatles" datasets gave it more trouble, but it still performed reasonably well. There were about 500 candidate conversations, and the decision tree learner found almost all of the 10–12 interesting conversations in each topic in about 30 tries.

In terms of running time the trees built almost immediately at the beginning of a search, but took up to 3 minutes to build towards the end, which was acceptable for our purposes but not ideal for an operational setting. Much of this performance can be attributed to implementing these tools in Java. Java was chosen for ease of implementation, and these techniques could be re-implemented in another environment to address speed concerns.

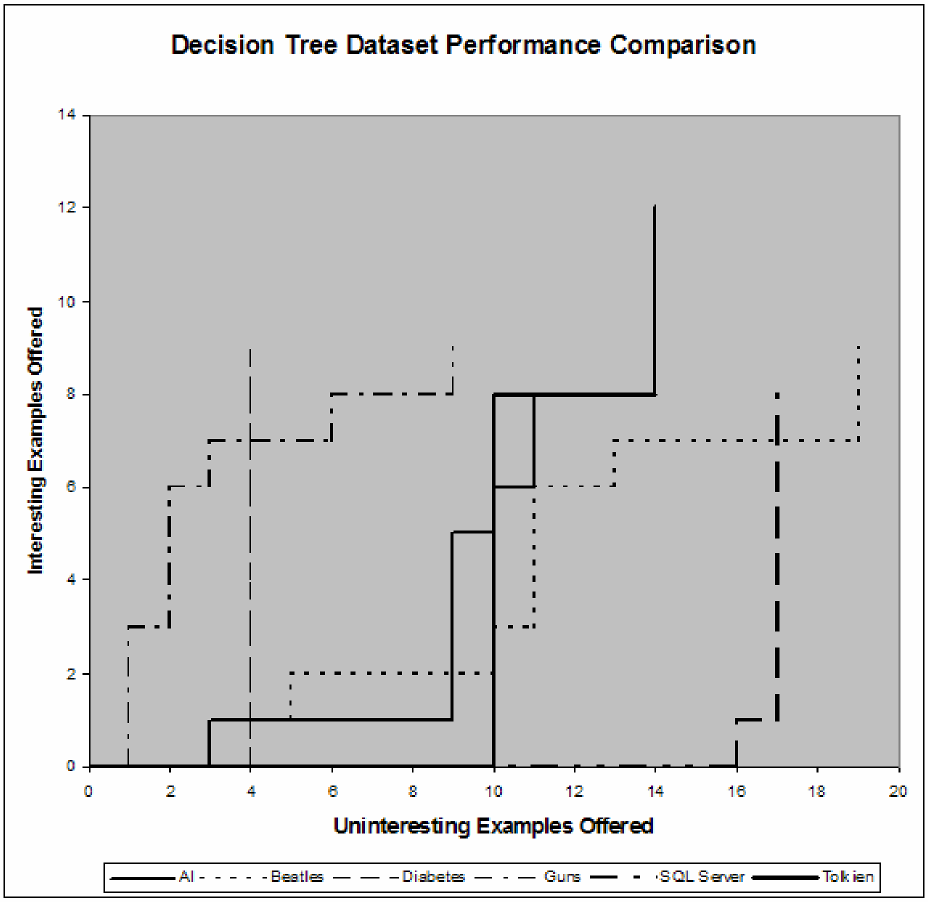


Figure 7. Decision tree performance.

6. Naïve Bayes Learning

The second machine learning method that we applied to the problem was a Naïve Bayes classifier. In contrast to a decision tree, which selects a few important attributes to use in classification, a Naïve Bayes classifier uses all available attributes to make a classification decision. The Naïve Bayes classifier arrives at its decisions by applying Bayes' rule. Formally, Bayes' rule is as follows:

$$P(A|B) = P(B|A) * P(A) / P(B)$$

$P(x|y)$ indicates the conditional probability of x given y, and $P(x)$ indicates the prior probability of x. For our work, the specific application of Bayes' rule that we use is:

$$P(class|conv) = P(conv|class) * P(class) / P(conv)$$

Here, *class* is a random variable which can take on the values *interesting*, *uninteresting*, or *unclassified*; and *conv* is the conversation for which we are attempting to calculate the probability distribution across classifications.

Our conversation model represents each conversation as a Boolean vector, $conv = \{w_1, w_2, \ldots w_i, \ldots\ldots w_n\}$ in which $w_i = true$ if the conversation contains word i and $w_i = false$ if not. To simplify the modeling, we then assume that, for all i and j, element w_i is conditionally independent of w_j given the correct classification of the conversation. In less formal terms, this assumption is saying that once we know the correct classification for a conversation, the probability of any specific word appearing in that conversation is independent of the presence of any other word in the conversation. This assumption is why the model is described as "naïve."

The conditional independence assumption, however, allows us to calculate the *P(conv|class)* term as a product of individual word probabilities for the given classification. Thus, we can rewrite *P(conv|class)* as follows:

$$P(conv|class) = P(word_1=w_1|class)*P(word_2=w_2|class)\ldots*P(word_n=w_n|class),$$

where $P(word_i=true|class)$ is interpreted as the probability that a conversation contains $word_i$ given that its correct classification is *class*. This observation is the key to the implementation, because we can estimate the conditional word probabilities simply by dividing the number of conversations of a given class that contain the word in question by the total number of conversations in our dataset of that class (Maximum Likelihood Estimation). We can estimate the prior probability of the classification, *P(class)*, in an analogous manner. The prior probability of the conversation, *P(conv)*, can be ignored since it will be constant for a given conversation. Thus, we have all of the information that we need to solve the equation and derive the probability distribution across classifications for a given conversation.

To integrate this procedure into the Conversation Analyzer, all conversations start out as *Unclassified*, with the exception of any initial positive seeds. We can then use Maximum Likelihood Estimation to calculate word probabilities and class priors (as described above). Then we apply Bayes' Rule to obtain estimates for *P(Interesting|conv)* and *P(Uninteresting|conv)* for each conversation that is *Unclassified.* We find the one that maximizes the ratio *P(Interesting|conv)/P(Uninteresting/conv)* and return this conversation to the user. The user makes a classification decision for the conversation we returned and we repeat the process until boredom or exhaustion. Since the algorithm runs quickly, we can actually update our probabilities and re-rank all remaining unclassified conversations each time we get new information from the user. Since we are using only the classifier to rank-order the unclassified conversations by their potential for being interesting (i.e., we are not interested in making actual classification *decisions*), we need not worry about calibrating the raw probabilities that emerge from the classifier itself.

It is important to note that running the Naïve Bayes active learner with no seeds at all achieves results competitive with, and often superior to, those achieved by the seeded Naïve Bayes and the decision tree. Typical behavior for all three algorithms is an initial "calibration" period in which several uninteresting conversations are presented, followed by a brief period in which all (or almost all) of the interesting conversations are discovered in quick succession.

Below are two charts that summarize the performance of the seeded (Figure 8) and unseeded (Figure 9) Naïve Bayes classifiers, using the same datasets as the decision tree. Again, each experiment is labeled by the topic we chose as "interesting" for that experiment, and there were 10–12 interesting conversations in a set of about 500.

For both initial conditions, across all datasets, we observe a characteristic upward trend indicative of the behavior described above. The performance graphs are constructed in the same way for the decision tree learner. Comparison between the graphs will show that no method is always better than the others; we see that each learner outperforms the others on two of the topics.

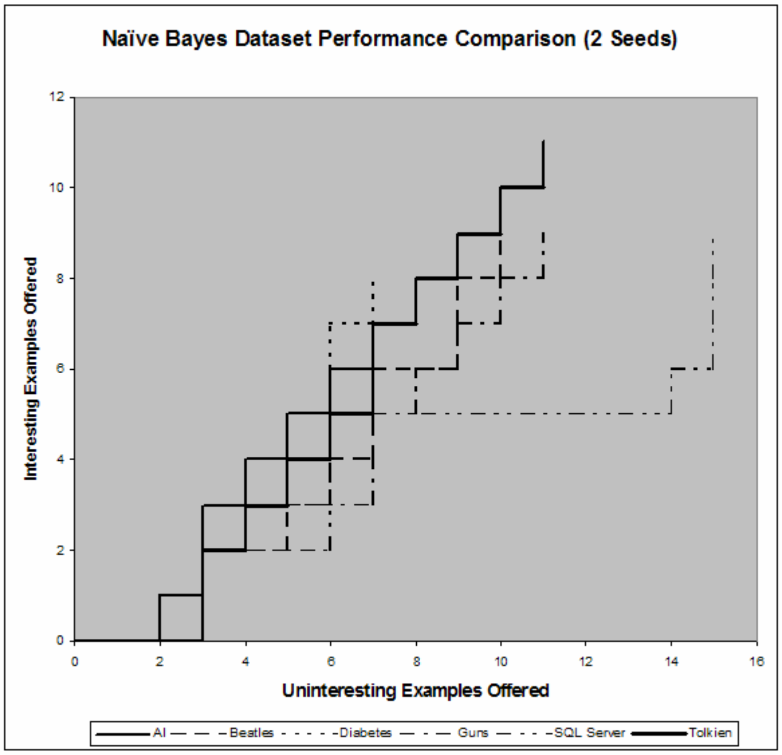


Figure 8. Naïve Bayes performance with seeds.

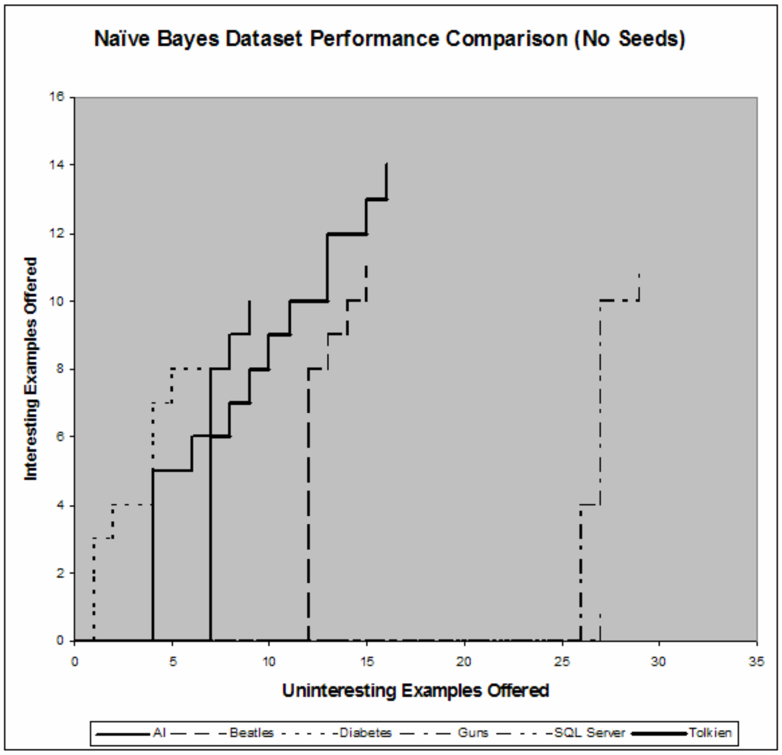


Figure 9. Naïve Bayes performance without seeds.

7. Future Work

The results of this preliminary work point to several promising avenues of future research to pursue. We believe it could be interesting to look at combining the Naïve Bayes and the decision tree active learners. Two advantages of the Naïve Bayes over the decision tree are speed and the fact that the former can be used without any initial seeds. The decision tree, on the other hand, seems to be less prone to getting stuck in local maxima than the Naïve Bayes. Perhaps we could use the un-initialized Naïve Bayes learner initially and, once it ceases to be effective, use the results generated to that point as seeds for constructing a decision tree.

Also, it is clear that there is more information contained in a conversation than can be captured by the simple “bag-of-words” approach. A possible long-term direction of research is to try to capture some of this additional information by building more complex models of conversations. Additional information such as word frequencies, word proximity, grammatical structures and conversation participants could yield a better classification of interesting conversations.

We have used synthetic data here, and an obvious improvement would be to use real conversation data, at least for the interesting conversations. We could devise or copy conversations that we want to be able to find, and test the algorithms’ ability to do so.

Finally, we believe it would be worthwhile to investigate the performance of active learners based on other well-known machine learning techniques, like hidden Markov models, support vector machines, clustering, probabilistic link index searching, latent semantic indexing and genetically trained neural nets. We have started to look into a hidden Markov model approach, and it appears to perform similarly to the decision tree and naïve Bayes learners. The HMM has the advantages of considering word sequence instead of just independent words, and it could also easily be used to classify conversations into more than two categories. It seems to favor short conversations, and in fact these appear to account for most of its misses; if we could repair this bias, it might perform substantially better. Our evaluations of the HMM are only informal at this point, but since the implementation is in place, work on it could be continued easily. Since the Conversation Analyzer is designed in a modular fashion, it is also relatively easy to incorporate new methods into it for further testing.

8. Conclusions

We have presented an active learning approach to the problem of searching large amounts of captured conversation data, and evaluated three learning techniques that we have applied. Our approach differs from the traditional machine learning paradigm, since it starts with a very small amount of information and proceeds incrementally instead of learning all at once with a large batch of data.

None of the three approaches was clearly the best; each out-performed the others on two out of the six datasets. However, we can offer observations on why some topics were easier than others. One characteristic of topics that were better retrieved is a unique vocabulary. The "diabetes" dataset has this characteristic, and tended to be picked up well by all the learners. Another characteristic is a sufficiently narrow focus. The "AI" dataset contained conversations on two distinguishable facets of the topic of artificial intelligence, and this often caused learning to occur in two separate runs. A third characteristic is a good seed, which applies of course only to the seeded Bayes and decision tree methods. Finding a seed that is representative of the dataset can substantially improve the results of the search process.

These observations lead to several hypothetical questions. What if clients use euphemisms instead of key vocabulary terms in order to conceal their conversations? There are two ways in which these learners could still catch the concealed exchanges. One is if non-euphemized words, those the clients do not consider to be important, are used similarly among their conversations. The learners contain no bias towards words a human might feel are important, and therefore they might find things we would not. The other way is if a client slipped and used an important term alongside euphemisms; this would draw the learner's attention to the euphemisms and lead to finding all euphemized conversations.

Another question that is important to address is how well these techniques would scale when the volume of data is much larger. The naïve Bayes methods are inherently linear in the number of conversations being searched, so they would still run quickly. The decision tree method is more computationally complex, but because of the changes in the algorithm, it also will tend to scale linearly with the number of conversations being searched. Therefore, we do not anticipate running time to be a large problem. Just as important, though, is how quickly the learners start to focus in on the relevant conversations. We would need to run large-scale experiments to discover the answer to this question.

It is important to note that the approach we have taken in this work does not result in any data being discarded. Rather we are presenting the data in an order that is likely to be more useful than random selection. At no time does our algorithm discard a conversation without being instructed to do so by the analyst. Thus, there is nothing to preclude an analyst from examining every captured conversation.

To conclude, the programs that we have developed and their uses are listed below:

- Message Simulator — simulates communicating agents; currently uses the AIM TOC protocol, but could be extended to other conversation protocols or different communication channels.
- Communication Capture programs — sniff and reconstruct communication streams; currently expects the AIM TOC protocol, the modular design simplifies extension.
- Conversation Analyzer — graphical interface for examining and manipulating conversations; supports several intelligent search algorithms and could be extended to include other search techniques or communication protocols.
- Naïve Bayes active learner — standalone intelligent search algorithm; could be integrated into the conversation analyzer.

9. References

1. *Machine Learning.* Tom M. Mitchell. McGraw-Hill, 1997.
2. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids.* R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. Cambridge University Press, 1998.
3. *An active learning framework for content-based information retrieval.* C. Zhang and T. Chen. IEEE Transactions on Multimedia vol. 4 no. 2, June 2002.

Distribution

1	MS 0123	LDRD Office, Donna L. Chavez, 1011
1	MS 0455	Steven Y. Goldsmith, 5517
1	MS 1202	Ann N. Campbell, 5940
1	MS 1205	Dept. 5900 file copy
6	MS 1397	Richard F. Maclin, 5946
2	MS 1397	Charles E. Nove, 5946
1	MS 9018	Central Technical Library Files, 8945-1
2	MS 0899	Technical Library, 9616