SmallTalk: Summarizing Instant Messaging Conversations

Simon du Preez
Department of Electrical and Computer Engineering
University of Auckland
sdup571@aucklanduni.ac.nz

ABSTRACT
Instant messaging is increasingly common with the availability of the internet. Chat conversations containing many people can end up with hundreds of messages. Users find it difficult to sift through unimportant messages to understand what has previously been said. In response, we have developed SmallTalk, an instant message summarizer. This provides a user with an array of tools to aid in the comprehension of large chat conversations.

Author Keywords
Instant Messaging; Summarization; Intelligent System, NLP

INTRODUCTION
Instant messaging has seen widespread adoption as it becomes ever more accessible. This is largely due to the popularity of smart phones and the availability of mobile internet. Instant messaging conversations provide many benefits over other means of communications, such as texting, as it allows people from all over the globe to communicate effectively. In addition, large groups of people are able to message each other with ease. Commonly these chat conversations have more than two people and become lengthy for someone to catch up what they've missed. Because of this, users find it hard to read lengthy chat conversations; sifting through all the messages for important information can take a long time. Similarly, understanding the context of the current conversation may require knowledge of what was previously said. In this paper we discuss the development of SmallTalk, a tool that will help users to read and comprehend a large chat conversation. This is achieved by extracting the important information from the conversation and presenting it in user friendly ways.

RELATED WORK
The field of research regarding text spans a huge range of topics. We focus on the topics such as speed reading, text summarization, topic detection and natural language processing (NLP). Initial investigations were into speed reading and presentation of the content. Through various means this has been shown to increase reading speed at the expense of fatigue and comfort [1]. Other techniques such as dynamic text presentation were considered as shown by Georgiev [3].

Instant messages are generally very short and as such are difficult to extract topics and meanings from individual messages. Tian et al. [6] propose an approach to text message summarization. Given a dataset of text messages the first step was to sort the messages into candidate conversations. Their approach uses attributes of time as well as features of the text to group the messages. Then the topic relevancy between conversations was analyzed and then a final cluster of conversations was developed.

Group chats often can contain hundreds of unread messages. Many of these are unimportant that add nothing to the topic of conversation. Many current topic detection techniques are for set documents and not instant message conversations. Topic detection in a message conversation is a way to structure a naturally unstructured environment. With an effective topic detection implementation, chat conversations can be split into sections under the relevant topics. In addition the unimportant messages can be stripped out. Zhang et al. [4] proposed a new method to extract information relating to the topic of chat conversations. It was effective only under the three conditions: useless/unimportant words are common, the messages are short and finally that multiple languages are used in the same conversation.

IMPLEMENTATION

Success
Design of our system commenced with a specific use case in mind. This is where a member of the conversation is unable to read the chat for a period of time. Different conversations are more active than others; more message get sent on average. Therefore the period of time is not defined in our use case but rather the number of messages. We are interested in upwards of tens and possibly hundreds of messages. Once the user returns to the chat, they will be faced with a large amount of unread messages. The user will then refer to our service to see a summarized version of the chat conversation. At this point the user should be able to continue on the conversation without having missed any important details. This defines the success of our project.

Overview
Our system can be broken down into 4 major high level components:

1. Chat input
2. API
3. Message processing
4. Frontend user interface

Chat input refers to retrieving chat messages from a messaging application and storing it in a database for later use. From here the chat messages must be processed via a number of algorithms and metrics to decide what is important. The frontend user interface allows a user to choose which conversation to summarize and displays the

outputs of the message processing. An API was developed which allows requests to retrieve data from the database. This is explained in detail further in the report.

Chat Input

As mentioned, the aim of this project was to summarize unread messages in a chat conversation. To do this a candidate messaging platform had to be chosen. A number of options were considered and discarded based on several factors.

Omlet

Our initial option was to use Omlet chat. This is a fairly new (2014) chat application designed to rival that of Facebook's messenger. It promotes the ability of developers to develop 3rd party applications which can be used in the chat. For example a polling application which allows users in the chat to vote on a number of options. These can be developed natively as Android or iOS apps or alternatively as web apps. Developer documentation however focused on inserting data into the conversation and had little to no information regarding extracting data out. After communication with the developers of Omlet themselves, it was discovered that there is an option which allows 3rd party applications to extract information. However this option can only be enabled by a hidden menu. In addition the exact API calls required to retrieve this information was not able to be found. For these reasons we decided to investigate other options.

Messenger

Facebook has an extremely large user base and would be the platform of choice for investigating chat conversations. Many conversations among large numbers of people occur every day. Facebook is moving towards the separation of the Facebook app and its Messenger app. This has been an ongoing process for several years. The distinction is that the Facebook app is primarily concerned with a user's feed and the Messenger app is streamlined for instant messaging. As a result, API deprecation for retrieving chat messages has been slowly rolling out for specific API calls. In addition to this Facebook is extremely concerned with user's data. Initial investigation into using Facebook's graph API required us to register an app and create access tokens for any user wanting to use our system. In the context of the project, other chat platforms would provide a better proof of concept environment.

IRC

Internet relay chat (IRC) is an application layer protocol which works on the basic client / server model. As it is a protocol, different client applications are able to be developed to work with each other. It is designed for large group chat conversations in rooms called channels. Each channel is dedicated to a particular topic and contains rules and etiquette about what can be said. IRC information including chat data is publically available and so it is relatively easily extracted. However, to extract the chat data a bot was needed to be developed. This was ultimately the platform of choice as retrieving the chat messages proved the most elegant solution when compared with the other platforms.

IRC Bot

As touched on, in order to retrieve the message log a bot was developed. This was made in python in such a way which allowed it to handle a large amount of different connections. This makes it a scalable solution. The bot is able to idle in more than one chat room at a time and record all the messages that have been sent while the bot has been in the channel. In addition to this all the associated metadata is able to be saved in a database for later processing. This includes: the nickname of the user who posted the message, a message ID, time sent, the channel and the type of message.

API

An API (application programming interface) provides functionality that is independent of the implementation. The messages from the IRC bot are stored in a relational MySQL database. An API was developed in PHP to provide a scalable architecture for the system. The API exposes a number of calls such as: fetch_logs_since_part, get_all_links, get_word_cloud_words, get_specific_log_lines and several other key calls. These accept parameter values such as nickname, channel name or message ids. The API returns the data formatted in JSON. This allowed standard JSON parsing libraries to extract the data from the logs easily.

Because the system uses an API as its backend, the frontend system is in no way tied to the data that is being handled. Processing of the messages is done server side. Therefore this provides a high degree of flexibility in future implementations. The current implementation is a website which makes requests for the relevant data. If however a mobile app (iOS, Android or even Windows) was developed this could in the same way consume the API and retrieve all the necessary data.

Message Processing

Message processing was executed upon specific API calls. Functions were primarily coded in PHP. A number of filtering techniques were used in order to breakdown the chat messages.

Stop words are words which are filtered out of text before or after the text has been processed. This involves compiling a list of common words in a language into a dictionary. This dictionary is then compared to a message and when there is a word match, it is removed from the message. Depending on the API call, the original message as well as the stripped message was returned. The dictionary our system used contained common English words including "the", "their" and many others. Additionally slang words such as "lol", "k", etc. were included in the dictionary as slang is typically used in a chat environment. The stop word filtering was used in several of the end user functionalities which are described below.

Message Processing Features

Topic detection of text is a non-trivial task. Slang, misspellings and metaphors in casual conversation can make it nearly impossible to determine a topic in a chat conversation. Often in a conversation with four or more people there is a smaller sub conversation within the conversation. This can occur simultaneously with another sub conversation and therefore the overall chat now has two different topics which are being discussed at the same time. This is an extremely difficult problem for computers to discern. Our system is therefore designed to shift decision making slightly toward the user. If you can present the relevant information in a user friendly way, the user will be able to understand the content of the conversation much quicker than if they had just read the message log. Therefore we focused on the following four main features:

1. Chat log since last left
2. Links categorization
3. Word cloud
4. Events detection

Chat Log

The bot which tracks all messages sent between users, also tracks the times users leave and join the channel. In terms of Facebook messenger, this would simply relate to the unread messages inbox meaning this feature is not restricted to only IRC. The full chat log of unread messages is presented on the dashboard to the user. The dashboard is explained in the design section of this report below. The decision to implement this feature was simple: users would want to refer back to the original messages given a particular topic. In this way the system is able to directly link users to specific portions of the message log. This is something you cannot do on Omlet, Messenger or IRC natively.

Links Categorization

Links are often shared among users of a chat conversation. Friends commonly share music (through YouTube or other services), pictures, text and other documents. Likewise for work purposes, links to forums and other websites are regularly shared. Typically it is difficult to locate a previous link in a chat conversation. This could be a google doc for a group project or simply a popular video. Therefore compiling all the posted links and categorizing them aids not only users who weren't in the conversation at the time, but also all users who would like to retrieve previous links. Links are categorized primarily based on their MIME type. MIME type is simply the type of document for a given URL. They are of the form: text/html; charset=UTF-8 or video/dl for example. There are over a hundred different specific MIME types however our system groups them into their media type: x-world, application, text, video, audio or image. A header request is sent via HTTP for the MIME type of each URL to determine what category it is. However this poses one major problem. Video streaming websites are html text websites with videos embedded into them. An example of this is www.youtube.com. For this reason a set of video URLs is stored in a dictionary. If the URL matches a URL in the dictionary then the link is categorized appropriately.

Figure 1. A sample link that was posted. Link is of type text.

Word Cloud

The aim of a word cloud is typically to show in a user friendly way, what topics or words have appeared most frequently. However frequency of words does not necessarily mean they are important. Some messages may contain very important words which are only used once or twice. It does however give you a fairly decent idea about the long term topics of a conversation if you filter it correctly. The messages were stripped of common words and then frequency was determined. This gave us the ordered list of words for the word cloud. Each message contains a message ID and this is able to be preserved during this algorithm. Therefore our final implementation included the feature where if you click on a word in the word cloud it will take you to that message in the chat log described above. This shows the user the context of the word used.

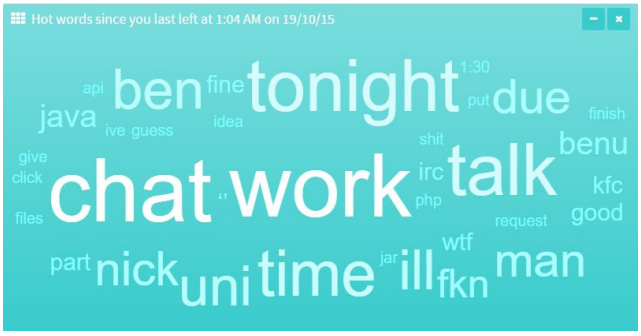


Figure 2. Sample word cloud

Events Detection

Events are often posted casually into a conversation. They are generally of the form "[event] at [time]" or "[something] is on [day]". Either way they contain information regarding the time, event and place. Therefore this information must be extracted from the messages. This is a non-trivial problem amplified by the fact that the event, time and location may be spread across several messages.

Initial investigations looked into NER (Named Entity Recognition). This is a type of information extraction that looks at text for names of people, organizations, times, values and other data. Stanford has produced an NER tool which is designed to extract locations, organizations and names. This worked reasonably well for the purposes of structured text, however it failed when given unstructured text. It relies on proper grammar and spelling. For example locations without capitalized letters were not recognized.

Instant messaging is unstructured and so this was deemed inadequate.

Natural language process is a field of computer science which is focused on gathering meaning from messages and sentence inputs. OpenNLP is a toolkit designed for the processing of a natural language. It uses machine learning to produce models which then generate output on the target text. There are a number of features of this library including, sentence detection, a tokenizer, a name finder, a document categorizer and a part-of-speech (POS) organizer. Our system makes use of the tokenizer and POS organizer in addition to our own algorithm to search for events.

Event Detection Implementation

The event detection logic was implemented in Java using our API framework that was developed earlier.

The algorithm overview is as follows:

1. Strip common English words
2. Search for time keywords
3. Tokenize and tag the messages
4. Remove everything that isn't a noun, verb or unrecognized
    a. If no words are left, look in the previous message for event keywords until an event is found
5. An event and time is paired.

First a request is made to the API for the stripped chat log. From here a graph structure is produced by forming a doubly linked list with all the messages. In this way the message log is able to be traversed forwards and backwards as needed. Specific keywords based on a regex string are searched for in the stripped messages. The keywords include times, dates, days, months, hours, seconds and minutes. In addition it includes colloquial versions of these keywords. Every message with a time keyword associated with it is extracted and further analyzed. The time portion of the message is extracted and saved for later use (E.g. "Monday", "at 6:10pm").

The message is now stripped of time and common English words. The words that are left are tokenized and tagged. The words that are chosen to be events are decided based on a set of rules. The rules are that verbs, nouns and unrecognized words are kept. The aim of this step is to extract words such as "coffee," "swimming," etc. Unrecognized words are chosen to remain as more information is generally better than leaving out potentially important information. For example words such as Skype or Pak'n'Save for example are not recognized and are clearly important.

At this point we will be left with one of 2 cases in the best case (ignoring misclassification of tokens). The first case is that we have an action and a time and our algorithm is complete. The second case is that the action was mentioned in a different comment. In this case we make the assumption it was a previous comment. The system searches all previous comments until a relevant match is found.

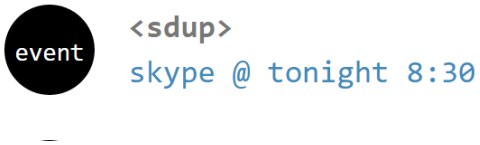


Figure 3. A sample event that has been extracted from the chat conversation

Performance

Performance of algorithms is vital. If an algorithm takes too long to execute, then the user could have read the chat in the time it took the algorithm to determine what was important. This defeats the purpose of the whole application. Overall our system was tested with an unread conversation of over 2,000 message lines. This contained 10,000 different words for an average of approximately 5 words per message. The initial implementation of the event detector took nearly 30 seconds to run on a high performance computer and around 2.5 minutes to run on the server. This was clearly unacceptable and a refactoring was needed.

Initially tokenization and tagging was done individually on each message. This made keeping track of message ID's much easier. However this was found to the cause of the inefficiencies. Instead each message was appended to one long string separated by a delimiting character. This character enabled the identification of each message. The entire message log was then tokenized and tagged as one string which cut execution time down to nearly 0.6 seconds on the high performance computer and only a few seconds on the server.

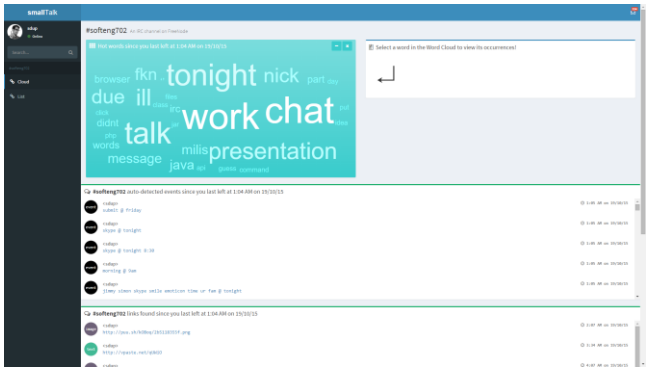
Frontend User Interface

The frontend of the application is a website. As mentioned previously, due to the API any type of frontend is able to be developed. The two main aspects of the frontend are the login page and the dashboard. Both went through an iterative design process of several different mock-ups. The login page allows a user to specify a nickname and a channel name. The dashboard will then show the previously mentioned features. A session is created for a particular user which saves the user and channel name. The final implementation made use of a myriad of technologies including HTML, JavaScript, CSS, AJAX, Bootstrap, jQuery and others. jQCloud was used for the generation of the word cloud.

Figure 4. Login form

DISCUSSION

As mentioned, finding a perfect topic detection algorithm for instant messaging is a non-trivial task. Therefore we implemented a number of different tools which enable a user to decide for themselves what parts of the conversation are important. This is achieved through the tools we developed as well as the easy navigation through the chat log. The links categorizer works as intended and could be further improved with added sites to the relevant dictionaries. The word cloud currently works only on frequency of words. This could be improved a number of ways through topic detection methods that weren't implemented. The event detector works as expected however it also picks up a fair amount of useless events. For example if I type "Meeting with Paul tomorrow at 12:00pm," the event will successfully be extracted and displayed as "Meeting Paul @ tomorrow 12:00pm." However some sentences such as "give me 5 mins?" are also detected which perhaps should not. In addition the specific context is not known but that can easily be found. An example of this is the event "submit @ Friday." This doesn't give any context however you can click the event and fairly quickly determine the context of the message.



CONCLUSIONS

The aim of our project was to summarize lengthy chat conversations to quickly get up to speed on what has been said. We have presented SmallTalk, an application which has achieved our goal through a number of tools which are at the disposal of the user. Visual presentation as well as intelligent filtering algorithms have been used together for the maximum benefit of the user. The final product is promising as a proof of concept.

ACKNOWLEDGEMENTS

The author would like to thank the lecturer Beryl Plimmer for her continued support and encouragement throughout the project. In addition the author would like to acknowledge Jim Warren. Finally the author would like to acknowledge all members of the team for providing excellent work and backing throughout the project.

REFERENCES

1. Dingler,T., Shirazi, A.S., Kunze K., and Schmidt, A. Assessment of stimuli for supporting speed reading on electronic devices. In Proceedings of the 6th Augmented Human International Conference (2015). ACM, New York, NY, USA, 117-124. http://doi.acm.org/10.1145/2735711.2735796
2. Oh, J., Nam, S., and Lee, J. Generating highlights automatically from text-reading behaviors on mobile devices. In CHI '14 Extended Abstracts on Human Factors in Computing Systems (2014). ACM, New York, NY, USA, 2317-2322. http://doi.acm.org/10.1145/2559206.2581176
3. Georgiev, T. Investigation of the user's text reading speed on mobile devices. In Proceedings of the 13th International Conference on Computer Systems and Technologies (2012), Boris Rachev and Angel Smrikarov (Eds.). ACM, New York, NY, USA, 329-336. http://doi.acm.org/10.1145/2383276.2383324
4. Zhang, H., Wang, C., Lai, J. Topic Detection in Instant Messages. Machine Learning and Applications (ICMLA), 2014 13th International Conference on (2014), vol., no., pp.219,224 http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7033118&isnumber=7033074
5. 5. Ahmed, F., Borodin,Y., Puzis,Y., and Ramakrishnan, I. V. Why read if you can skim: towards enabling faster screen reading. In Proceedings of the International Cross-Disciplinary Conference on Web Accessibility (2012). ACM, New York, NY, USA, Article 39, 10 pages. http://doi.acm.org/10.1145/2207016.2207052
6. 6. Tian, Y., Wang, W., Wang, X., Rao, J., Chen, C., and Ma, J. Topic detection and organization of mobile text messages. In Proceedings of the 19th ACM international conference on Information and knowledge management (2010). ACM, New York, NY, USA, 1877-1880. http://doi.acm.org/10.1145/1871437.1871752