INFORMATION FLOW IDENTIFICATION IN LARGE EMAIL DATASETS

by

Arseniy Akuney

B.Sc., Simon Fraser University, 2008

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

December 2011

Abstract

Identifying information flow in emails is an important, yet challenging task. In this work we investigate several algorithms for identifying similar sentences in large email datasets, as well as an algorithm for reconstructing threads from unstructured emails. We present a detailed evaluation of each algorithm in terms of accuracy and time performance. We also investigate the usage of cloud computing in order to increase computational efficiency and make information discovery usable in real time.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Identification of information flow in an enterprise is an important, yet challenging task. In an enterprise, information flow occurs in all forms of communication, both physical and digital (i.e. – formal/informal conversations and meetings, video conferences, chat systems, e-mail messages, text messages etc.) Speech and video based forms of communications are difficult to transcribe and for obvious employee morale and statutory privacy reasons such recording is rarely done. The same is also true for digital forms of communications, with the exception of e-mails which are far more conveniently recorded or transcribed. Ease of access to e-mail archives within an organization is the motivation for our focus in this work on identifying information flow in e-mails.

Capturing information flow in e-mails is important for a variety of different reasons. The purpose can be as simple as identifying a person in a company that is the source of valuable ideas or it could be used for more complex purposes like data retention policies and ensuring access control policies. E-discovery [4][5] processes can benefit greatly when information flow is presented in a coherent manner with a logical structure.

The current approach to monitoring information flows through e-mail is to assume that the information flow is confined to threads. Threads are created using e-mail header fields “References” and “In-Reply-to” as well as the “Subject” field. In the work described in this thesis we found that thread structure does not identify all flows and is prone to errors. Errors stem from the fact that headers can be missing or erroneously present. One of the examples of incorrect headers originates in using

“reply” button for e-mail address retrieval rather than for actually replying to the message. An example of information flow that does not belong to a thread is a copy/paste from one e-mail to another.

In this work we present an algorithm for thread reconstruction based on text matching. E-mails are assigned parent-child relationships based on text they share. We also present a set of tools that aid users in discovery of hidden information flows in e-mails via sentence similarity matching.

For the purposes of this study, the Enron e-mail dataset [6] was initially chosen as the corpus since it exhibits many properties that are common to a real world setting. It contains a large number of e-mails (>500,000) that are spanned across a time period of over two years. However, as discussed below, a comparison of our reconstruction algorithm versus header-based threading of the Enron dataset was not useful due to the variability of the header information in the Enron dataset. We turned to a mailing list archive dataset for testing our thread reconstruction algorithm.

The remainder of the thesis outline is as follows. Chapter 2 provides overview of information flow while Chapter 3 discusses information flow in e-mails. Chapter 4 describes related works. Chapter 5 highlights some of the problems with dealing with e-mails. E-mail parsing and algorithms for sentence matching are located in Chapters 6 and 7 respectively. Chapter 8 presents an evaluation of sentence matching algorithms. Chapter 9 discusses performance improvements with the help of the cloud. Chapter 10 describes thread reconstruction algorithm while Chapter 11

presents its evaluation. I conclude paper with suggested improvements in Chapter 12 and conclusion in Chapter 13.

Chapter 2: Information Flow

Information flow can be described as the transfer of information from one person to another. One example of information flow that we encounter on a day-to-day basis is rumor spreading or gossiping. In the extreme case of celebrity gossip information can travel very fast and reach many people in short period of time, although original meaning can be lost in such a transfer.

In an enterprise, information flow is most commonly exemplified in the form of written documents (such as e-mail conversations, official company proceedings, legal documents, etc.) or person-to-person interactions (such as telephone or regular conversations). On a way from one person to another, information can pass through many people and through various mediums**.**

Identifying information flow is important because it allows us to answer the following questions:

2.1 What was the source of information?

In large organizations many people contribute to the discussion of ideas. Often it is beneficial to know from whom an idea originated, as often only a few people are the source, and very often credit goes to someone else. It is important to identify such valuable people and retain them within an organization. Source of information is also useful for identifying who caused an error, or identifying who is spreading the information. If information happened to originate outside of an organization, it might be an important factor in considering its validity and credibility.

2.2 How did/does information propagate and change?

If you are aware of all information flows in your system you have capability to enforce access control policies [7]. Access control policies let you ensure that information distribution is limited to those who are allowed to see it. This is especially useful for preventing accidental information disclosures such as leaking confidential company information prior to an offering of shares to the market.

2.3 Where is the information located?

Today's business and legal requirements mandate the implementation of data retention policies [8]. Data retention is a set of policies that regulate the lifespan of documents. It would be of great benefit to know if a document was discarded in an information flow in order to prevent the loss of valuable information.

2.4 Who had access to information?

Consider the following situation: John A was promoted to a new position in another department. He happily accepted and was gone as fast as he could. New employee, Dave B, was hired to replace him. Getting Dave up-to speed is a priority however, with John gone, this is hard to do. In this situation knowledge of all information flows that involved John can be used to help Dave with getting familiar in all company affairs.

2.5 E-discovery

Electronic discovery is a process in which electronic information such as e-mails, instant messaging chats, documents, and other is searched with the intent of using it for legal purposes. An example of such a case that involves the e-discovery process is a legal deposition. A typical scenario for a legal deposition case includes lawyers obtaining all electronic data from a party that is relevant to the ongoing litigation process. After information has been obtained it has to be arranged so that conclusions can be drawn from it. As we can see identifying information flows is major task in electronic discovery.

Chapter 3: Information Flow in E-mails

Identifying and extracting information flow from person-to-person interactions is an extremely difficult task since very seldom conversations are actually recorded and transcribed. A much more tractable, yet still difficult task is to identify information flow in an enterprise, through the form of e-mail communications.

3.1 E-mails

E-mails are used for transferring digital messages from an author to one or more recipients. An e-mail is composed of Header and Body [21]. E-mail header contains information about the message arranged into header fields. E-mail body contains message itself which can be in a plain text or an html format. Message can also contain attachments such as documents or images.

In this work we are going to be referring to several e-mail header fields. Their description is given below:

- *From:* An e-mail address of the author of the message. This field is required.
- *To:* An e-mail address of the recipient(s).
- *Date:* A date and time when the author sent the message. Note that this field is populated by the sender's mail client. This field is required.
- *Received:* This header field contains information about date and time when the message arrived to the mail server. A message usually passes through several mail servers from origin to destination and its path can be traced via this field.

- *Subject:* String describing topic of the message.
- *Cc:* (carbon copy) A list of email addresses where the copy of this message is sent to.
- *Bcc:* (blind carbon copy) A hidden list of email addresses where the copy of this message is sent to.
- *Message-id:* A unique message identifier.
- *In-reply-to:* “Message-id” of the message that this is a reply to.
- *References:* A “Message-id” of the message that this is a reply to joined with “References” field from the parent. The intention is for this field to hold "Message-ids" for all messages in an e-mail chain that led to this email.

An example of the e-mail header with the fields described above is shown in Figure 1.

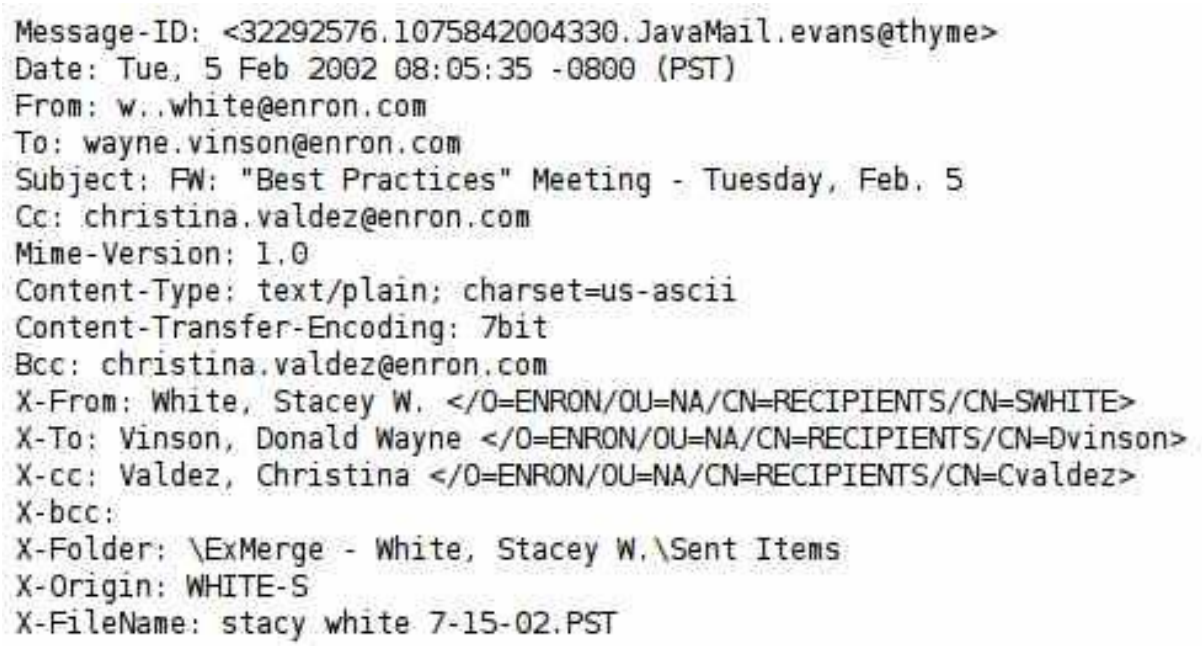

Figure 1: An e-mail header.

3.2 Header based e-mail threads

One of the more popular ways of tracking information flow in e-mails is done by the header based e-mail threads [9]. An e-mail thread consists of messages that are linked together by a user action of replying to or forwarding a message. When a user presses “reply” or “forward” buttons in an e-mail client e-mail header fields “References” and “In-reply-to” are adjusted storing information required for e-mail thread reconstruction. Some e-mail clients realize the importance of identifying threads as easily as possible so they build their clients around thread idea. One such example is Google Gmail that keeps messages in a thread view. Other clients use simple threading to maintain information flows.

JWZ algorithm [20] is one of the widely used algorithms for threading messages. The algorithm relies on “References” and “In-reply-to” e-mail header fields to provide information about threading. The algorithm also makes use of the “Subject” header field to aid in threading.

3.3 Why header based e-mail threads are not sufficient

One might think that header based threading is all we need in order to identify all information flows in e-mails, but this is far from being true. There are several important reasons why threading by headers is insufficient:

- E-mail headers are not always present in the e-mail corpus. They could be discarded or simply not saved.
- E-mail header fields can be incorrect or incomplete. An example of such behavior is used by all of us almost daily. Whenever you open an e-mail and

click “reply” “References” and “In-reply-To” header fields are populated. However often we use “reply” in order to reuse an e-mail address rather than actually replying to an e-mail.

- E-mails from different threads can share subjects. In this case threading based on subject lines will fail.
- Any information that is passed outside of thread is not captured; that includes something as simple as a copying and pasting a part of the e-mail into new e-mail or, in a more complicated scenario, paraphrasing an idea expressed in some e-mail.

3.4 Our approach to identification of information flow

We developed an algorithm for reconstructing threads and tools that aid discovery of additional links between e-mails that might be of interest. In addition, we developed several filters that we apply before thread reconstruction, for dealing with missing header fields, duplicate emails and other issues discussed below. We made no assumptions about dataset thus it can contain all kinds of problems which are found in a real world scenario.

Reconstruction of threads is done based on matching e-mails that have sentences in common. Additional links are found by matching e-mails that contain similar sentences. A more detailed explanation of these algorithms as well as identification of similar sentences is presented in later chapters.

3.5 Enron dataset

Enron Corporation was a large American company that was bankrupted after their irregular accounting procedures were discovered. The Enron case is a great example of a typical deposition. As a part of the litigation process, lawyers for the claimants obtained some of the electronic information from the company and went through the discovery process in order to identify guilty parties. An e-mail dataset was made public during an investigation into the case. That e-mail dataset contains data from about 150 users, mostly senior management of Enron [6]. The corpus consists of about 0.5 million e-mails. These e-mails retain almost no header structure required for successful reconstruction of threads. In particular e-mail header fields “In-reply-to” and “References” are missing. Any information about mailing lists is also lost. This corpus contains invalid e-mails, lots of duplicates and other artifacts and as such is a great example of a real world e-mail dataset. We used Enron corpus for e-mail parsing and evaluation of algorithms for identification of similar sentences.

3.6 Mailing lists archive dataset

While the Enron dataset is a good example of an e-mail dataset we realized that it is impossible to properly evaluate our thread reconstruction algorithm against it. Lack of header fields is to blame. We were able to obtain an archive of several mailing lists from the list moderator. There are 5 lists with a total of about 7,000 e-mail messages over 5-7 years. All e-mails in this archive had their header fields

intact so conventional thread reconstruction was possible. We have used this dataset to evaluate our algorithm for identification of information flows.

Chapter 4: Related Works

Identifying information flow in an enterprise is far from being considered a solved problem. Work has been done on attempting to identify desktop workflows for each user [1]. However the authors decided to use coarse grained actions such as copy/paste, file copy and SaveAs for tracking information flow. Coarse grained flow will allow tracing users involved in the lifespan of the document but the extent of their involvement will be unknown. Additionally this approach assumes that an enterprise adopted use of action tracking for all user machines. We decided that such conditions are a limiting use of our tools in the real world.

Identifying information flow is not solved even in a domain of e-mails. There has been work done in data-mining social graphs based on information flow [3]. The authors assume that information flow is confined to traditional thread structure defined by header fields. We know that such assumptions do not hold in the presence of erroneous header fields.

An attempt to reconstruct threads in absence of “References” and “In-Reply-To” header fields has been made before [2]. Messages were assembled into threads using text matching. However the authors only considered conventional threads and required for all e-mails in a thread to share a subject. Our approach for thread reconstruction builds on a text matching concept from their work. The differences are:

- We did not assume that e-mails in the same thread have same subjects.
- Rather than identifying replies, we identified shared sentences between e-mails.

Chapter 5: E-mails are Complicated

When dealing with e-mails one might make an erroneous assumption that e-mails are simple textual documents. However this assumption is quite far from the truth. E-mails have several unique characteristics that distinguish them from other textual documents. In this chapter we present a list of complications that we encountered with our e-mail datasets.

5.1 Language choice

A lot of people consider e-mail communication to be informal. In these cases language used can be quite non-traditional and colloquial. Such colloquialism creates an abundance of problems related to this issue:

- Not capitalizing the start of the sentence. Sentence parsing (discussed later) relies on this information in order to achieve acceptable accuracy.
- Omitting '.' character in the end of the sentences which also greatly affects sentence parsing.
- Spelling errors were present in many e-mails. A large number of such errors lead to missed connections between sentences. A definite improvement on this project is to add a spelling checker and fix blatant mistakes. These improvements are planned in a future work.
- Presence of emoticons. Emoticons are facial expressions represented by punctuation and letters. Some of the examples are ":) : -)". Presence of

emoticons complicates parsing process by introducing extra characters that do not carry significant meaning.

- Use of internet slang such as “lol”, “rofl”. Just as emoticons, slang clutters sentences.

An example of an e-mail that uses informal language is provided in Figure 2.

```
hey - how's your sexy body? don't have much time right now, but thought i'd
drop a line to tell you that i'm so broke i'm considering picking up some
unmentionable weekend job and therefore cannot make it down there for
prezzy's day weekend. i'm so pissed! but it will happen soon - and anytime
you're up here you need to stay with me - we have a guest bedroom for gosh
sakes. also, i met malia at a party the other weekend - talk about a freakin'
small world! she was so sweet, but then i'm not surprised - she'd have to be
to put up with you. (ha, ha) anyway, take care, call me soon, and toast my
absence with a bottle of wine tonight.
```

Figure 2: E-mail body that does not use formal language.

5.2 Message structure

Although e-mails are primarily textual documents they do not necessarily contain only written text. Here are some examples of such artefacts:

- Some e-mails contain textual tables, probably pasted from Excel worksheets.
- Several people had ASCII pictures for e-mail signatures.
- Programming code was included in the body of e-mail.
- HTML messages. Such messages require special parsing which was not done in the scope of this project
- Attachments. Some e-mails include attachments such as images or files. At times it is hard to recognize that e-mail contains attachment without looking for in text headers, which can vary.

An example of an e-mail that contains tables is provided in Figure 3.

U.S. INDICES
(12:30 p.m. EST)

Market	Value	Change
DJIA	9,827.00	- 74.38
Nasdaq Comp.	1,859.74	- 20.77
S&P 500	1,134.68	- 7.98
NYSE Advancing Issues		1,102
NYSE Declining Issues		1,830
NYSE Trading Volume		543 mln
NASDAQ Advancing Issues		1,361
NASDAQ Declining Issues		1,875
NASDAQ Trading Volume		883 mln

U.S. TREASURIES

Value	Yield	Change
6-month bill	2.02%	n/a
5-year note	4.29%	- 15/32
10-year note	4.95%	- 21/32
30-year bond	5.35%	- 15/32

Figure 3: E-mail body that contains tables.

5.3 E-mail client artifacts

In this category we list artifacts that arise from using various e-mail clients. Each client uses different formatting options.

- How e-mails are replied to or forwarded. A conventional way is to include brief header and body of an e-mail that one is replying to or forwarding. Conventionally each line of the included message is prefixed with “>” character. However there is no single standard and each e-mail client follows their own rules for replying and forwarding emails.

- Some e-mail clients insert “-” character whenever a word runs over the line limit which resulted in random inclusion of this character before, after or in between 2 words.

An example of an e-mail that does not prefix included message with “>” characters is shown in Figure 4.

```
Jeff-
Sorry, I am a bit out of it today.

-April
----- Forwarded by April Hrach/SF/ECT on 10/18/2000 10:21 AM -----

        April Hrach
        10/18/2000 09:40 AM

                To: Jeff Dasovich/NA/Enron
                cc:
                Subject: 2 logos

Jeff-
 Here are the two logos.  If you set them up on the page the way you want
them, then I will print them out for you.

-April
```

Figure 4: Forwarded e-mail without “>” characters in included message body

5.4 Extreme issues

Besides the problems mentioned above, there were several unique problems that stood out amongst others when we worked on our datasets:

- E-mails from the past. Several e-mails had date header field set to year 1978.
- E-mails from the future. Several e-mails had date header field set to year 2040.
- An e-mail without message body. A unique case of a bug in mailing list archiving code of some sort.

- An e-mail with header that could not be parsed by existing e-mail parsing algorithms. This particular e-mail had subject header field with “\nDate:” substring. And as such it was impossible for the parser to distinguish between date header field and a long subject header field.

An example of an e-mail with incorrect “Date:” e-mail header field is shown in Figure 5:

Figure 5: E-mail header with incorrect “Date:” field.

Chapter 6: E-mail Parsing

Before we started looking into sentence similarity we had to parse the e-mails in the Enron corpus. The corpus contained a lot of irrelevant information, which should be eliminated. In this section we describe the necessary steps to parse the data before we can apply any information extraction algorithms.

6.1 Duplicates

The original Enron corpus contained more than 500,000 e-mails with almost 50% of them being duplicate e-mails. There were two reasons behind the presence of duplicates: E-mail was stored in several folders for the same user (simple duplicates) and e-mail was being stored in folders of multiple users (complex duplicates). Simple duplicates were identified by creating hash of payload (body of an e-mail) and headers of an e-mail. If two or more e-mails have identical hashes they are identical. The subsequent repeated e-mails were removed from the corpus. In the case of complex duplicates, headers were merged in order to retain all original information.

6.2 Signatures

An e-mail signature is most commonly a paragraph that sender appends to all/most of his/her e-mails. Signatures can contain a user name, phone number, address and sometimes even carry a legal disclaimer. It is important to identify signatures as they do not contribute to the transfer of information flow. An example of the signature is shown in Figure 6:

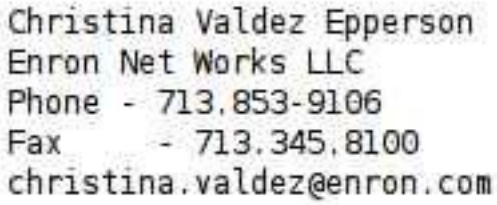

Figure 6: E-mail signature.

Simple signature identification was used for the purposes of this study. For each user present in the Enron corpus the last paragraph of the user's e-mail was extracted. If a certain paragraph appeared in more than 10 e-mails then it was deemed to be a signature paragraph.

To validate this selection criterion, one hundred randomly selected paragraphs, identified to be signatures were manually checked for errors, of which 95% were marked correctly. Paragraphs that were marked incorrectly were of type: "Thanks, <person>" or "<person>" which are common closing paragraphs in an e-mail, and since they do not carry any useful information, they were not removed from the signature list.

6.3 Included headers

The header paragraph is a paragraph that contains the header information of the included e-mail, which is being forwarded or replied to. This header inclusion is a common occurrence and is done automatically by e-mail clients. Since information included in headers is not important to sentence matching, it should be excluded from the sentence set. Headers can be of various types [2]. Two headers that were

the most common in the Enron corpus were identified in order to capture the general version of a header. These headers contained either “[]+*Original Message*[]+” or”[]+ *Forwarded by* ” lines. Generalized version of the header is assumed to contain at least two header tags such as “From:”, “To:”, “Sent:”, “Subject:” and “cc:”. An example of the included header is shown in Figure 7:

```
 -----Original Message-----
From:    Beck, Sally
Sent:    Monday, February 04, 2002 12:39 PM
To:      Valdez, Christina; Reeves, Leslie; Gossett, Jeffrey C.;
Subject:         RE: "Best Practices" Meeting - Tuesday, Feb. 5
```

Figure 7: Included header

After randomly sampling the output of the header tagger, it was determined that header paragraphs do not always come as a standalone paragraphs, a sentence can precede the header in the same paragraph. The initial assumptions were refined to include a check for a sentence before and after the header tags. We then proceeded to randomly chose 100 paragraphs that were tagged as a header or signature and manually verified them. One hundred randomly selected paragraphs that were tagged as header were manually verified, of which 97 were correctly labeled.

6.4 Sentence identification

At this stage sentence sets were extracted from the e-mail payloads. This was done by first splitting the payload into paragraphs and then into sentences. All automatically inserted bird (“>”) characters were removed using regular predefined

expressions before splitting the payload into paragraphs. Payloads were then split into paragraphs using double newline characters as separators ('\n\n'). Each paragraph was checked for being a signature or included header paragraph.

If a paragraph was not identified as a signature or header it was split into sentences using the Punkt sentence tokenizer [10] that comes in nltk [12] package. A dictionary of sentences was created and entered into a database, along with paragraphs that were not parsed. As a result, we obtained 3,793,935 sentences (2,092,593 unique).

6.5 Word identification

Each sentence was split with regular expression that selected strings of alphabetic characters of length more than two, provided that a string had a word boundary before and after. If a word has a length of 2 characters or less then it is either a non-dictionary word or one of the stopwords so it can be ignored. Stopwords are short words that are commonly used such as 'a', 'the', 'is', 'at' and so on. Each string was then checked for being one of the stopwords and also for being less than 22 characters long. A study [22] has shown that the longest words likely to be encountered in general text are “deinstitutionalization” and “counterrevolutionaries”, with 22 letter each. Therefore 22 characters is a generous upper boundary that should remove all long strings that are not words. To generate a stopword list, the *nltk* english stopword list, together with 4 most common words that were found in the corpus (which were 'http', 'com', 'enron' and 'would') were used. Two dictionaries of words were gathered. One of the dictionaries contained unaltered words and the

other one contained words that were stemmed using Porter stemmer [11] that comes in *nltk*.

After checking lists of words that had an occurrence count of only 1, two obvious problems were apparent; a lot of the time a double dash (“--”) was present in between two words and single dash (“-”) was either a leading or trailing character, resulting in proper words making that list. After removing all double dashes from sentences and the trimming words, 127,187 unique stemmed words that appeared more than in one sentence and 158,547 unique full words that appear in more than one sentence comprised the list. The number of words that appeared only in one sentence was more than 100,000 and is due to several factors like: spelling errors, writing words with dashes in them and others which can be attributed to a not so strict style of e-mail writing.

6.6 Bigrams and trigrams

During sentence parsing into words we also records bigrams and trigrams. Bigrams are sequences of two words while trigrams are sequences of three words. They were used for searching through the corpus whenever we had a word of interest.

6.7 Sentence rewriting

As a final parsing step, each sentence was rewritten in terms of words that it contained, if certain conditions were satisfied. Each sentence was checked for several factors:

- The total number of words was taken to be a number of words that were identified before any of the words were discarded due to being stopwords or uncommon words. If it was less than 4 or more than 100 then either sentence was too short and matching could return meaningful result or the sentence was too long and most likely not a proper sentence. The minimal and maximal lengths were chosen rather arbitrarily, but keeping in mind that the average sentence length should be 15 to 20 words, accordingly to Guidelines of English language [13].
- The length of identified words was checked against initial length of the sentence. If it was found that more than 50% of a string consists of characters not in identified words then we concluded that the sentence contained too many nonalphabetic characters and any match to this sentence was deemed to be questionable. Moreover, it was likely that the particular sentence was not a proper sentence.
- The number of good words is the number of words that appear in more than one sentence. If the number of good words is less than 30% of all of the words in the sentence or number of good words is less than 2 then we can conclude that there is not enough information in the sentence to perform matching.

If a sentence passed the above-mentioned test, then two lists were created, one consisting of the stemmed words present in the sentence and the other consisting of full words. As a result 1,577,695 unique lists of stemmed words were identified and 1,584,356 unique lists of full words. One hundred randomly selected sentences, that

were marked as skipped, were checked for proper labeling, all of which were indeed properly labeled. Thresholds were adjusted several times to ensure that valid sentences were not discarded. It is worth noting that only about 100 words had an extremely high occurrence, appearing in more than 50,000 sentences.

Chapter 7: Similar Sentences

Once the corpus had been parsed to the desired structure, we were ready to perform sentence matching. Several algorithms were tried out and evaluated for their performance.

7.1 'Exact' matching algorithm

The 'exact' matching algorithm was designed to catch such things as extra, missing or different nonalphabetic characters which were the result of different e-mail clients processing or a sign of recurrent e-mail such as a newsletter. These matches were calculated in the parsing stage when sentences were converted into alphabetic strings, and therefore required a single request to database. 'Exact' matches were used as a baseline comparison for other methods.

7.2 'Near exact' matching algorithm

The goal of 'near exact' matching is to find sentences that differ from given sentence only slightly, in particular to find matches that have either spelling errors or extra/missing letters or words. By trial we have decided that 10% difference between the original sentence and its matches would be considered an acceptable near exact match. 'Near exact' matches are found by applying minedit [14] distance algorithm to strings of alphabetic characters. However, since there are nearly 2 million such strings running minedit distance on all of them would take a significantly long time. In order to reduce the number of strings that need to be compared directly, the following assumptions about given sentence A and its matches are made:

- The difference between string lengths should be within 10%.
- The difference between ASCII sums should be within 10%. An ASCII sum of a sentence is a sum of ASCII values for each character in a sentence.
- The distance between frequency vectors should be within 10%. A frequency vector for a sentence is a vector of 26 dimensions. Each dimension represents the number of times respective letter of the English alphabet appears in a sentence.

A query for all strings that lay within the allowed length and ASCII sum is sent to the database. The distance between frequency vectors is checked on the resultant strings, and only if the above tests are satisfied, the strings are sent to the minedit distance algorithm. All of the thresholds can be adjusted in order to increase/decrease the number of strings that are passed to the minedit distance algorithm.

7.3 'Simple similar' matching algorithm

The goal of the 'simple similar' algorithm is to identify matches based on the number of words that sentences have in common [15]. For a given sentence *A*, the first step in 'simple similar' matching is to retrieve a list of stemmed words associated with *A*. The next step is to count the number of words that each sentence shares with *A*. A Sentence is considered to be a match provided that it shares at least half of the words from the word list of *A*, and is close in length to *A*. Checking the length of a sentence is necessary to make sure that longer sentences do not

match smaller ones. In the implementation, only the top 10 matches are chosen to avoid flooding output.

7.4 ‘WordNet similar’ matching algorithm

'WordNet similar' algorithm is a slight modification to the 'simple similar' algorithm described above. Besides considering words in a given sentence *A* we also consider their synonyms. First the full list of words that is associated with *A* is retrieved, and for each word in *A* its corresponding synonyms, as given by WordNet [16], are found. The resulting list of words is then stemmed and the algorithm proceeds in the same manner as the 'simple similar' search. It is expected that 'WordNet similar' algorithm will produce more matches than any other algorithm. This fact was confirmed in the evaluation section.

Chapter 8: Evaluation of Sentence Matching Algorithms

Evaluation was performed by two human subjects on 100 randomly selected sentences and their respective matches. Evaluators recorded the number of matches returned by 'exact', 'near exact', 'simple similar' and 'WordNet similar' algorithms and ranked each match by assigning a value of “Very good”, “Good” or “Bad”. The following instructions were given out about grade meaning:

- “Very good” match should have same meaning as original sentence.
- “Good” match should be close in meaning but would not qualify to be a “Very good” match in some way. Sentence that has opposing meaning also counts toward good match since it discusses same subject.
- “Bad” match means the match is completely off besides containing same or similar words.

The corresponding evaluation results are detailed below.

8.1 Sentence matching results

Several statistics provided by the evaluators were gathered and summarized. The number of sentences for which a match was found, total number of matches and accuracy shown by each algorithm were gathered.

As expected, 'WordNet similar' algorithm found matches for a larger number of sentences than any other algorithm followed by 'simple similar' despite the fact that for 15 sentences similar algorithms weren't run due to sentence being improper (containing too few words or having too many garbage characters). Table 1 shows

the number of sentences for which at least one match was found by the four algorithms.

Algorithm	Number of Sentences
Exact	26
Near Exact	32
Simple Similar	44
WordNet Similar	57

Table 1: Number of sentences that had a match found by the four algorithms

For the total numbers of matches algorithms performed as expected with 'WordNet similar' returning the largest number of matches, as shown in Table 2.

Algorithm	Number of Matches
Exact	50
Near Exact	58
Simple Similar	244
WordNet Similar	323

Table 2: Total number of matches found by each algorithm

Accuracy results were not surprising either. 'Exact' and 'near exact' algorithms had 100% and near 100% accuracy respectively while 'simple similar' algorithm turned out to be slightly more accurate than 'WordNet similar'. Accuracy results are summarized in Table 3.

	Value		
Algorithm	***Very Good (%)***	***Good (%)***	***Bad (%)***
Exact	100	0	0
Near Exact	100	0	0
Simple Similar	76	18	5
WordNet Similar	73	17	10

Table 3: Accuracy performance of the four algorithms

Overall the results of sentence matching were found to be quite satisfactory. 'WordNet similar' algorithm outperformed other algorithms due to the larger number of matches despite having slightly lesser accuracy.

8.2 Time performance

One of the main goals of this study was to ensure that the algorithms performance was seamless to the user issuing the query. The time performance was measured by running each algorithm over a set of 100 randomly chosen sentences and paragraphs. All experiments were conducted on an Intel® Core™ i7 2.67 GHz workstation with 2 Gbytes of memory. The resulting acquired running times for sentence sets algorithms are summarized in Table 4.

Algorithm	Running Time (seconds)
Exact	0.002 ± 0.001
Near Exact	35 ± 5
Simple Similar	0.2 ± 0.1
WordNet Similar	2 ± 1

Table 4: Average execution time of each of the four algorithms

Both 'exact' matches and 'simple similar' matches are found in a near instant time since they do not require any heavy processing, and are not computationally expensive. 'Exact' matches are determined as simple queries to the database. 'Simple similar' matches query the database and look through results, so if the number of words in the sentence is high, and those words are heavily used, then the performance is slightly reduced. However, its performance was acceptable for real time use, so if a user finds a sentence of interest and asks for its' simple matches, the server will return results seamlessly. The nominally fast performance is most likely attributed to all the pre-processing that has to be performed.

The 'near exact' algorithm takes too long to run to completion for real time use. This is due to the fact that after the first phase of trimming there is still more than 100,000 sentences left. In certain cases performance is almost instant, while in the worst cases it can take up to 40 seconds for the algorithm to run to completion, which is unacceptable for most real time application.

'WordNet similar' matches take variable time since the algorithm has to consider all possible synonyms for all the words in the sentence. One possible

implementation of reducing processing time is by employing concepts from cloud computing, which is described in the next chapter.

Chapter 9: Use of Amazon Cloud for Processing Improvements

Originally we were planning on using single machine to do e-mail parsing and similar sentence identification. We found out that parsing .5 million e-mails can be completed in about a day which is reasonable amount of time for such a large corpus. We also found that identifying similar sentences for a single sentence takes 2 seconds on average for 'WordNet similar' algorithm. If we want to allow user to quickly browse through e-mails and look up similar sentences on demand such a delay on each request is not acceptable. That means that we have to precompute similar sentences. Running 'WordNet similar' algorithm for 2 million of sentences would take several weeks on a single machine, which is unreasonable.

E-mail parsing and similar sentence identification were found to be easily parallelizable so we turned to the Amazon EC2 cloud [17] as a solution.

We identified following stages required for achieving our goal:

- E-mail duplicates removal. At this stage we identified simple and complex duplicates and dealt with them as outlined in section 6.1.
- E-mail payload parsing. This stage is responsible for splitting e-mail payload into paragraphs. During this stage we identified signature paragraphs and included headers.
- Parsing paragraphs into sentences. Sentences were checked for invalid length and non-textual content.
- Sentence parsing. Sentences were split into words, bigrams and trigrams. Only words of length less than 22 characters were allowed. Word-to-

sentences dictionary was created at this stage to facilitate ‘WordNet similar’ algorithm.

- Running ‘WordNet similar’ algorithm on all sentences.

Stages were run using Hadoop [19] implementation of MapReduce [18]. Each stage was run on multiple machines (10-100) that were more powerful than the original machine in our lab. After stages were completed, results were loaded into a MySQL database. At this point all user requests were translated into simple SQL queries, making real-time performance possible. Using cloud computing enabled us to bring running time of all stages for parsing e-mails and finding similar sentences using 'WordNet similarity' algorithm to several hours for the entire Enron dataset.

Chapter 10: Information Flow Identification

In this section we describe in detail how we approached information flow identification in e-mails. E-mail threads are a native way of separating information flows in a corpus thus we attempt to reconstruct them. However we also know that information flows are not confined to threads. Below we present a method for discovering links between e-mails based on sentence similarity rather than on the "Subject:" header.

10.1 Thread reconstruction

Thread reconstruction is a task of discovering parent-child relationships between messages in an e-mail corpus. These relationships are formed whenever an e-mail is being replied to or forwarded. Our algorithm attempts to recover such links by measuring content similarity between e-mails. This is possible because by default replying to an e-mail or forwarding one includes its body in a new message. Most attempts at thread reassembly [2] insist that all e-mails in a thread must share same subject. Our algorithm does not impose such restrictions because it acknowledges that e-mails can have different subjects and yet belong to same thread.

We will explain our algorithm's inner-workings with the aid of an example. Suppose we have messages *M*, *N* and *O* which form the parent-child relationships pictured in Figure 8. Messages *N* and *O* are replies to *M* and thus are *M*'s children. *N* is a sibling of *O*.

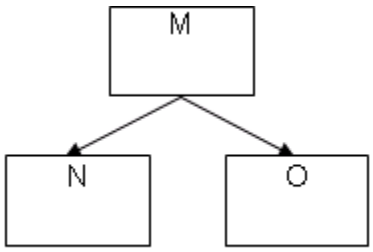

Figure 8: Parent-child relationships between messages M, N and O.

The algorithm is tasked with finding parent for message *N*. The following steps were taken in an attempt to find e-mail *M*:

- Find set of e-mails that share sentences with *N*. For sake of clarity call that set *S*.
- Discard e-mails from *S* that have date field set after *N*. This is done to ensure that we do not mistakenly assign *N's* child as a parent for *N*.
- Rate each e-mail in *S* based on a number of shared sentences with *N*. However we only use a sentence if it has been quoted less times than sentence in *N.* Quotes are usually represented by consecutive bird characters ">>" in the beginning of the line. This exclusion is done to exclude accidental parenting between messages *N* and *O*.
- Update rating for an e-mail in *S* if it shares subject with *N*. We compare subjects that have been stripped of "Fw" and "Re" tags.
- Search *S* for an e-mail with highest rating. If such e-mail exists it is labeled a parent of *N*.

After the algorithm has identified parents for all e-mails in the corpus thread reconstruction is complete.

Several notes about our algorithm:

- Our algorithm makes use of "Subject:" and "Date:" e-mail header fields. It is reasonable to expect these fields to be present in any e-mail dataset. If these fields are absent e-mail thread reconstruction would be a daunting task even for humans.
- Our algorithm relies on "Date:" e-mail header field to order e-mails in the dataset accordingly to the time they were received. However it is not as accurate as we could hope for. Inaccuracies arise due to the sender's mail client filling out this field. Often system time at sender's mail client is incorrect when it stamps an outgoing message, resulting in e-mails from a past or future as was mentioned before. While we can easily detect a "Date:" field that is completely incorrect it is complicated to do so when mail client is off by several hours. Such e-mails will break timeline for messages received. "Received:" is a much more reliable e-mail header field. It tracks timestamps that were filled by the servers when they receive an e-mail. In most situations datasets consists of the emails that were collected either by a single mail server, or several mail servers that are synchronized. Unfortunately we cannot rely on "Received:" e-mail header being present in the dataset as is the case with Enron.
- Our algorithm has an option of imposing stricter rule for identifying parent-child relationship. E-mails will be assigned a parent-child link only if a "To:"

header field of a parent e-mail is equal to the "From:" header field of the child. This option was not used in the evaluation that follows in the next section due to the fact that all e-mails in mailing list archive are addressed to the e-mail address of the mailing list and not an individual user.

- Our algorithm can optionally return not only the best match for a parent but also other top matches that were found. This will allow user to easier fix errors introduced by our algorithm.

10.2 Finding information flow outside of thread scope

Threads are not the sole source of information flows in e-mails. A simple example of information flow that escapes threads is copy-pasting part of the e-mail into new e-mail. In a more complicated scenario user can paraphrase an idea expressed in some e-mail. Automatic discovery of such cases is prone to large number of errors. False positive errors occur whenever e-mails share a lot of words but they do not carry a lot of meaning. False negatives occur when information of interest is not whole e-mail but rather a subset of sentences within an e-mail. In these cases it is hard to find correct links.

We chose to abandon the automatic approach and developed a set of tools that aid the user in discovery of such hidden information flows. These tools assume that the user found an e-mail of interest and wants to find all the links from and to this e-mail. This situation is a typical scenario for e-discovery.

- Browse corpus by looking through the all available “Subject:” fields.
- Search corpus for e-mails with “Subject:” field containing a specific word.

- Search corpus for a specific word.
- Search corpus for bigrams and trigrams that contain word of interest.
- Search corpus for sentences similar to sentence of interest.

By giving the user all these tools we let him drive discovery of information flow that lies outside of conventional threads.

Chapter 11: Thread Reconstruction Algorithm Evaluation

In this section we present evaluation of our thread reconstruction algorithm. Very few e-mails in the Enron dataset retained complete original headers and thus it is not useful to evaluate our work against it. We chose to use an archive of several mailing lists as a dataset for thread reconstruction evaluation. This archive was generously provided by our colleague. All e-mails in the archive have full headers and thus exact thread reconstruction is possible.

We chose to use popular JWZ algorithm as a baseline for e-mail thread reconstruction [20]. This algorithm performs threading based on header fields “In-Reply-To” and “References” as well as “Subject” field.

We ran both algorithms on the new dataset recording parent ids for each e-mail. The results for JWZ algorithm are presented in Table 5.

Parent/No parent	Number of e-mails
No parent found	2538
Parent found but not in archive	295
Parent found	3972
Total	6805

Table 5: Parent-child relationships found by JWZ algorithm

From the results we can see that some of the parent e-mails identified by JWZ algorithm are not present in an archive. These e-mails are called missing e-mails. There are several reasons for presence of missing e-mails in a corpus:

- The corpus is a snapshot of user e-mail directories. An example of such snapshot is the Enron dataset. In this case users were not required to keep all of their e-mails and deleted many of them.
- The corpus is a complete collection of e-mail exchange between users. An example of such dataset is the archive of mailing lists that we are using for this evaluation. However some e-mails to the mailing list were a reply to an e-mail outside of a mailing list creating missing e-mail links.

Results for our algorithm are presented in Table 6. Our algorithm did not perform missing e-mail identification and thus parents were restricted to be present in the corpus.

Parent/No parent	Number of e-mails
No parent found	2292
Parent found	4513
Total	6805

Table 6: Parent-child relationships found by our algorithm

We can see from the results that both algorithms found parents for approximately the same number of e-mails. As a next step we compared whether found parents were matching. For this comparison we counted missing parent e-mails of JWZ algorithms as ‘not found’ entries. Table 7 describes matching results.

Match/No match	Number of e-mails
Algorithms match	4796
Algorithms did not match	2009
Total	6805

Table 7: Comparison between JWZ and our algorithm

We can see that algorithms resulted in a match for 70.5% of all e-mails. A lot of e-mails in the archive included parent e-mails in their body which improved performance of our algorithm.

Below are some reasons as to why the algorithms produced different results.

11.1 Case 1

JWZ algorithm assigns e-mail *M* as a parent for an e-mail *N,* while our algorithm does not find a parent. This occurred 552 times which is 27.5% of all errors.

If *M* is not a parent of *N* then JWZ algorithm erred. This occurs for following reason:

- An e-mail *N* has an “In-Reply-To” header field pointing to *M*, however *N* is not a reply to *M*. Sender pressed reply button because of convenience of finding e-mail address but e-mail client still added header fields. An example of this behaviour is shown in Figure 9 and Figure 10.

If *M* is a parent of *N* then our algorithm erred. It fails to find this link for following reasons:

- Subject field of *N* differs from subject field of *M* and *N* contains no quotations from *M* in body of the message. In these cases our algorithm has no information to use for threading and fails to find such a link.
- *M* has date field set after *N*. In these cases our algorithm does not find a parent due to date restrictions. Difference in date fields comes from desynchronization of some e-mail servers.
- An e-mail client added additional characters such as “=20” to *N*. This results in incorrect parsing of *N* and failure to identify link between *N* and *M*.

```
Hi, I'm totally new to this site and list, and not really qualified to
help work on it, but are you considering transitioning from e-lists to
a message board? Despite a 20+ year history with e-list communities, I
for one would strongly prefer a message-board based community (with
separate forums for each of the current e-lists) for a long laundry
list of reasons that I won't trouble you with since I have no idea if
it's already been planned or rejected.

And while I'm no expert on setting up a messageboard, if it's phpBB
based (i.e. no learning curve for me) I would definitely volunteer to
help with its maintenance.

Just my two-cents before I slip back into the background...

Leslie

Leslie Vincent
CIS Instructor
Delaware Tech
Georgetown, DE 19947
302-856-5400 x2040
```

Figure 9: The e-mail *N* with “In-Reply-To” header field pointing to e-mail *M* shown in Figure 10

```
Mary Gardiner writes:
> Do you see the front page news that currently exists as failing to be up
> to date, out of curiosity? I mainly update it from announce@ and random
> overheard snippets.

Actually I didn't go to the front page very often -- the old web
site was very slow, so I tried to remember the urls I tended to
use, like /content/courses, and go there directly. But when I did
go to the front page, the news always seemed remarkably up-to-date
and I always wondered who was keeping it that way. :-)

--
    ...Akkana
    Check out my book, "Beginning GIMP: From Novice to Professional".
    Now shipping! For more information: http://gimpbook.com
```

Figure 10: The e-mail *M*, parent of *N* shown in Figure 9 as found by JWZ algorithm.

11.2 Case 2

Our algorithm assigns e-mail *M* as a parent for e-mail *N* while JWZ algorithm does not find a parent. This occurred 1093 times which is 54.4% of all errors.

If *M* is a parent for *N* then JWZ algorithm erred. It failed to find parent for following reasons:

- *N* is a reply to or a forward of *M. N* has quotes from M. However threading headers are missing thus making it impossible for JWZ algorithm to correctly identify a parent. An example of this behaviour is shown in Figure 11 and Figure 12.
- *N* has several e-mails in “References” header and JWZ incorrectly picks one as a parent.

If *M* is not a parent for *N* then our algorithm erred. Reason for our algorithm failing is:

- *N* shares one or more sentences with *M* that do not have any significant meaning. For example if *N* and *M* share signature paragraph that was not removed.

```
On Thu, May 31, 2007 at 07:36:15AM +0200, Isabelle Hurbain wrote:
> And then, when you explain that no, you can't, your considered either
> as a bad or as an incompetent person in your field. Nice indeed.
I would consider someone who made *that* assumption to be both stupid
and arrogant.  Who are *they* to declare that you're incompetent in a
field that *they* know nothing about?  Geeze, that's like declaring an
aerospace engineer incompetant because they aren't a motor mechanic.
(eyeroll)

Kathryn Andersen
--
 _--_|\      | Kathryn Andersen  <http://www.katspace.com>
/      \     |
\_.--.*/     | GenFicCrit mailing list <http://www.katspace.com/gen_fic_crit/>
    v        |
-------------| Melbourne -> Victoria -> Australia -> Southern Hemisphere
Maranatha!   |   -> Earth -> Sol -> Milky Way Galaxy -> Universe
```

Figure 11: The e-mail *N* with and empty "In-Reply-To" header field, but with quotations from e-mail *M* shown in Figure 12.

```
> - "You know all about computers, can you fix my PC for me?"
> I've managed to train my family out of that by repeated "No, I don't
> use MS-Windows, I can't help you."

Many people do not have any idea of what is computer science. "I'm a
Ph.D. Student in CS" got me more than once "oh, then maybe you can fix
my printer" and other stuff of this kind... Or, because my thesis
subject had to do with H.264 encoding, the assumption that I know every
tiny timsy option of every piece of software ever written to encode or
read a DiVX.
And then, when you explain that no, you can't, your considered either
as a bad or as an incompetent person in your field. Nice indeed.

Isa
```

Figure 12: The e-mail *M*, parent of *N* shown in Figure 11 as found by our algorithm.

11.3 Case 3

Both algorithms find parents for an e-mail but they are different. This occurred 364 times which is 18% of all errors. Reasons behind it are:

- Header fields are present but they are incorrect. In this case our algorithm finds correct parent. An example of this behaviour is shown in Figure 13, Figure 14 and Figure 15. From these figures we can see that e-mail *M1* found by our algorithm is a better match for being parent of *N* than *M2* found by JWZ algorithm.
- Our algorithm finds too many sentences in common with e-mail that is not parent.
- Our algorithm incorrectly picks e-mail sibling for its parent.
- JWZ algorithm picks wrong message id from headers when many headers are present.

```
Dancer Vesperman wrote:
> Website is now moved.

I noticed when the site went down. Then it came back and was suddenly oh
so very very fast!  Good work! Yay Val, Dancer, and everyone else who
made this happen!

Jen
```

Figure 13: The e-mail *N* with "In-Reply-To" header field pointing to email *M1* shown in Figure 14, and with quotations from e-mail *M2* shown in Figure 15

```
Here are the photos of our new, on the net LC server!

http://infohost.nmt.edu/~val/pix/lc_server.jpg

This is one with Scott Kveton (the guy who runs the Open Source Lab
and who helped us put all this together) posing next to it because I
made him:

http://infohost.nmt.edu/~val/pix/lc_server_scott.jpg

kernel.org is hosted in the rack to the left of our server. :)

And I can even log in!

val@goober:~$ ssh root@linuxchix.osuosl.org
Last login: Tue May  9 12:19:37 2006 from c-24-6-188-92.hsd1.ca.comcast.net
linuxchix:~# uname -a
Linux linuxchix 2.4.27-2-386 #1 Wed Aug 17 09:33:35 UTC 2005 i686 GNU/Linux
linuxchix:~# cat /etc/issue
Debian GNU/Linux 3.1 \n \l

linuxchix:~#
```

Figure 14: The e-mail *M1*, parent of *N* shown in Figure 13 as found by JWZ algorithm.

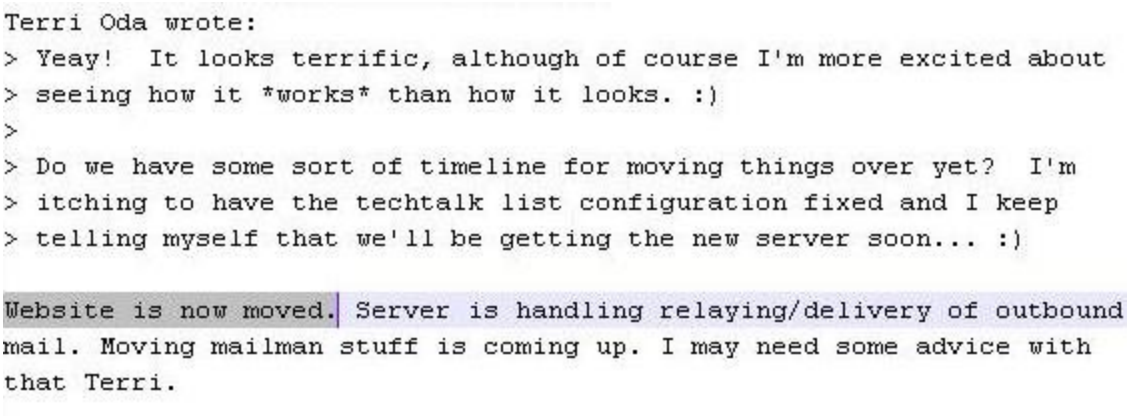

Figure 15: The e-mail *M2*, parent of *N* shown in Figure 13 as found by our algorithm.

Chapter 12: Suggested Improvements and Future Works

In this section we present future improvements that will aid thread reconstruction and identification of similar sentences.

E-mail parsing is the most important stage of our project. Quality of thread reconstruction and similar sentence identification depends on quality of sentences produced by parsing stage. Particular parsing stages that should be improved are:

- Signature paragraphs identification.
- Hidden header identification.
- Identification of parts of e-mails that contain non sentenced text such as tables, programming code, hyperlinks html messages and others.

We are looking into several ways to improve our e-mail thread reconstruction algorithm:

- Missing e-mails is one of the sources of errors in our algorithm as was shown in evaluation section. We are working on identifying these emails to improve accuracy of the thread reconstruction algorithm.
- We are planning to add support in our algorithm for using e-mail header fields that could be present in the dataset. One of such fields is “Received:”. Benefits for using this field were discussed before.

As we have shown in previous section, JWZ algorithm produces errors even in a corpus that retains all original e-mail header data. We believe that by supplementing JWZ algorithm with our e-mail thread reconstruction algorithm we will achieve higher

accuracy in identifying parent-child relationships. We are looking into combining these algorithms and evaluating the resulting algorithm.

Chapter 13: Conclusions

Identifying information flow through non-transcribed means of communication, such as telephone conversations or video chats is an extremely difficult task. In this work we focused on a slightly simpler, yet still challenging undertaking of identifying information flow in e-mail conversations. We presented an algorithm for thread reconstruction based on text matching as well as a set of tools that aid users in discovery of hidden information flows.

Prior to applying any information discovery algorithms to an e-mail corpus, a significant amount of effort was put into pre-processing the dataset. The pre-processing step included the removal of duplicate e-mails, discarding headers and signatures, and parsing the e-mails into corresponding sentences and words. This is particularly important because a properly pre-processed dataset will significantly increase the accuracy of thread reconstruction algorithm.

Following the preprocessing we applied several algorithms for identifying similar sentences in e-mails. The algorithms included in this study were 'Exact', 'Near Exact', 'Simple Similar' and 'WordNet Similar'. Each algorithm has its advantages and disadvantages but during the evaluation process we found that the 'WordNet Similar' algorithm gave us the best results when we consider both accuracy and coverage.

In order to increase the computational efficiency of the 'WordNet Similar' matching algorithm, and make it applicable for real-time use, we had to resort to cloud computing. This allowed us to separate the different steps in the algorithm among several different processors, and run the steps in parallel. This allowed us to

significantly increase the computational efficiency of not only the sentence matching, but also the pre-processing steps.

We presented an algorithm for e-mail thread reconstructions based on identifying parent-child relationships between pairs of e-mails. We showed that it performs successfully in complete absence of e-mail header fields necessary for threading. However even if such header fields are present in the dataset our algorithm helps identify inconsistencies that arise due to invalid headers. We also presented a set of tools that aid the user in discovery information flows that lie outside thread structure.

References

[1] Jianqiang Shen, Erin Fitzhenry, and Thomas G. Dietterich. Discovering frequent work procedures from resource connections. In *Proceedings of the 14th international conference on Intelligent user interfaces*, pages 277-286, 2009.

[2] Jen-yuan Yeh. Email thread reassembly using similarity matching. In *Proceedings of CEAS*, 2006.

[3] Thomas Karagiannis and Milan Vojnovic. Email Information Flow in Large-Scale Enterprises. Microsoft Research, 2008.

[4] Marcus, Richard L. E-Discovery & (and) Beyond: Toward Brave New World or 1984. 25 Rev. Litig. 633, 2006.

[5] Carey S. Meyer and Kari L. Wraspir. E-Discovery: Preparing Clients for (and Protecting Them against) Discovery in the Electronic Information Age. 26 Wm. Mitchell L. Rev. 939, 2000.

[6] Bryan Klimt and Yiming Yang. Introducing the Enron Corpus, 2004.

[7] Ravi S. Sandhu and Pierangela Samarati. Access control: principle and practice, 1994.

[8] Skupsky, D. S. Establishing retention periods for electronic records. Records Management Quaterly 27, pages 40, 43-43, 49, 1993.

[9] David D. Lewis, Kimberly A. Knowles. Threading electronic mail: A preliminary study, 1997.

[10] Tibor Kiss and Jan Strunk. Unsupervised Multilingual Sentence Boundary Detection. Computational Linguistics 32, pages 485-525, 2006.

[11] Porter M. F. An algorithm for suffix stripping. Program, 14, pages 130-137, 1980.

[12] Edward Loper and Steven Bird. NLTK: the Natural Language Toolkit. In *Proceedings of the ACL-02 Workshop on Effective tools and methodologies for teaching natural language processing and computational linguistics - Volume 1*, 2002.

[13] George Spache. A New Readability Formula for Primary-Grade Reading Materials. The Elementary School Journal. Vol. 53, No. 7, pages 410-413, 1953.

[14] Levenshtein, V. I. Binary codes capable of correcting deletions, insertions and reversals. Soviet Physics - Doklady 10, pages 707-710, 1966.

[15] Giuseppe Carenini, Raymond T. Ng, and Xiaodong Zhou. Summarizing email conversations with clue words. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 91–100, 2007.

[16] George A. Miller, Richard Beckwith, Christiane Fellbaum, Derek Gross, and Katherine J. Miller. Introduction to WordNet: An On-line Lexical Database, 1990.

[17] Amazon Web Services. http://aws.amazon.com/.

[18] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6*, pages 107-113, 2004.

[19] Hadoop. http://hadoop.apache.org/

[20] Jamie Zawinski. Message Threading. http://www.jwz.org/doc/threading.html.

[21] RFC 5322. http://tools.ietf.org/html/rfc5322.

[22] Ross Eckler. Making the Alphabet Dance: Recreational Wordplay. 1996.