Predicting Email Response using Mined Data

Christian Bird
University of California, Davis
bird@cs.ucdavis.edu

ABSTRACT

Mailing lists are the primary medium of communication in open source projects. For some projects the sheer volume of emails on the mailing lists becomes unmanageable and messages may begin to be ignored. This can have a number of negative effects on an open source project. We present a way to predict who is most likely to respond to an email, thus providing the potential of giving mailing list developers only the mail that they are interested in. We use mined source code repository commit history, social network data, and a natural language search and indexing technique called latent semantic analysis on the bodies of past emails from the Apache server project to make these predictions using a neural network.

1. INTRODUCTION

Mailing lists are the primary medium of communication in open source software (OSS) projects. As these projects grow in size and popularity, more and more developers and users participate on the mailing lists. This leads to a high volume of emails that can become unmanageable. As an example, although there are many mailing lists for subsystems within the Linux kernel, the main Linux kernel mailing list receives between 200 and 300 messages per day[1]. For projects like these, the chance that a contributor may miss or not respond to a relevant message rises as the number of emails per day increases. Given the fact that the majority open-source contributors have full-time jobs in addition to their project work, most do not have time to read through large numbers of mailing list messages. Even if they did, people do not want to read messages that are not relevant to them. Often this judgement of relevancy is based solely on subject line and author possibly causing important messages to be missed. This can have a number of negative effects. If list newcomers do not get responses to their emails, they may cease to participate. Possible contributors who submit bug fixes and new features via email will likely be deterred if their patches are ignored.

Due to the open nature of OSS projects, a large amount of historical data is freely available via the web. Most projects keep archives of their mailing lists which contain information including the author, date, text, and reply information for each email sent on the mailing list. In addition, nearly all projects make their source code repositories freely available with read access. These repositories contain the source code for all project files as well as a history of the changes made by developers to these files.

In response to the problem of mailing list volume, we present a method for predicting likely responders to mailing list messages by taking advantage of this historical information. Specifically, we use source code repository commit history, past email recipient information, and natural language processing of email text as data to train an artificial neural network (ANN). The resultant neural network is able to predict the n most likely responders to any email message given only the body and author of the message.

2. RELATED WORK

Hipikat is a tool created by Cubranic, Murphy, Singer, and Booth [3] which uses mined data such as source code repository history, mailing list and newsgroup archives, and web documents to create a project memory. This project memory was used to aid new comers to a project in learn about the software. It was successfully tested on the 2.1 release of the Eclipse Java IDE, a large open source project.

Zimmerman, Weßgerber, Diehl, and Zeller [9] used data mining techniques on source code repositories to predict when how changes to one file might affect others. Users of their tool, *ROSE* would make a change to some file in a repository and prior to checking the change in, *ROSE* would attempt to determine if other files needed correlated modifications.

Ying, Murphy, Ng, and Chu-Carroll [8] used source code repository history to create rule associations for files in the repository. Given a set of source code modifications, they were able to predict with reasonable accuracy, what additional related changes would occur in the future.

Cao, Lioa, and Li [1] used principal component analysis (similar to latent semantic analysis in this paper) and neural networks to accurately classify incoming mail as spam. They used features of the text of the email as well as non-textual features such as domain type of the email sender, header length, date and attached documents as inputs to their artificial neural network.

In the next section we present our methods for collecting and processing training and validation data. In section 4 we present the setup of the learning method and discuss parameters. Our results are presented in section 5 with conclusions in section 6 and future work in section 7.

[1]according to wikipedia `http://en.wikipedia.org/wiki/Linux_kernel_mailing_list`

3. DATA COLLECTION

In order to use data to train an ANN, we first had to data mine information from an open source project. We decided to use information from mailing list archives and source code repositories. We picked to use data from the Apache web server project because it is a well known project with a moderate number of developers and mailing list activity and has email and source code repository history dating back to 1995. In addition, the Apache server is written in C which allowed us to use existing source code analysis tools to extract information from the files in the repository.

We decided to use data from four sources as input to our ANN. We used the date of the email as the first piece of data. For the second piece of data, we examined the body of each email to see what files and functions it referenced which exist in the source code repository. For each developer the total number of commits to those files and functions was extracted from the repository and used as input to the ANN. As the third source of data, we examined how many emails had been sent from each mailing list user to the author of the email message over the six months leading up to the message. The fourth piece of data was a vector of natural language "concepts" (similar to a vector of word frequencies) from analysis of the email using a process known as latent semantic analysis (LSA). LSA is discussed in more detail in section 3.2.

3.1 Datamining

In order to use the data to train and test an ANN, we first had to gather it from the web and process it into a form understandable by a neural net. In order to organize our data and be able to retrieve it efficiently, we stored mined Apache data in a PostgreSQL database[2]. The data mining process took a number of steps.

The first step was to locate, download, and parse the email archives for the Apache project. We wrote a number of python scripts to automatically download the archives from Apache's site and took advantage of some email header parsing libraries to extract the author, message ID, reference ID[3] , date, and body of each email and store that information into our database[4]. The Apache archives contains over 100,000 messages dating back to February of 1995. We were able to accurately parse 101,233 of the messages. Most of the messages that we could not parse had incomplete or corrupted header information. In some cases, emails that contained whole emails inside of them (which can occur when forwarding messages) caused our parser to fail as well.

One problem that we had to overcome was the fact that one person can have multiple email addresses. We define an identity as a single person participating on a mailing list through one or more email addresses. As an example, Andrew Ford used both `andrew@icarus.demon.co.uk` and `a.ford@ford-mason.co.uk` to post messages on Apache's mailing list. By attributing each message to an identity instead of just an email address, we were able shrink the number of entities participating on a mailing list and increase the information available for each identity. In many cases, mapping email addresses to identities was a simple process of extracting the name from the email header. For headers that didn't contain names, Alex Gourley came up with a similarity algorithm for determining whether two email addresses were for the same person.

Because our goal was to predict responders to messages, only messages that were replied to could be used as data points. A message A can be classified as a reply to message B if the reference ID of A is the same as the message ID of B. A mailing list thread, which can be thought of as a conversation among participants, can be re-created by finding a message with no reference ID (the original message of a thread) and finding all messages with reference IDs matching the original's message ID and then finding messages with reference IDs matching those messages' message IDs, etc. 50,477 Of the messages that we parsed and stored into our database had replies.

In addition to email data, we also took advantage of source code repository history. The Apache server project used the Concurrent Versions System (CVS)[5] as its source code repository system during the time period of our analysis. Fortunately, this repository is publicly available with read-only access. We were able to collect commit histories for the Apache developers for each file in the repository. We also used the LXR Cross Referencer[6] to extract the names of the functions from each file in the Apache source tree. This information was useful because it allowed us to determine what files and functions were mentioned in the body of each email. Our hypothesis is that developers who have a history of committing to the files mentioned in an email message have a higher likelihood of responding than developers who do not.

3.2 Latent Semantic Analysis

We believe that the body of an email message strongly affects who responds to it. The hope is that the topic space of the messages divides the participants into groups. That is, if a responder replies to a message asking about threads in Apache, he is likely to reply to other messages that mention threads or topics related to threads in the future. If he has never responded to a message regarding installation on Windows, then he is not likely to answer a question about that topic. In order to take advantage of this belief, we used a technique called latent semantic analysis (LSA) to extract information from the bodies of emails.

Deerwester, Dumais, Landauer, Furnas, and Harshman proposed LSA as a searching and indexing technique in 1990 [4]. The idea behind LSA is that words that appear in similar contexts are related in some way. By using latent semantic analysis, it is possible for two documents to be related to each other due to having words that are related even if the documents do not share any common words. Experiments have shown that LSA performs roughly 30% better than simple

[2] `http://www.postgresql.org`

[3] The message ID uniquely identifies a message. The reference ID corresponds to the message ID that an email is in response to.

[4] A large part of this step was accomplished by Alex Gourley (a student in this class) over last summer as part of a larger project.

[5] `http://www.nongnu.org/cvs/`

[6] `http://sourceforge.net/projects/lxr`

word matching techniques in document retrieval [5]. This in part due to the fact that LSA has been shown to overcome two issues in text retrieval, *synonymy* and *polysemy*. *Synonymy* describes the fact that one object, concept, or idea can be referred to by many different names (e.g. concurrency, synchronization, scheduling). *Polysemy* refers to idea that one word can have multiple meanings. For example, architecture can refer to the high level design of an application or to the type of microprocessor in a computer. Standard word-matching techniques may not relate documents whose words have *synonymy*, but may relate documents that have words in common with *polysemy*, both leading to false positives.

Latent semantic analysis works in the following way.

First a term-document matrix, X, is created that created from the corpus of text. Each row in the matrix represents a word that occurs in the corpus and each column represents a document (in our case, an email message). Each cell_{ij} represents the frequency of term_i in document_j. Then LSA applies singular value decomposition (SVD), a form of factorization, to the matrix. This step factors the matrix, X into the product of three other matrices, T, S, and D.

$$X = TSD^T$$

T describes the original terms as vectors of derived orthogonal factor values. D describes the original document vectors in the same way. S is a square matrix with dimension equal to the smaller dimension of the terms and documents. It contains descending scaling values along its diagonal and zeros everywhere else. When these three matrices are multiplied together, the original matrix is produced. We can reduce the dimensionality of these matrices by taking only the first n rows and columns from S to obtain S' and then taking only the first n columns from T and D to obtain T' and D' respectively. Now, even though we have removed rows and columns, $X' = T'S'D'^T$ still closely approximates X. One premise here is that by approximating X using the highest value dimensions, X' contains less noise. Each column in D'^T does not represent a term vector anymore. Rather it can be thought of as a combination of orthogonal semantic "concepts". By shrinking the dimensionality (often from thousands or tens of thousands down to hundreds), LSA essentially shrinks the concept space of the documents from terms to combinations of terms. Two documents can be compared by taking the cosine of their respective vectors in $S'D'^T$. Documents with high relevance will have vectors that are close in the reduced dimensionality concept space. To compare a document not in the original corpus, one simply needs to construct the term vector for the document as done in the beginning and multiply it by S. The resulting vector can then be compared to documents already in the corpus by taking the cosine between the two.

Susan Dumais [5] showed that better results can be obtained by making each cell in the original matrix X a product of the term's global weight in the corpus $G(i)$ and its local weight in the document $L(i,j)$ as follows.

$$G(i) = 1 + \frac{1}{\log(N)} \sum_{j=1}^{N} p_{ij} \log(p_{ij})$$

$$L(i,j) = \log(1 + tf_{ij})$$

$$p_{ij} = \frac{tf_{ij}}{d_i}$$

N is the number of documents in the corpus, d_i is the number of documents containing the term i, and tf_{ij} is the number of times term i appears in document j. This has the affect of giving more weight to words that appear in few documents and making documents with the different frequencies of the same word appear to be more similar.

In our system, we have implemented LSA as described above. Prior to the LSA step, we remove signatures and reply lines[7] from the email messages using Cohen and Carvalho's Jangada Java library [2]. We also use a stoplist of approximately 2000 terms to remove common words and use Martin Porter's widely used stemming algorithm [7] to stem the terms. Lastly, to avoid random sequences of letters and other oddities, we include only words that occur at least twice in the email corpus and are at least three letters long. Function and file names in email bodies were not modified or filtered in any way. Despite this filtering, we still indexed using approximately 33,000 unique terms. By using LSA, we were able to shrink the dimensionality of term vectors for documents from this unmanageable level down to 200. When testing the results of LSA on our corpus of messages, the results were favorable. One search for documents relevant to the query "win32 threads" returned an email message discussing multi-threading and POSIX as the second hit. Note that "multi-threading" and "threads" were not stemmed to the same word so the message did not contain any terms in the query, but still was highly related.

4. LEARNING SETUP

The process of selecting which machine learning method to use was largely a process of elimination. Because of the format and the large amount of diverse data available to us, we needed to use a learning method that could handle a high number of continuous attributes. By the nature of the problem we are trying to solve, we expect that our data contains a large amount of noise so we can only expect to get decent performance out of learning methods that are not susceptible to noisy data. Although it would be nice to understand the concepts learned by the method, it is not critical. According to Tom Mitchell [6], one of the best candidates given this criteria is an artificial neural network. In the following sections we describe exactly what the training, validation, and testing instances contained as well as what parameters we used in the learning process.

4.1 Instances

The data contained in each training, validation, and testing instance came from distinct sets of information. We believe that the date that an email is sent has a strong correlation to who responds to it. The rationale behind this is that over time, mailing list participants may come and go. If someone is active during a specific time period, then they will have a higher likelihood of responding to emails in the same time period than a participant who has not posted a reply in years.

[7]Reply lines are the lines in a response that are included from the original message.

Unfortunately, there is no way to tell our neural network this hypothesis. We can however, provide the date of an email as a continuous value and let the network attempt to learn the relationship. We compute the timestamp for a message e_i as

$$\text{timestamp}(e_i) = (y_i - 1995) * 365 + m_i * 30 + d_i$$

where y_i is the year that the email was sent, m_i is the month, and d_i is the day in the month. Since the mail archive starts in 1995, timestamp(e_i) gives roughly the number of days since the archive began.

We also believe that there is strong relationship between past mailing list correspondence and future replies. In an OSS mailing list, participants do not send messages directly to other participants. Rather they send messages to the list, which are then forwarded to all participants subscribed at that time. For our purposes we say that identity I_a responded to identity I_b if I_b posted a message to the mailing list and I_a posted a reply to that message. We can thus use the information in our database to determine how many messages a given identity has received from other identities. There are 1106 distinct identities on the Apache mailing list. To be fair, we only include messages sent prior to the message being predicted. We have decided to only include correspondence in the six months leading up to the month that the message was sent. The correspondence history for e_i is computed as

$$\text{history}(e_i) = [\text{msgs}(a_i, I_1, m_i), \ldots, \text{msgs}(a_i, I_n, m_i)]$$

where a_i is the author of e_i, $I_1, \ldots, I_n$ are each of the identities that have posted responses on the mailing list, and m_i is the month that the message is sent in. Msgs(a, I, m) is the number of messages sent from I to a in the six months prior to m.

Many of the mailing list participants on an OSS mailing list are also developers, and nearly all developers are on the mailing list. Thus we have information regarding the commit history for many of the identities. The correlation of CVS username to email identity was done by visual inspection and added by hand to our database. For the period of time examined, there were 53 CVS users. For each email, we extract all files and functions that are recognized from the source code repository. For each function mentioned, we can determine what file the function is in. From this we create a set of files referenced by a message. We then calculate the number of commits made to the set of files for each CVS user. F is a function that maps a function name, f to the file containing f in the repository.

$$\text{files}(e_i) = \{\text{each file in } e_i\} \cup \{F(f) \text{ for each function } f \text{ in } e_i\}$$

$$\text{cvs}(D, f) = \text{ number of commits to the file } f \text{ by } D$$

$$\text{msgcvs}(e_i, D) = \sum_{f \in \text{files}(e_i)} \text{cvs}(D, f)$$

$$\text{commits}(e_i) = [\text{msgcvs}(e_i, D_1), \ldots, \text{msgcvs}(e_i, D_n)]$$

Commits is a list of the commits made by developers $D_1, \ldots, D_n$ to the files and functions mentioned in e_i.

Lastly, we want to include the LSA data for the body of each email message that used in the neural network. This is because this data represents the vector of the message in our reduced dimensionality concept space. Our hope is that each participant tends to respond to messages that LSA maps to certain regions in the concept space. If X_i is the high dimensionality term vector for an email body, the reduced dimensionality "concept" vector can be computed as

$$\text{LSA}(e_i) = SX_i^T$$

In this case, LSA(e_i) is a vector of size 200. Each element of the vector is an attribute for e_i.

The input instance data for e_i that is used to train, validate, or test the neural network is a list of values.

$$\text{IN}(i) = [\text{timestamp}(e_i), \text{history}(e_i), \text{commit}(e_i), \text{LSA}(e_i)]$$

When including all of this information, each instance has 1360 attributes.

For the output data, we create a list of values corresponding to mailing list identities. A value of 1 is used if the identity responded to the message and -1 is used otherwise. Note that it is possible for more than one identity to reply to a message. Therefore, although most of the values in this list will be -1, there will be at least one and maybe more 1 values.

$$\text{OUT}(i) = [\text{response}(I_1, e_i), \ldots, \text{response}(I_n, e_i)]$$

$$\text{response}(I, e_i) = \begin{cases} 1 & \text{if } I \text{ replied to } e_i \\ -1 & \text{if } I \text{ did not reply to } e_i \end{cases}$$

4.2 Neural Network

There are a number of parameters that can be tuned when creating a neural network. These include the number of input units, output units, hidden layers, neurons per hidden layer, and the squashing function use for each neuron. While there have been many attempts to estimate what values to use for these parameters based on number of inputs, outputs, etc., the best recommendations are simply rules of thumb and educated guesses at best. After some trial and error with subsets of data, we decided to use a neural network with one input per instance attribute, one hidden layer consisting of 200 neurons, and one output per mailing list identity. We used the standard sigmoid function in the hidden and output neurons. A visual depiction of the network is shown in Figure 1.

Normally, when an ANN is used to classify instances, there is one output node per classification. When an instance is given to the network, the instance is classified according to the output node with the highest value. In our case, each output node represents a participant on the mailing list who may respond to an email instance. Due to the nature of the problem, assessing the network based on its ability to predict the exact responder to an email results in very low accuracy levels. To deal with this issue, instead of just one, we take the n highest output nodes and say that the neural network has predicted correctly if one of the n nodes corresponds to the actual responder. This is a reasonable approach as even the ability to pick the top ten or twenty most likely responders out of over a thousand still dramatically reduces the number of mailing list messages that a participant would need to read. For our testing, we chose n to be 20. Although this affects how accurate we say the neural network is, it does not affect the actual

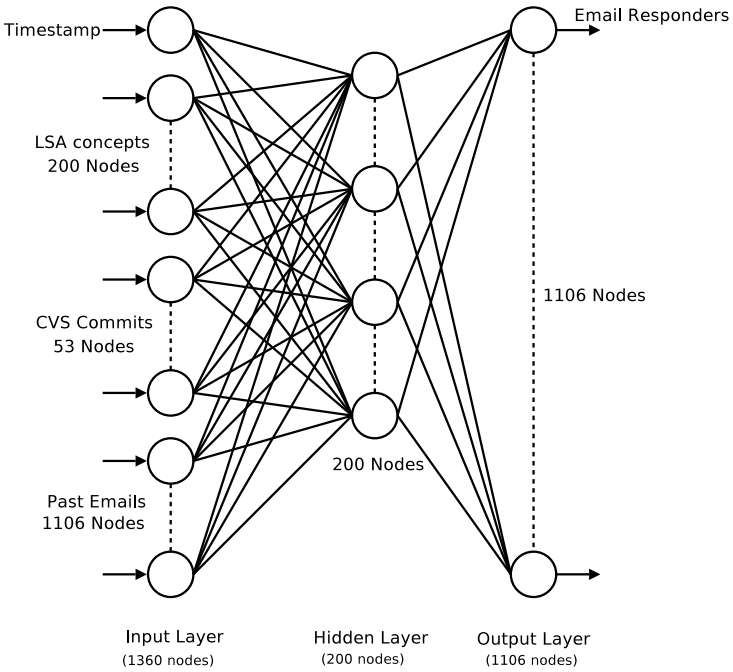

Figure 1: Setup of neural network

backpropagation algorithm. The error for each instance still comes directly from the output data for that instance without modification.

We used the *Fast Artificial Neural Network Library* (fann) to create, train, and test the neural networks. This is a mature open source neural network library written in C. Since the fann's source code is available, we were able to make modifications to the library in order to gather desired data from the network.

5. RESULTS

For our testing, we had approximately 50,000 email instances. Because of the large amount of data, we extracted 10% of the instances at random to be used as validation data. In our different experiments, the same instances were extracted from each run. In order to see which sets of input attributes were most important, we trained six our neural networks, each using different sets of attributes. We first trained and validated our neural network using all available attributes. Then we tested training the network by removing each of the attribute sets in turn. Thus we have a network that does not examine the CVS commit history, one that does not use the timestamp, etc. We also created a data set that included the reply-lines of the emails in the text analysis to see if there was any noticeable increase or decrease in classification accuracy.

Because of the large size of the neural network and the high number of data instances, training out neural networks took quite a long time. We trained and tested each neural net on its own machine. When running on a 3 Ghz Pentium 4 with 1 GB of RAM, it took over 6 hours to run only 100 epochs. The results of our experiments are found in Figures 2 through 7.

As you can see from the graphs, the ANNs reached 100% accuracy on the training data very quickly, often near or just after 20 epochs. In most cases, the accuracy on the valida-

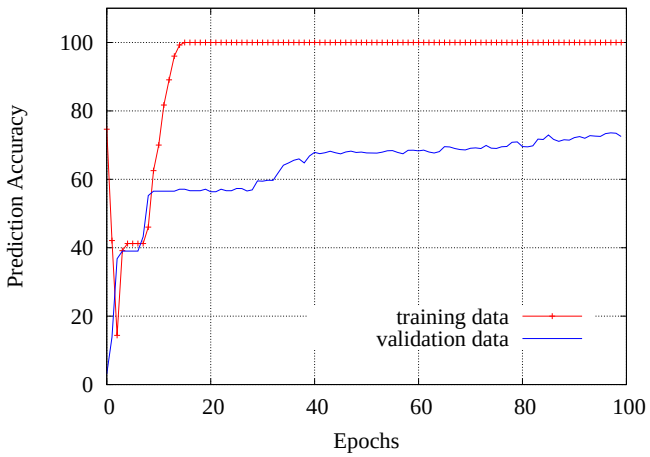

Figure 2: Results when using all available data

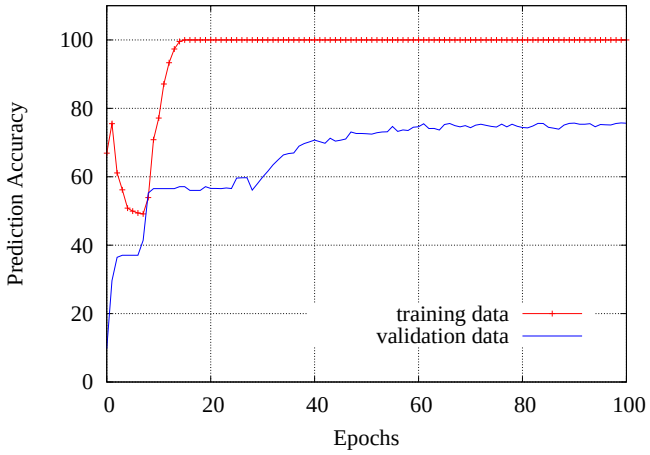

Figure 3: Results when commit history is not used

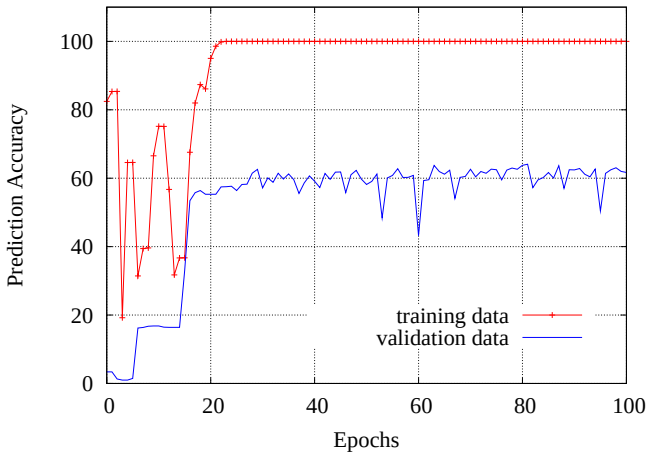

Figure 4: Results when past email correspondence is not used

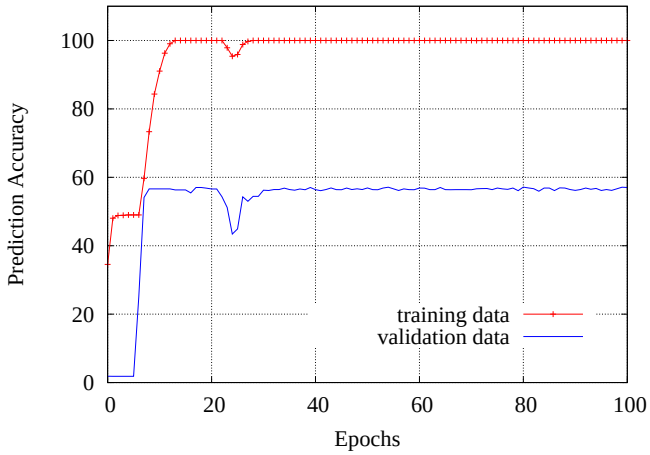

Figure 5: Results when email timestamp is not used

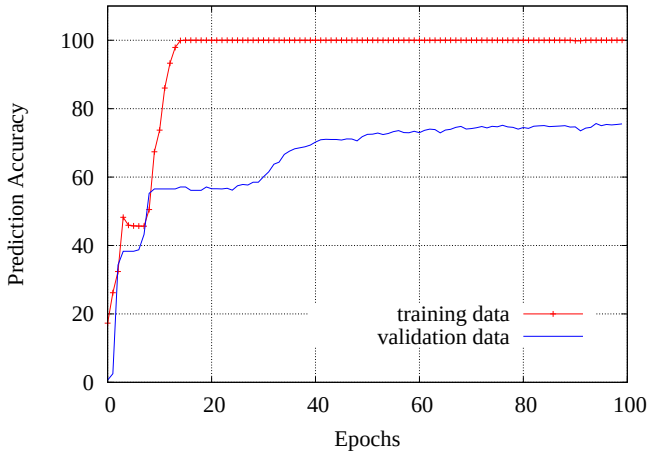

Figure 6: Results when text analysis (LSA) is not used

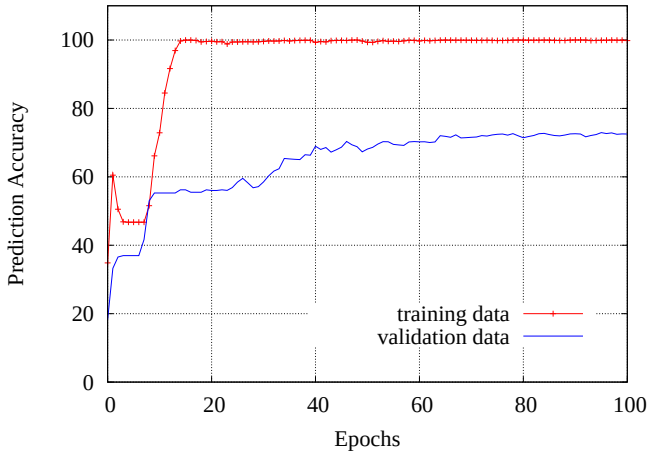

Figure 7: Results when including the reply lines in the text analysis

Figure 8: Results when using only timestamps and past email correspondence is used

tion data quickly rises to a level around 50% and then slowly grows over time. By the 100th epoch, the accuracy on the validation data had mostly stabilized and in some cases even began to decrease slightly indicating overfitting. Surprisingly, the best accuracy was achieved by the neural network that did not include CVS commit history (75.6%). The ANN that did not have access to LSA data did almost as well, achieving an accuracy of 75.5%, while the results of using all available data yielded 72.5%. The inclusion of reply lines in the text analysis did not affect accuracy much, resulting in 72.9% prediction accuracy. It would appear that the timestamp of an email and the past email correspondence are the most important attributes when predicting email response. When trained without email timestamps, the network only correctly predicted 57.1% of the validation instances. The network that had access to everything but the past email correspondence had 65.0% accuracy.

While we used randomly selected instances from the data for validation, the real use of the neural network is to predict responses to messages in the future. To this end, we kept out the last month (July, 2005) when testing and validating each neural network for use after the training was complete. The results of this testing indicate even stronger that the timestamp and email correspondence history are the most important attributes. The ANNs that did not have access to timestamp and email correspondence history had accuracies of 29.8% and 33.5% respectively, while the ANNs that did not look at LSA text analysis and commit history had prediction rates of 58.4% and 57.1%. Table 1 lists the results.

Since these tests seem to indicate that the email timestamp and correspondence history are the most important sets of attributes, we decided to run another test where those were the only attributes available. The validation accuracy was similar to the tests for leaving out commit data and leaving out LSA data. When used to predict responses for future months, however, the trained neural network did far worse. While the other two trained ANNs correctly predicted responses in 57.1% and 58.4% of the cases, the network that used only timestamps and correspondence only achieved 49.1% accu-

Data Description	Training Accuracy (%)	Validation Accuracy (%)	Future Month Accuracy (%)
All data	100.0	72.5	36.0
No commit history	100.0	75.6	57.1
No LSA information	100.0	75.5	58.4
No correspondence history	100.0	65.0	33.5
No timestamp	100.0	57.1	29.8
With reply lines	100.0	72.9	42.0
Only timestamp and correspondence	100.0	75.7	49.1

Table 1: Results of training neural networks with different attribute sets

racy. This indicates that although not as important as other attributes, the commit history and LSA text analysis do help somewhat in the prediction process. The results of this test are shown in Figure 8.

6. CONCLUSIONS

From the results of the experiment we can conclude that past email history and the date that an email was sent are the most important attributes to use when predicting email response. When analysis of the body of the email and CVS commit history is used, the prediction accuracy of the learned ANN actually decreases. This indicates that there is not a strong relationship between how often someone has committed to files in a source code repository and how likely they are to respond to an email mentioning those files. The fact that latent semantic analysis did not increase accuracy would seem to imply that participants do not respond only to emails that fall within certain regions of the concept-space. Rather, any given participant appears to respond to emails about many different subjects. We have a few guesses as to why this might be.

Since this is a developer mailing list, most participants are developers or contributors in some way to the project. Although very popular, Apache is not a very large project, composed of roughly 300 files. Given this, it is likely that a large number of developers have knowledge about many areas, and in some cases, perhaps all of the Apache code base. These participants are qualified (or at least feel qualified) to respond to messages about virtually any topic related to Apache. This means that the concept space spanned by messages responded to by any of these participants will be very large and attempting to discern a pattern or trend will be in vain.

After some simple analysis of the email history we found that on average, only 50 unique responders post replies to messages on the Apache mailing list in any given month. The maximum number of responders is 90 (August 2002). This may explain why the timestamp data is so important to response prediction. If the neural network has the timestamp for a message then it can essentially shrink the number of likely responders from 1106 down to an average of 50. Although we cannot force the network to learn this concept, the data indicates that is has captured the relationship between date of message and possible responders. After making this observation, the accuracy of the ANN's prediction is much less impressive. Predicting the 20 most likely responders out of the 50 who were active in the month of the message does not seem as difficult a task. Random guessing with no information other than the timestamp would lead to 40% prediction accuracy! The graph of the number of unique responders active per month is shown in Figure 9.

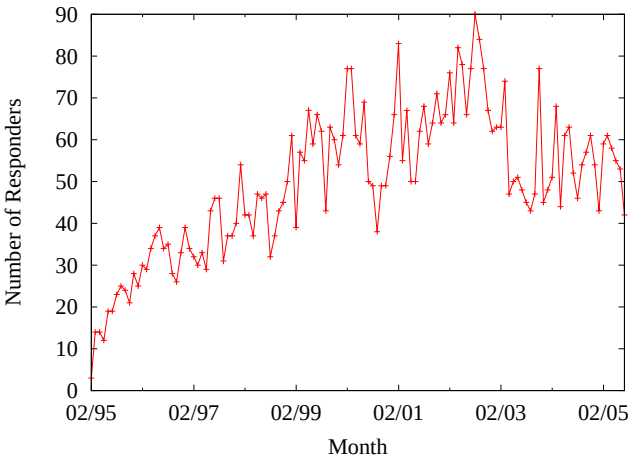

Figure 9: Number of responders active on the mailing list in a given month

7. FUTURE WORK

The results of our experiment are useful in that they give us some insight into what attributes are important in predicting email response on OSS mailing lists. We feel that the moderate size of the Apache codebase and the small number of active participants at any one time may have contributed to the fact that text analysis of the messages did not yield increased accuracy. In the future we plan to experiment with higher volume mailing lists that have more active participants and whose source code is more complex. Our hope is that in a complex project, participants will tend to become experts on certain parts of the project rather then the whole. Possible candidate projects are the Gnu Compiler Collection (GCC) and the Linux kernel.

One problem with our experiments was the amount of time needed for each step in the process. We dealt with over 100,000 email messages, a database that held hundreds of megabytes of data, SVD on matrices with billions of cells, and neural networks with thousands of neurons and nearly half a million interconnections. The total computing time needed from start to finish for this experiment was on the order of days. This was for a project the size of Apache. In order to store and process the amount of data available from projects such as GCC and the Linux Kernel, we plan to modify our soft-

ware so that it can run on a cluster. Although some of the computationally expensive operations were written in C, a large portion of our system exists as Java source code. These modifications may include porting some Java to C.

There were a number of parameters for this experiment. For the most part we did some preliminary testing and picked what seemed to work best. For example, we used the top 200 dimensions of the LSA concept space, one hidden layer of 200 neurons, and a window of 20 responders. It is possible that we might get better results by changing these parameters. Due to time constraints we could not evaluate fully the affect of tuning these values, but in the future we hope to.

8. REFERENCES

[1] Y. Cao, X. Liao, and Y. Li. An E-mail filtering approach using neural network. In *ISNN (2)*, pages 688–694, 2004.

[2] V. R. Carvalho and W. W. Cohen. Learning to extract signature and reply lines from email, Apr. 16 2004.

[3] D. Čubranić and G. C. Murphy. Hipikat: recommending pertinent software development artifacts. In *Proceedings of the 25th International Conference on Software Engineering (ICSE-03)*, pages 408–418, Piscataway, NJ, May 3–10 2003. IEEE Computer Society.

[4] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.

[5] S. T. Dumais. Improving the retrieval of information from external sources. *Behavior Research Methods, Instruments, & Computers*, 23 (2):229–236, 1991.

[6] T. Mitchell. *Machine Learning*. McGraw Hill, New York, 1997.

[7] M. F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[8] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining revision history. *IEEE Transactions on Software Engineering*, 30:574–586, Sept. 2004.

[9] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *IEEE Trans. Software Eng*, 31(6):429–445, 2005.