UNIVERSITY OF CALGARY

Auto-tagging Emails and Instant Messages with User Stories to Build a Knowledge Base for Distributed Agile Projects

by

S. M. Sohan

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

December, 2010

UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "Auto-tagging Emails and Instant Messages with User Stories to Build a Knowledge Base for Distributed Agile Projects" submitted by S. M. Sohan in partial fulfillment of the requirements for the degree of MASTER OF SCIENCE.

Supervisor, Dr. Frank Oliver Maurer
Department of Computer Science

Chairman, Dr. John D. Doe
Department of Academic Computing Services

Chairman, Dr. John D. Doe
Department of Academic Computing Services

Chairman, Dr. John D. Doe
Department of Academic Computing Services

Date

Abstract

Paragraph 1

Paragraph 2

Paragragh 3

Acknowledgements

I have been blessed with the trust and support of a number of people and organizations while working on this research. In this regard, I would like to thank -

Dr. Frank Maurer and Dr. Michael Richter for your help with formulating and advancing this research. You have provided me with useful guidelines and suggestions at various steps and kept trust in me all the way.

Syed Rayhan and the Code71 team for providing me with an excellent environment for learning and innovative thinking. Working with you have been a memorable experience and helped me coming up with this research idea.

University of Calgary for your financial support without which I couldn't carry out this research.

My loving wife, Shahana, for giving me a happy family life. You have come all the way from Dhaka to Calgary to accompany me and that proved to be a lot for my well being.

My parents, S. M. Afaz Uddin and Mrs. Kh. Shirin, for inspiring me all through my life. Your continuous inspiration has given me the enthusiasm and courage at all walks of the life and this graduate research is one example of that.

Bangladesh, my country, for your promise to educate and raise me. You have given me the strong foundation to face the world.

Table of Contents

List of Tables

List of Figures

Chapter 1

INTRODUCTION

Agile software engineering process is a collaborative approach to incrementally deliver software in small iterations (e.g. two to four weeks). Agile teams commonly use "user stories", a small description of a desired feature in everyday language, to capture the requirements. An example user story is as follows:
As a shopper, I want to pay online to checkout my shopping cart using MasterCard, Visa or Amex credit card from a secured web page only.

The details of such user stories are discussed as an ongoing basis among the people involved in a project. For example, as the engineering team start working on the stories, they often consult with customers and other teammates to further clarify such user stories. Whenever possible, face-to-face communication is preferred as the principal communication medium between customers and developers in agile processes [**?**] [**?**] [**?**] [**?**]. In collocated setups, where the people work in close proximity, such informal communication relays the tacit knowledge among the people.

However, for distributed projects, especially when there is a huge time zone difference among people, face-to-face communication become infeasible. As a result, people use alternatives to mitigate this shortcoming such as phone calls, emails, instant messaging, wiki, web forums and so on. So, in such setups communication is often text based instead of tacit knowledge sharing. In this sense, distribution makes it easy to capture the knowledge since much of it is textual and electronic. For an example, considering the aforementioned user story, the developer may send the following email to the customer to further clarify expectations.

Subject: *Clarification required on online credit card payment*

Hi Bob:

Please clarify if the shoppers need to provide the security code of the credit card while doing checkout.

Since email is a persistent tool, one can use this knowledge in future. But, email is also a personal tool and its hard to transfer a selection of project related emails from the rest in a reusable format. So, when someone leaves the team, it becomes hard to access that knowledge when needed later down the road. To retain these useful knowledge, this thesis implemented and evaluated a tool called "Taggy" that automatically grabs and tags the emails and instant messages with relevant user stories.

The relationship between a user story and email or instant message is not explicit. To establish the implicit relation, Taggy uses a machine learning technique, Case Based Reasoning (CBR). The CBR system looks at the text, people and temporal similarity between emails and user stories. The evaluation results show that a well trained software can auto-tag emails with user stories. Also, we found that combining context information with text similarity helps to find the related user stories with higher accuracy than using just text match alone.

In the existing literature attempts have been made to capture software project related knowledge following several alternative approaches. For example, commercial tools exist that link software code commit messages with a requirement or ticket given the ticket number is present inside the commit log. Also, another approach is to use wiki or online message threads, where the contents are manually linked with the user stories. However, to use such approaches, one needs to remember identifiable tokens or learn markup syntax (e.g. wiki), which is often burdensome for business users. To reduce this learning curve and human efforts, Taggy builds a knowledge base by automatically tagging emails and instant messages with user stories.

The remaining part of this chapter is organized as follows: First, I discuss the necessary background material about user story and distributed agile projects to develop a shared understanding of the terms used in this thesis. Then, the research motivation is discussed. Finally, I present the research goals and related challenges.

1.1 User Stories - An Overview

In agile projects, user stories are used to capture requirements. In short, user stories represent the need for a feature from the view point of a potential user of a software. Mike Cohn[**?**] defines user stories as follows:
A user story describes functionality that will be valuable to either a user or purchaser of a system or software. User stories are composed of three aspects:

- a written description of the story used for planning and as a reminder
- conversations about the story that serve to flesh out the details of the story
- tests that convey and document details that can be used to determine when a story is complete

As outlined in the above definition, conversation plays a central role in fleshing out the details of the user stories. Ron Jeffries [**?**] and Rachel Davies[**?**] also emphasize the role of conversation being a key component of user stories.

Another popular acronym INVEST suggested by Bill Wake [**?**] that defines good stories being Independent, Negotiable, Valuable to users, Estimable, Small and Testable. However, to hold all these characteristics yet being Small essentially points to conversation as means of detailing the user stories.

In an agile project, the customer or a representative of the customer is responsible for coming up with these user stories with the help of the stakeholders. Next, the user stories

are prioritized in a list known as "Product Backlog". At every iteration (typically two to four weeks), the team picks the top user stories from the product backlog known as "Iteration Backlog" with the target to deliver the user stories at the end of the iteration. A user story is supposed to be limited to a scope so that it can be delivered within the iteration. However, the details about the user stories are laid out as on going basis during the iteration through customer collaboration and feedback.

One or more team members are assigned to deliver a user story, which may involve design, development, testing and such tasks. It is a common practice that the customer and the assigned people collaborate to fine tune the user story details.

1.2 Distributed Agile Projects

The members of a distributed agile project work from different geographical locations and adopt agile principles. In practice, the distribution can take several models such as the following three outlined by Braithwaite et al [**?**]:

1. Agile outsourcing - the development team is offshore to the customer.

2. Agile dispersed development - developers spread over different locations, such as open source projects.

3. Distributed agile development - customers are distributed and one development team is distributed evenly to stay close to the customers.

Again, the geographical distribution can be anywhere between offices across the road to opposite part of the world. For example, a team may work for a customer at the same city or a country separated by ten hours of time zone difference. Distributed agile teams often use different electronic communication channels such as Emails, Instant Messaging,

Wiki and Forums as well as online project management tools to uncover the important details about the user stories.

Cost has been identified as one of the principal deciding factors for distributing projects across the globe [**?**]. However, projects also go distributed to utilize global talent supply and local knowledge [**?**]. Previous studies have reported success with distributed agile projects. For example, Sutherland et al. reported a linear scalability with a globally distributed outsourced team following a Fully Distributed Scrum (Scrum is a popular agile method) model [**?**]. Similarly, Yahoo! had success with distributed teams that valued people over process and exercised the right information sharing formats and timing [**?**].

To ensure a shared view of the "Product Backlog" and "Iteration Backlog" distributed teams often use web based project management tools. There are a number of such tools available commercially as well as open-source [**?**]. Such tools help people working remotely to have a single backlog and collaborate around that.

1.3 Research Motivation

The existing literature about collaboration in distributed agile software projects suggest the use of various electronic mediums to counter for missing face-to-face communication. Because of time zone difference, such teams often use asynchronous communication tools such as email, wiki in addition to synchronous ones such as telephone call, instant messaging etc. For example, Robarts [**?**] found that teams would follow up with a written email after they had a telephone meeting to ensure the message is clearly understood. Moreover, of all the available written communication tools, email has been regarded as the most-widely used one by several former studies [**?**] [**?**]. Additionally, even in collocate settings, people often use emails to communicate as it doesn't require immediate response

and overheads like scheduling meetings.

Although having multiple communication channels adds flexibility for collaboration, it also means the knowledge gets fragmented across different channels. This fragmented knowledge may be spread over the project management tool, wiki, emails, instant messages and other places. To retain the important details about the user stories, one needs to combine these fragmented pieces. But looking into email inboxes of colleagues may be difficult, if not impossible. Even worse, when a teammate leaves, the emails are gone as well. So, the fragmented pieces of knowledge is either lost or very hard to combine together. Even if one has access to the emails, manually copy-pasting all the emails and combining with the relevant user stories would be a time consuming task to say the least. The following user story and email collaboration shows an example of the knowledge fragmentation:

User Story (Available in Project Management Tool)

Title: "As a/an VM System I want to load Auction data from NADA data file"

Description: Please see the 3 documents related to Auction data

1. Auc Layout
2. **Regions Table**
3. AucData.zip

In this example, the user story represents a data integration with a company named NADA. NADA provides vehicle auction information for regions denoted by a "region code". However, as the developer starts working on NADA integration, she asks the following question about the region code to the customer through email.

Email#1 (From Developer to Customer)

Subject: "Initial Questions regarding NADA region value"

Body: We found that the Region value can be **Alpha-numeric** that will be received

from NADA Auction. But, the NADA WS accepts only **Numeric** Region value. Is there any conversion rule to convert the region value into numeric?

Thanks

To answer this question, the customer replies with the following email.

Email#2 (From Customer to Developer)

Subject: "RE: Initial Questions regarding NADA region value"

Body: I talked to NADA and they provided me with the **attached conversion table** for **Alpha-numeric** to **Numeric** region codes.

Please note that the email from the customer included a conversion table for "region codes".

Given this example, if a person only looks at the user story, she will find partial information about it. So, the whole conversion of "region code" may be confusing to her. However, when she can see the emails along with the user story, she may have a better understanding about this user story.

This information may be required since a distributed agile team have to cope up with changes in team composition as new people will join and some people may leave. So, if it is possible to combine the fragmented knowledge from different places such as emails, instant messages and project management tools, one can find the important details of the user stories. Such details many not be obvious when only the user story is available. Similarly, the knowledge can help in testing of the features since it is expected to contain customer feedbacks and clarifications on user stories.

The motivation for this thesis stems from the pursuit of developing a knowledge base by combining the fragmented knowledge from emails, instant messages and project management tools with the least possible human efforts.

1.4 Research Problem

In software development projects, developers sometimes leave the team for various reasons. As said before, in distributed agile projects, important knowledge is often shared by emails. To transfer this knowledge for future use, one first needs to find the project related emails. Secondly, such setups people often use web based project management tools to capture user stories and planning information. If the emails can be related to specific user stories in the project management tools, then another developer of the team will be able to easily find the necessary information about a user story when required.

From an end users' point of view, manually finding and relating the emails with user stories is a time-consuming and costly process. This essentially means copying the emails from inbox and putting into a system and linking with a user story. This manual process is cumbersome to say the least. However, if a software can do this, then it may work as a knowledge-base even after someone leaves the team. Such a software needs to be unobtrusive for the most part so that it doesn't require much human effort. So, a technical solution needs to be in place that can capture the project related communication without violating people's privacy. For example, it cannot look into the email inbox of a developer nor should ask a developer to manually enter the chat log into the system.

The core technical challenge in devising such a software solution is understanding the emails and relating them to specific user stories. The relation between an email and a user story is not explicit. Alternative approaches to make it explicit, such as using identifiable tokens in email subjects, also makes the communication process obtrusive as it adds an extra step to lookup the identity or memorizing it before writing an email. To alleviate this extra step, one can modify email clients to do the lookup. But doing so for the multitude of emails clients including the ones on mobile phones is cumbersome. Also, it imposes a behavioral change or a learning curve for the for the people at the business

end.

To find out the implicit relation between a free-format email and a use story, a software needs to handle similarity between two texts, which is a standard problem. Emails may not show very high text similarity with the related user stories. This text component makes the relevance between a user story and an email an approximation or fuzzy match. Also, pure text retrieval limits how accurate the assignment of email and user stories can be. This is because people use different vocabulary and often a tacit communication is there underneath a written form. Using context information has the potential to increase this accuracy. So, a software needs to be able to combine both the text and context similarity for auto-tagging emails with user stories.

To account for the context, one needs to use the associated meta information that are available. The contexts of an email and a user story are not exactly similar. For example, the context of an email has a time-stamp, sender, recipients and past conversations. On the other hand, user stories have a different context, such as their development time or iteration timeframe, developers and customers. An auto-tagging solution needs to use these context meaningfully so that each component gets its deserving share of importance in the relevance computation. So, it is important to have a formula that produces similarity ranking considering the fuzzy text similarity and associated context relevance.

Next technical challenge is to evaluate the accuracy of the auto-tagging system. For the auto-tagging to be useful, it shouldn't make too many mistakes in finding similarity. The accuracy of this system needs to be tested using real world data.

Another technical challenge is to design an adaptive auto-tagging system so that it can adapt to different communication patterns. Since distributed agile teams across the world have differences in who, how and when they communicate about their user stories, the auto-tagging system needs to adjust its similarity computation accordingly.

For example, one team might spend more time communicating about user stories that are currently under development while another team might prefer to discuss about the upcoming ones. An auto-tagging system that uses the meta information of time needs to learn this pattern to make a right decision.

1.5 Research Goals

The principal goal of this research is to develop a tool that can automatically read and relate emails with user stories based on agile project context. The tool needs to minimize human efforts in doing so. Another complementary goal is to evaluate the tool using real world data. The underlying approach and evaluation of Taggy is discussed in Chapter X.

This research also aimed to investigate the difficulties in solving the various aspects of the research problem, namely i) relevance ranking of user stories for a given email, ii) using text similarity in addition to context relevance, iii) evaluating auto-tagging accuracy and iv) developing an adaptive auto-tagger. The findings from this investigation is expected to provide a foundation for next level of research in this area.

1.6 Key Contributions

This research work has been published and presented at XP 2010 [**?**].

The key contribution of this research is, it demonstrates a novel approach of building a knowledge-base for distributed agile teams from emails and user stories. This approach has the similar motivation as found in some previous works, such as clustering emails archives[**?**], tagging forum messages with source code change set[**?**] and using wiki to share knowledge. However, this research shows a machine learning approach to conglomerate fragmented knowledge from emails, instant messages and project management tools

minimizing human efforts. This new technique adds to the existing body of knowledge.

Another key contribution is, it shows that text and context similarity can be used to find relevance between two different types of entities such as emails and user stories, where the degree of text similarity may be inadequate. We have used Case Based Reasoning (CBR) for auto-tagging. As this research findings suggest, other CBR systems concerning different entities may utilize this technique.

Chapter 2

LITERATURE REVIEW

Researchers have attempted to solve the problem of knowledge management for software development projects from different points of view. Also, the industry is using a number of available commercial and open-source tools to retain and manage knowledge. To find the similarity and contrast of this research with the existing approaches, this related work chapter is divided into sections dedicated to the following five areas:

1. Communication in distributed teams
2. Capturing knowledge from emails
3. Wiki based knowledge sharing
4. Context based knowledge management
5. Review of existing tool support for knowledge sharing

2.1 Communication in Distributed Teams

The Agile Manifesto[**?**] puts heavy emphasis on customer collaboration. Understandably in distributed projects, the geographical and temporal distance accounts for communication challenges compared to a collocated setup. Several existing research focused on the use and effectiveness of the different communication channels that are used in distributed projects.

Thissen et al. points out that a distributed team to become successful must use appropriate tools to ensure easy flow of information [**?**]. Lanubile provided a categorization

of such collaboration tools [**?**] that include software configuration management, bug and change tracking, build and release management, product and process modeling, knowledge center and communication tools. Lanubile also pointed out that asynchronous communication tools include emails, mailing lists, newsgroups, web forums and blogs while synchronous ones include telephone and conference calls, chat and video conferencing. Of all the asynchronous tools, email has been identified as the most widely-used and successful collaborative application because of its flexibility and ease of use.

The use of synchronous vs. asynchronous mediums depends on teams and their temporal as well as geographical distance. For example, Hole et al. found that teams often preferred email and chat over telephone calls for meeting between two sub-teams at two different locations due to language barriers [**?**]. Here is an excerpt from a Scrum Master, one of the participants of their study:

"we tried to use telephone conferences, but it did not work well, because of language problems. It is also easier to understand each other when relying on written communication. Also extensive use of chatting makes it possible to ask a question right away. It takes time to organize a telephone-conference."

Korkala et al. found that despite the availability of richer communication media, the customers preferred to use emails for most of their mid-iteration communication needs[**?**]. Here is an opinion from a customer of their case study about the selection of mid-iteration communication:

"Emails will do just fine. They are enough."

Hildenbrand et al. suggested that customers and testers of distributed agile teams need to work closely to flesh out the acceptance criteria for the user stories in the iteration backlog [**?**]. Also quick feedback from the customers is needed during the design and development of a user story. They also emphasized that this knowledge could be of great use and must be shared with the whole team. To collaborate and capture this knowledge,

they suggested the use of online groupware and instant messaging.

Cataldo et al. looked into the types of knowledge contained in the emails of a distributed project and found that email was the most preferred medium for information exchange and task negotiation related topics[**?**]. Although the target was to use groupware for such activities so that the knowledge could be easily retained shared, their data suggests that developers relied heavily on emails for information acquisition. Layman et al. classified the emails exchanged in a distributed agile project into several types such as: i) Changes in specification, ii) Clarification, iii) Prototype, iv) Feedback, v) Planning/scope, vi) Bug, vii) Acceptance test and viii) Post-release bug related emails [**?**]. They also found that clarification related emails were the most frequently exchanged of all these categories.

To summarize, this section of the literature review suggests that distributed teams often use emails and other text-based asynchronous mediums alongside real-time collaboration tools to share important knowledge about their projects. Based on these findings, this thesis emphasized on automatically capturing project related emails and instant messages. Since emails and instant messages are widely-used to share knowledge among people in distributed projects, the solution shown in this thesis can be used to retain and reuse this knowledge.

2.2 Capturing Knowledge from Emails

In the existing literature, there have been attempts to extract knowledge by mining software project related emails. Largely, the concentration of such mining efforts has been around clustering the archives to meaningful groups. Also, there has been work around the visualization of email archives so that one can easily navigate the corpus of emails.

Berlin et al. designed and implemented a group memory system called TeamInfo [?]. TeamInfo let people to use CC: or carbon copy feature of emails to forward a project related email to the TeamInfo system. Once TeamInfo reads the email, it finds the predecessors if any and try to classify the email into one of the preset categories. These preset categories are similar to what we see in email folders or filters, namely grouping by one or more of conditions involving sender, recipients, subject and text containment. This classification is intended to help in browsing the emails based on their high level categorization. However, they faced challenges with coming up with categorizing conditions as at times one email could be part of multiple categories. Also, a fine grained categorization would lead to hundreds of such classes, which again, makes it hard to easily browse to a topic of interest. To write a filter, TeamInfo required significant experience as the rules were expressed using a declarative language. TeamInfo helped them in retaining and sharing knowledge. Taggy has a similarity with the data feed process of TeamInfo, as it uses the same CC: feature of emails to get hold of the knowledge. However, Taggy is designed to help agile teams, where the presence of assigned developers, customers and time box provide a context to the plain text-based user stories. As a result, instead of trying to group the emails, Taggy tries to automatically link up emails with the user stories so that one can follow the discussions related to specific features of a software. This alleviates the overhead of creating and managing shared grouping rules.

A knowledge-base is often used to recommend a developer about required source code change locations. Hipikat is such a recommender that uses interlinked information from several sources, such as CVS log messages, online documents and newsgroup threads[?]. Here, the newsgroup threads are essentially the group emails exchanged among people working on a project. To infer the relationship between a thread discussion and other artifacts, Hipikat uses “References”, a header element that can identify the thread’s topics of interest. However, in agile projects such email threads are often exchanged

between customer and developers and it might require extra effort either to memorize or to lookup the "References" identity every time before starting a discussion. Taggy emphasizes seamless customer collaboration and reduces this extra look up cost through utilizing machine learning techniques to infer the hidden relation between an email and user story.

Existing research also looked into mining and visualizing the contents of software related email archives to extract out the important patterns. For example, Medynskiy provided a multi-modal visualization of email, instant message and CVS commit logs from the Python project[**?**]. They mined the text information to extract out meaningful groups and interlinked among the groups to contributors of the contents. This way, one could navigate related knowledge from different sources that are otherwise fragmented. In effect, Taggy also tries to unite the fragmented pieces of information. But the concentration is at a higher level (user stories) as opposed to lower level artifacts (CVS logs). Also, instead of mining the archived data sources, Taggy builds the knowledge-base on the go.

Software requirements traceability helps understanding a requirement based on its underlying reasoning[**?**]. However, for distributed agile projects, this is hard since knowledge is dispersed. Capturing the emails with user stories has the potential to improve the traceability in such cases. To attain this goal, the approach of Taggy is in alignment with the aforementioned research about building a knowledge-base from emails. However, it is extended to be used in agile projects, where the use of available context helps the computer to make an informed guess about if an email is related to a user story or not.

2.3 Wiki Based Knowledge Sharing

The overwhelming success of Wikipedia [**?**] is a result of collaborative editing. Being inspired by this, several researchers explored the potential of using Wiki for software knowledge management. For example, Decker et al. have found successful requirement capturing and stakeholder participation through wiki platform[**?**]. They proposed a document structure or template for using Wiki in software projects so that different artifacts such as user stories, tasks and plans can be managed following a standard approach. However, they also discovered that using wiki for such tasks adds complexity resulting from its syntax to denominate metadata.

Decker et al. also proposes the use of semantic wiki, an approach where additional meta information is used, so that it is possible to reuse the contents using machine learning [**?**]. In this regard they introduced Wikitology, a term that combines Wiki and Ontology. The Ontology is contained as a part of meta-data in the templates for different software engineering artifacts, such as user stories, discussions etc. Once these templates are defined, a software engineering artifact becomes an instance of one of the templates. So, the relationship between two different wiki pages can be established based on their underlying templates' meta information. Given the ontology of different wiki pages, they used two techniques, Latent Semantic Analysis and Case Based Reasoning, to compute the similarity. In this thesis I used the second technique, CBR, to find similarity between an email and a user story. However, instead of using a predefined ontology, Taggy uses the available meta information or context from an email. Also, Taggy combines the context and text similarity in the computation as opposed to solely relying on the context.

Tosic et al. developed a Collaborative Semantic Web Portal Prototype that utilizes wiki platform for collaborative knowledge acquisition to be used in agile project management [**?**]. Their solution added access control to the wiki pages and also several indicators

such as number of contributors, the creator and so on for every page. Using this system they observed encouraging results such as teamwork and collaboration, open information flow and the light-weight agile vigilance.

To ease the authoring of wiki pages, OntoWiki shows a few interesting techniques [**?**]. For example, they present a rich editor with support for real-time search from existing content to reduce the time required to produce an article and cross-link with different pages. Also, to make it collaborative they brought the concept of commenting and rating. While these techniques can greatly help in authoring wiki content, to be used in software knowledge management, the developers and the customer need to actually put the contents on the wiki and ensure the contents are updated accordingly. This may be a barrier, as it requires considerable human efforts to a cause that have lower immediate business value compared to some other tasks, such as developing and testing the software.

Chau et al. studied the contribution to a software wiki contents from people working on different roles [**?**]. They observed the use of MASE, a wiki-based knowledge sharing system, in a medium-sized software company. They discovered a very high usage, exceeding 90%, of MASE as an asynchronous collaboration medium. But surprisingly, only 10% of the wiki content were produced by the managers, while the rest was contributed by the technical team. Moreover, none of the managers were in the list of top 10 contributors. They also found that there was a greater need for unstructured than structured knowledge. In agile software engineering, informal and continuous customer collaboration is heavily utilized. So, it is important to choose a knowledge sharing media that offers most flexibility as well as less learning curve for the customers. This thesis recognizes the need for unstructured knowledge and uses email as a source of such knowledge. As a result, the knowledge base is generated while people communicate as opposed to "documented after the knowledge is shared".

Wiki has also been used to write executable acceptance tests with the motivation

that customers can provide the acceptance criteria for a user story in simple wiki tables [**?**]. For example, Young et al. mentioned the use of wiki for automated acceptance tests so that the continuous integration system could execute the tests[**?**]. However, such tabulation of acceptance tests often do not capture the underlying knowledge behind the facts, which may be necessary to understand a user story.

While wiki can work as a knowledge sharing tool, an addition of email discussions offer several unique benefits. For example, email is a general purpose communication tool, so most people are already familiar with this. Wiki is great for collaborative editing. But knowledge sharing often takes the form of discussion where information flows back and forth between people. For example, it is easy to follow a discussion than to find the latest changes in a wiki page. As Chau et al. pointed, wiki and similar centralized knowledge capturing approach often employs people who are not involved in the day-to-day software development and as a result, there are concerns raised about the usefulness of this approach[**?**]. Capturing knowledge from emails and linking them with user stories will complement the collaborative editing benefits of wiki. So, it will bring the useful details that are already discussed from emails and add to the knowledge from the wiki, if any.

2.4 Context Based Knowledge Management

Maalej et al. provided a context based solution for lightweight knowledge sharing in distributed software projects [**?**]. They identified two key steps in the knowledge sharing process, namely, knowledge access and knowledge sharing. They outlined a framework to facilitate the access to relevant implicit knowledge from the large amount of dispersed sources. The framework relies on the context of different knowledge items as well as knowledge consumer's usage pattern to proactively provide access to the available knowl-

edge. To facilitate knowledge sharing, they identified that a knowledge provider needs to present the knowledge in generalized format so that it can be applied on a different context by another person. This requires additional efforts from the provider without much of an immediate benefit.

To minimize this burden, they proposed the use of additional semantic information or ontology in knowledge items, which can be used in computer-based information retrieval through context matching. Their proposed knowledge management solution derives a profile of the developers from their usage history. Based on this profile and the semantic information present at the knowledge items, the relevant ones are found. This proposed solution mainly targets the knowledge capturing at a fragment of source code level. Although this framework provides an abstract form for a knowledge management solution, a concrete description of how the acquisition of dispersed knowledge from sources like emails, wiki, instant messages and forums are incorporated is not discussed. As suggested in this approach, Taggy uses the context information in addition to text relevance to interlink different knowledge items. However, Taggy is designed to capture high level knowledge from emails and user stories as opposed to the level of source code fragments.

Ratanotayanon et al. developed Zelda, an Integrated Development Environment (IDE) plugin to interlink source code with user stories [**?**]. This plugin provides an interface where a developer can select one of the user stories as an active user story. Next, when she is ready to push the changes in the code repository, the plugin automatically links all the changes against the active user story. It also allows a developer to manually select source code that are related to the user story. The goal of this link recording is to help other developers who might need to make changes to the feature from an existing user story. To complete the future modification, one can navigate to relevant source code from the stored links. Since it is a common practice to modify source code and keep revisions, Zelda updates the user story and code links whenever the code has

a new revision to keep the links pointed to the latest revision. In their evaluation, they found that Zelda helped developers to focus on relevant source files given a new user story to modify an existing one.

The concept behind Zelda, linking source code changes with requirements and other high level artifacts, is also present in FEAT[**?**] and Mylyn[**?**]. For example, Mylyn is an IDE plugin that builds a task context as developers select a task and make necessary changes. The context of a task includes information about the source code changes, API usage and documentation lookup as a developer works on it. Having this context, a developer can easily navigate and search though relevant source code for a task. Mylyn also has integrations with several task and bug tracking tools.

As seen with Zelda and Mylyn, high level artifacts such as user stories and tasks can be used as an entry point to a knowledge base. Taggy follows this same approach. However, instead of linking the source code with the user stories, Taggy links the email discussions. So, the knowledge base produced by Taggy is likely to complement Zelda with relevant higher level information from emails that can provide more insights into the user stories. In a sense, Taggy interlinks two high level artifacts which complements the knowledge found from the interlinked low level items.

2.5 Review of Existing Tool Support for Knowledge Sharing

Distributed agile projects often use globally available project management tools to share knowledge [**?**]. These tools allow the teams to capture the product and iteration backlogs, user stories and project planning information. Typical project planning information includes the estimation and assignment of tasks or user stories to developers and testers. Also, some tools provide support for collaboration about the user stories.

There are a number of commercial tools available for agile project management such as

VersionOne[?], Mingle[?], ScrumPad[?], ScrumWorks[?], IBM Rational Team Concert[?] etc. These tools offer templates for capturing various agile software engineering artifacts such as user stories, backlogs, acceptance tests and so on. Also, they provide process specific workflows, for example, a workflow for Scrum includes a virtual story wall, burn-down and other charts for distributed project tracking. Some of these tools let people to collaborate using message threads and wiki. These tools are generally useful in planning and tracking activities. In addition to commercial tools, one can also use open-source agile project management tools such as XPlanner[?] and Trac[?].

Several issue or bug tracking tools such as Jira[?], Bugzilla[?], FogBugz[?] are being used in distributed agile projects as well. To support agile processes, these tools offer plugins for agile projects that include some of the features from the aforementioned project management tools. These bug tracking tools provide a messaging system where people can collaborate around bugs. The knowledge shared during this collaboration is often used to solve new bugs[?].

Distributed agile teams often use general purpose project management tools such as Basecamp[?], Teambox[?] etc. While these tools are not tailored to provide agile specific artifacts, they are often picked for their simplicity to use. So, instead of providing a template for an user story or an issue, this tools simply provide a to-do list. Similarly, instead of iterations, people rely on milestones. And as seen with other tools, the general purpose web-based project management tools also allow people to use messaging systems to collaborate on their to-do lists.

Alongside project management tools, some distributed agile teams use general purpose shareware tools such as Microsoft SharePoint[?]. SharePoint provides an infrastructure to manage websites, communities, content, search and reporting that is mainly configuration based and does not require coding efforts for the most part. In large distributed agile projects, where multiple teams are involved, SharePoint is often used to manage the

knowledge.

Some agile projects are now using source code hosting platforms for project management as well. For example, github[?], CodePlex[?] and similar source code hosting platforms have support for defining issues and user stories. Also, they allow some project planning features such as iteration backlogs, estimation and assignment of work items etc.

Agile teams also use continuous integration tools such as Hudson [?], CruiseControl[?], Microsoft Team Server[?] etc. so that the status of the current build is automatically communicated. These tools provide a dashboard and detail view of a project's health that include build stats, test results and commit history. Distributed agile teams often use a separate tool capture knowledge about the high level artifacts that are not found in the continuous integration tools.

The aforementioned tools help the distributed agile teams to collaborate and share project related knowledge. Almost all of them send out notification emails when a change takes place. For example, when a user story is assigned or a build succeeds. Such email notification is helpful since it reaches the inbox of people instead of waiting for them to visit the tools. Some of the tools, such as Basecamp, FogBugz etc. also accept incoming emails from people. But none of the the tools interlinks the incoming emails to the user stories unless a user does it manually. As a result, the benefits of a message thread attached to a user story is not readily available. To ensure the message threads are kept attached to the user stories, the users are forced to use the tools. But, similar to email notification, this could be convenient if a developer or a customer could just send an email to the tool and the tool would automatically attach the email to the relevant user story. Taggy attempts to solve this problem by utilizing the planning information from the project management tools to infer the relevance of an user story against an email.

Chapter 3

TAGGY

These chapter is dedicated to the implementation details of Taggy. First the underlying assumptions behind Taggy are discussed. Then, a definition of the adapted agile project context is given. With this background information, a high level architecture of Taggy is explained. Next, the workflow of auto-tagging is discussed. The mathematical and algorithmic details are provided in the similarity computation section. This chapter also includes the details about the software frameworks used to implement Taggy. Finally, an illustrative example is given to explain Taggy in action.

3.1 Assumptions

Taggy is designed based on the following assumptions to auto-tag the emails with user stories:

1. An email is potentially relevant to a user story when:

    - It is sent during the iteration time frame of the user story. Since agile projects are developed in small iterations and the core concentration during an iteration is to deliver the user stories from the iteration backlog, it is highly likely that the email conversation will be about the user stories from current iteration backlog. However, it is also possible to see some conversations about near past or near future iteration backlogs. Such conversation are mainly used to provide post-delivery feedback and collaborate about upcoming work. Taggy uses this assumption to shorten its search space for relevant user stories by filtering out the ones from far past.

- The developers and/or customers of a user story participate in the email. For an example, if Alex (a developer) is working on a user story for Jane (the customer), and Alex writes an email to Jane, they are more likely to discuss about the user story than Peri (another developer) and Jane. However, Peri can always participate in a discussion about Alex's work in an agile team, where open communication is encouraged. But, it is highly unlikely that two people who are neither assigned developers or customers of a user story will write emails about that. This assumption about people's participation in email provides an important context in Taggy's similarity computation.
- There is a minimum degree of text similarity between an email and a user story. An email has text in terms of its subject, body and attachments. Although not explicit, it is likely that such text in the emails will show some relevance to the user stories. This may not be true in all cases, especially if there is a lot of face-to-face communication. However, this is not the case for in distributed projects with huge time zone difference. Taggy computes a text similarity between the email and user story and discards the ones that show very poor match.

2. A web-based project management tool is used to manage the distributed agile project. To automatically link up emails with user stories, Taggy looks into this tool for information about user stories. This assumption is required because if teams only use volatile physical artifacts, such as sticky notes for user stories on a whiteboard, it is not possible to automatically find the user stories. As discussed in literature review, distributed agile teams use a number of different types of such tools.

3. The project management tool captures user stories with its planning information

including a) assigned developers, b) customer and c) iteration timeframe. The presence of this planning information is essential as it serves the meta data that is necessary to auto-tag emails. While it is generally expected that this data be available, it is not required that each and every user story contains all the required planning information. Since Taggy uses context alongside text similarity, having the context helps in making a more informed decision in auto-tagging.

4. Each project has its own email address so that when people are sending emails about a project to someone, they can keep the project's email in the copy. This serves as the input to Taggy for auto-tagging. Also, giving every project a unique email address ensures Taggy can correctly determine the target project for an email. Since most people who use email are already familiar with the CC: feature, this adds little learning curve or communication overhead. It is assumed that indicating the project in CC: serves a convenient input mechanism compared to manually copy-pasting the email contents into a system for every useful email.

5. The subject of an email carries an important clue about its relationship with a user story. Although, subject is just a text similar to body or attachment contents of an email, due to professional etiquette and for the sake of grabbing attention, people write revealing subject while writing emails about projects. Taggy distinguishes the text relevance of the subject from the rest of the email contents based on this assumption.

3.2 The Agile Project Context

Taggy uses two kinds of context information that are available for agile user stories, namely Temporal context and People context. These contexts are defined below:

1. **Temporal Context.** User stories in agile projects are grouped by small iterations

so that a bunch of new user stories are potentially deliverable at the end of each iteration. These iterations are confined within specific start and end dates. This time-box works as the temporal context for a user story. For example, if a user story is developed during Iteration#2, June 1 to June 14, then Taggy assumes people are more likely to write emails about the user story within this period than in the far past or future.

2. **People context.** The people context for a user story is formed by its assigned developers and customers. A user story may have one or more customers, who are mainly responsible for providing the details proactively and also as questions arise during implementation. On the other hand, a user story is broken down into tasks and assigned to developers, testers and other technical team members. Taggy uses all these people relevant to a user story as its people context. For example, if an email is exchanged among the people in a user story, then Taggy puts a higher similarity rank than when the people context is different. The identification of the people context is done based on email addresses.

Since Taggy combines context similarity with text similarity, the above contexts provide more confidence in auto-tagging. However, when some or all of the context information are missing for a user story, Taggy still applies whatever information is available to auto-tag emails with user stories.

3.3 High Level Workflow

Now that the assumptions and definitions of important terms are discussed, the Figure **??** shows the two main steps involved in the process of auto-tagging an email with user stories. Essentially as with most other machine learning techniques, Taggy needs to learn the important parameters before making decisions. Once the learning is done,

Taggy uses the learned parameters to auto-tag emails.

As seen on Figure **??**, the learning process involves several iterations. Since the similarity computation uses multiple components, the learning process needs to address the importance of each component relative to others. To achieve this, first it produces an initial weight for the components and then learns the relative weights based on the training data. The details about learning is discussed later in Section X.

Once learned, Taggy can be used to auto-tag emails with user stories. This process is supposed to be in a live system as illustrated in Figure **??**

As discussed in assumptions, Taggy contains the user stories with planning information. Next, it follows the steps as shown in Figure **??** to complete the auto-tagging of emails against the saved user stories.

1. **Copy emails to project mail address.** This is the input step for Taggy. As discussed in assumptions, Taggy identifies each project by its own email address. So, whenever someone sends an email about a project, they put the "project email" in the CC:. This is the only change in the business process that needs to be implemented by the distributed team. This intake process was successfully adapted in a previous work [**?**]. In case someone forgets to do this while sending the email, it is possible to simply forward the email to "project email" later.

2. **Grab email.** One the project related emails reach the inbox of "project email", Taggy picks up the email. It is common for email servers to allow access via POP or IMAP protocol. Taggy uses POP3 to read all incoming emails, including the attachments, if any. This grabbing runs on a background process, which can be scheduled to check for new emails in desired intervals. However, this email grabbing step can make use of any other email transfer protocol.

3. **Save email.** After grabbing, Taggy saves the email into its database keeping a

link to its project. After this step, even if auto-tagging fails, the email is stored in a shared place with the user stories. This step makes an email available for search and browsing without the need for looking into other's inbox.

4. **Filter.** To auto-tag the email just grabbed, Taggy first reduces its search space by discarding the user stories of a project that were done in the far past or scheduled to be developed in the far future. This filtering process essentially finds the current iteration of the project, if any. Then considers the user stories from the current iteration and its neighboring iterations as potential candidates for auto-tagging the emails.

5. **Compute local similarity.** In the reduced search space, Taggy computes local similarities for people and temporal contexts as well as separate text similarities for subject and body. The details of these local similarity computations are shown in equations x, y, z, w.

6. **Compute global similarity.** Next, for each of the user stories the similarity values from previous step are combined to produce a global similarity score. This computation is done based on the formula presented at equation x. This global similarity assigns a numeric value of the relative relevance between a user story and the email.

7. **Sort.** The user stories are then sorted descendingly according to their global similarity value from previous step. This produces a list of user stories with the potentially most relevant user story at the top.

8. **Pick.** Next, Taggy picks the user stories having global similarity scores above a predefined threshold. Having a threshold ensures Taggy only picks the ones that show sufficient relevance based on its learning. However, this also means, for some

emails no user story may be picked.

9. **Auto-tag.** Finally, Taggy auto-tags the email with the picked user stories from the last step. This step adds database level links between the email and the user stories to be auto-tagged.

However, to auto-tag instant messages, first three of the aforementioned steps differ significantly. The following the steps are used for the intake process of instant messages:

1. **Activate instant message plugin.** Instant message clients often allow the use of plugins. For example, Skype [**?**] has a plugin framework and Taggy has a plugin for Skype. To input the instant messages, one needs to activate the Taggy plugin and select the project under discussion. Then, as the chat messages are exchanged, the plugin sends out the messages to Taggy over a web-based service. Typically the instant message clients provide meta data such as unique identification of a chat session, its individual messages, people and also the time stamp.

2. **Grab instant message.** Taggy exposes a web service for the intake of instant messages. As soon as it finds a chat message from the plugin, it identifies the conversation based on the meta data.

3. **Save instant message.** Taggy extracts out the context from an instant message and saves it in the database. For example, it matches the instant message identifier for people that are already stored in the database against the ones participating on a chat session. Also, it keeps track of the time stamp. As a result, an instant message is stored with a similar people and temporal context as that of an email. Taggy stores the content of different instant messages together as a session that belong to a single conversation as per the instant messenger.

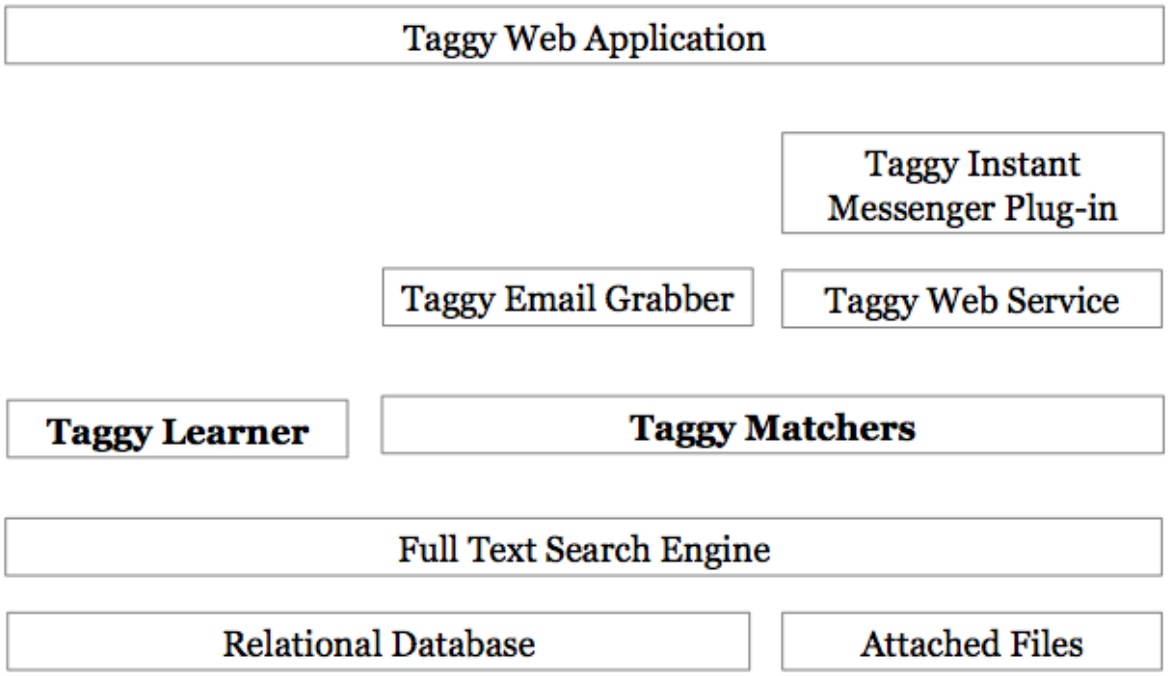

Figure 3.1: Taggy Components

Since instant messages don't capture any subject, Taggy cannot produce a text similarity for this field. As a work around of this limitation, Taggy can be trained differently for instant messages where the contribution of subject similarity is ignored.

On a live system, as like any other machine learning solution, Taggy cannot discard the possibility of a wrong auto-tagging decision. When someone spots such a faulty auto-tagging in Taggy, she can manually correct it.

3.4 Architecture

Taggy is composed of a number of different components to support the workflow as outlined in the previous section. The architecture of Taggy is depicted at Figure 3.1. Next, the details about each component from the figure is discussed following a bottom-up approach.

1. **Relational Database.** This relational database component persists the agile project related information. To address the concern of auto-tagging, this database keeps the following information:

    - People. People information includes the name, email address and instant messenger id, if any. Also, a project vs. people mapping is stored.
    - Iteration. Iterations, defined by start and end dates, are also saved.
    - User story. Each user story may have a title and description. Also, if the story is planned, then information about its iteration and assigned people are stored in the database.
    - Email. The database also stores the subject and content of the emails with a link to sender and recipients if they are found in the people information. If the email is tagged with a user story, then this information is kept in the database as well.
    - Instant Message. Similar to emails, the database stores the instant messages and their tagging information.

2. **Attached Files.** Attached Files is a simple file system storage where all attachments are kept. An attachment may be part of a user story definition or can be extracted from an email.

3. **Full Text Search Engine.** This component produces full text index from contents found in user stories, emails and their associated attachments. This is a third party component that can handle synonyms and language specific stems. The attached files are only indexed for full text search if it is meaningful, for example image files are not indexed where PDF file contents are.

4. **Taggy Matchers.** This is the core component of Taggy that produces the auto-tagging, i.e. the similarity related computations are done inside this component. There are two matchers inside Taggy for now: email matcher and instant message matcher. These two matchers share some common computation logic. However, instant messages don't have subjects as found in emails and as a result, the two matchers use different relative weights and minimum threshold similarity scores to auto-tag. The matchers depend on the full text search engine and the relational database to lookup for required data.

5. **Taggy Learner.** As discussed before, Taggy requires training to learn the different parameters. This component takes care of the learning process. During learning the Learner uses the Matcher to auto-tag an email and depending on the outcome, it may learn the parameters of interest. Once the learning is complete, the learner sets the relative weights to be used in the Matcher. So, the core part of auto-tagger is a mix of its Learner and Matcher components.

6. **Taggy Email Grabber.** The Email Grabber component works as a background process that periodically checks for incoming emails in any of the project emails. If it finds one, it reads the email, saves a copy in the relational database as well as downloads the attachments into Attached Files. Next, the text contents are all indexed through the Full Text Search Engine. With this data persisted, it invokes the Matcher component to auto-tag the email against the relevant user stories. This handles the email intake process to Taggy.

7. **Taggy Web Service.** Taggy exposes a web-service so that an instant messenger plugin can push instant messages into Taggy.

8. **Taggy Instant Messenger Plug-in.** This component is installed as a plug in to

an instant messenger. Once activated, this component sends instant messages to the Web Service Component.

9. **Taggy Web Application.** The web application provides interfaces for manipulating everything in Taggy. For example, one can browse a user story and see all related emails and instant messages or vice versa. Also, one can search through all the contents, including the attachments. In a nutshell, this is a light-weight agile project management tool with the additional feature that it auto-tags emails and instant messages.

3.5 Similarity Computation

Taggy uses a machine learning technique called Case Based Reasoning to find out relevant user stories against an email or instant message. A CBR system uses a library of existing cases to match against a new case. For example, Taggy has a library of existing user stories, these are treated as cases in the CBR system. Each case, or user story, as described above is defined by several attributes, such as, its title, description, assigned developers, customers, iteration start date and end date. Also, the data types of the attributes are different that include numeric date time values, symbols for emails and free text values. This library of cases is not exactly similar to a new case, an email, since the attributes are different. However, it has been discussed that it is possible to match some of the email attributes against the user story attributes. This action is performed inside the CBR system of Taggy to compute the similarity. Figure 3.2 shows a high level view of the CBR system:

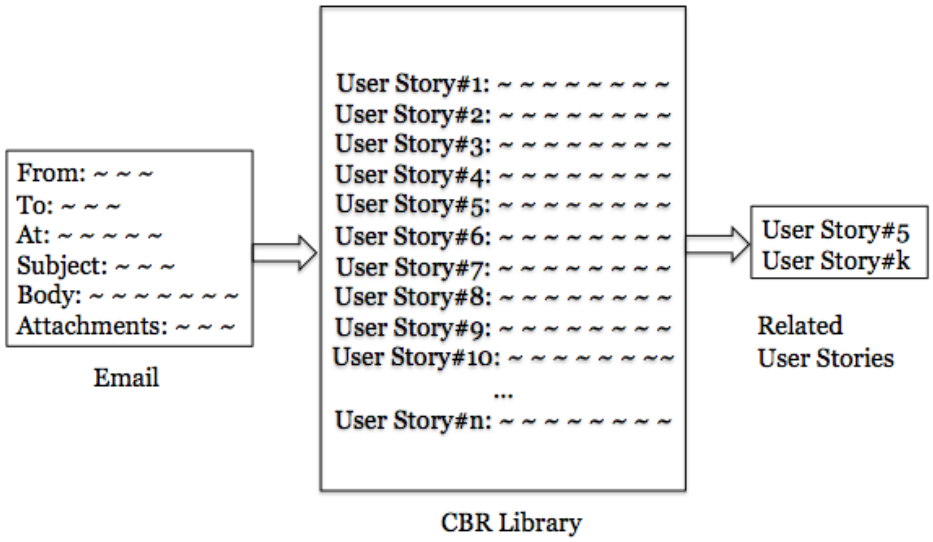

Figure 3.2: CBR System Input and Output

3.5.1 Local-Global Principle

The actual computation inside the CBR system is done following the local-global principle[**?**] where it takes a two step approach, **i)local similarity computation and ii) global similarity computation.** Local similarity computation is confined to the similarity between only one attribute of two entities. For example, here a local similarity may be the similarity between the date of an email and the iteration start date of the user story. These local similarities are calculated in isolation from the rest of the attributes. So, the local similarity of email date and user story start date is not impacted by their local similarity for people, subject or description.

Table 3.1 shows the adapted mapping of email attributes to user story attributes. According to this mapping, the people in email are matched against the people in user stories, so no distinction is made between the sender or recipient of an email. Similarly developers and customers of a user story are not distinguished for similarity computation. The temporal similarity again is a result of comparing the email date against a range defined by the start and end date of the iteration. And as it was stated in the assumptions,

Table 3.1: Mapping Between Email and User Story Attributes

Local Similarity	**Email**	**User Story**	**Type**
People	Sender, Recipients	Developer or Customer	Symbol Array
Temporal	Email Date	Iteration start and end date	Numeric Date
Subject	Email subject	Title, Description and Attachments	Free text
Body	Email Body, attachments	Title, Description and Attachments	Free text

Source: *primary*

subject similarity is distinguished from the body similarity to look for important clues in the email subjects. However, in both cases the comparison is done against all the text found in a user story by looking into its title, description and attachments. This is done because, the subject or body of the mail can contain similar text to any part of the user story.

For instant messages, this table remains same for all but the Subject similarity row, since a subject is absent in such cases. So, instant message matching requires one less local similarity measure.

On the other hand, global similarity combines the local similarities and produces a single similarity value between two entities. In this case, the global similarity combines the local similarities for people, temporal, subject and body of an email against the appropriate attributes of the user story. The combination is performed using a weighted sum of the local similarities, where the relative weight for each component is learned during the training phase.

3.5.2 Similarity Function

Using the aforementioned local-global principle, we adapted different equations to compute the local and global similarities as follows:

$$S_{Temporal} = \begin{cases} 1 & \text{iteration start} \leq \text{email date} \leq \text{iteration end} \\ 0 & \text{if email date is within the buffer of the story's iteration} \\ -1 & \text{else} \end{cases} \tag{3.1}$$

S_{Temporal} in Equation 3.1 denotes the Temporal local similarity between an email and a user story. This equation can be interpreted as follows: if the email is sent while the user story is in development, during its iteration, then they show highest temporal similarity. However, it is also likely that an email is sent in the near past or near future of an iteration, this nearness is designated by the buffer. This buffer is set to one iteration length. So, if an email is sent in the immediate past or future iteration of a user story, the temporal similarity is supposed to be 0. Finally, a -1 value for the rest ensures emails from far past and far future are treated as highly distant in terms of temporal context. This interpretation is in alignment with the provided assumptions.

$$S_{People} = \begin{cases} 1 & \text{email people equals user story people} \\ \#of\ common\ people/\#of\ user\ story\ people & \text{Email and user story has some common p} \\ -1 & \text{else} \end{cases} \tag{3.2}$$

S_{People} in Equation 3.2 denotes the People local similarity between an email and a user story. This function can be interpreted as follows: in an email includes all of the people in the user story, it is highly similar in people context. Otherwise if only a few people are common, the similarity is pro-rated accordingly. However, when an email does not

include any of the people in the user story, then it is marked as highly distant from the user story in terms of people context. In other words, this equation produces a numeric value of user story people's participation in the emails.

$$S_{Subject} = [0, 1], \text{ Free text similarity score (See below)} \tag{3.3}$$

$$S_{Body} = [0, 1], \text{ Free text similarity score (See below)} \tag{3.4}$$

S_{Subject} and S_{Body} represents the free text similarity score for the two attributes. The values can be anything between 0 and 1 as shown in Equation 3.3 and Equation 3.4. Computing the textual similarity between two free format texts is in itself a research topic and beyond the scope of this thesis. However, Taggy used an industry standard open-source full text search engine to produce this score. The search engine uses Vector Space Model [**?**] to compute the similarity between two free text documents. This model first defines the index of a free text document in terms of a vector that captures the frequency and relative weight of each term in the document. Next, an inner product of two such vectors is used to produce their similarity. Also, to compare the similarity of a document against a library of documents, the similarity results are normalized for the length of the documents. This approach is highly scalable since the similarity computation of two different documents is turned into simple vector computation. Also, the generation of the vectors can benefit from using synonyms, stop words and other language specific stems. Very high traffic applications have used this approach to facilitate full text search. For example, Twitter, serves 12000 searches/second, adapted this at the time of writing this thesis [**?**].

The choice of the above equations are based on the experience from looking into the data. However, since the software is trained to learn the relative weights of these

equations, the impact of these equations are likely to be weight adjusted so that the global similarity computation minimizes the possibility of a wrong decision.

Next, the global similarity function combines the local similarities from the above equations using the following formula:

$$S_{Global} = (W_{Temporal} * S_{Temporal} + W_{People} * S_{People} + W_{Subject} * S_{Subject} + W_{Body} * S_{Body}) / (W_{Temporal} + W_{People} \quad (3.5)$$

where,

$$W_{Temporal} = \text{relative weight of temporal similarity} \quad (3.6)$$

$$W_{People} = \text{relative weight of people similarity} \quad (3.7)$$

$$W_{Subject} = \text{relative weight of subject similarity} \quad (3.8)$$

$$W_{Body} = \text{relative weight of body similarity} \quad (3.9)$$

As Equation 3.5 shows, the global similarity is a weighed sum of the local similarity values. But the relative weights are not known a priori. Instead, Taggy learns the relative weights for the different components so that it can potentially reflect the relative importance of each of the component based on a training data set. Using weighted sum provides transparency about the decision making logic inside Taggy.

3.5.3 Learning Relative Weights

Taggy uses a Reinforcement Learning approach to learn the relative weights of the different components to compute the global similarity [**?**]. The learning algorithm is designed

to adapt the relative weights of the local similarity measures based on the feedback from its decision being correct or wrong.

Since the local similarity measures participate in the process of global similarity computation, in case of a correct decision, the matching local similarity measures get rewarded by getting more weight while the contradicting ones are punished by lowering their relative weight. For example, if a decision is correct and the people similarity score is positive, then the relative weight of people similarity score, as seen on Equation 3.7 is increased. On the other hand, if the decision is correct but the people similarity score is negative, then it is punished by lowering the weight. So, depending on the decision of an individual component and the final outcome, the relative weights are adjusted accordingly. This process continues unless all training data are exhausted or the adaptation converges.

However, it is important to initialize the relative weights of the local similarity measures based on a meaningful foundation so that the entry point is not absolutely random. This is done by running the training data for each local similarity while keeping the other in isolation. For example, trying to auto-tag emails with user stories solely based on their people similarity and doing the same for other local similarity components. Once this is done, the number of correct decisions made by each component provides a rough estimate of its relative importance over another component. This number of correct decisions are normalized to produce the initial relative weights of the four components, namely, people, temporal, subject and body similarity measures.

Once the initial weights are found, the following algorithm is used to implement the reward-punishment scheme of the reinforcement learning approach:

```
date_weight     = initial_date_weigth
people_weight   = initial_people_weight
subject_weight  = initial_subject_weight
```

```
body_weight     = initial_body_weight

for all_training_emails do |email|
   result = find_most_similar_user_stories(email)
   is_correct = result.guessed_stories == email.actual_stories

 #Adjust temporal similarity weight
   is_date_similar = result.date_weight > date_threshold

   if (is_correct and is_date_similar) or
      (!is_correct and !is_date_similar) then
    date_weight = reward(date_weight)
   else
    date_weight = punish(date_weight)
   end

   #do the same for people, subject and body similarity
.
.
.

   end
end
```

This algorithm shows the reward-punishment in action for the temporal similarity weight. This essentially rewards the local similarity weight of a component that contributes in making a correct decision while punishes the one that influences a wrong decision. This

reward-punishment continues unless all training emails are seen. Also, it is possible to stop if this converges to a point when adding new emails do not alter the relative weights. However, as with most machine learning techniques, the usefulness of this learning is related to the quality and quantity of the training data. As shown before, the Taggy Learners component implements this algorithm. Once the learning is done, relative weights are tuned to auto-tag emails with user stories. This same algorithm can be used to learn relative weights for instant messages, where the subject is missing.

The choice of this reward-punishment scheme makes it transparent, as one can easily follow the changes in relative weights based on the logic. Other machine learning approaches such as different variations of back propagation algorithm could also be utilized instead of this one. However, this approach was chosen in Taggy because of its expressiveness and lucidity.

3.5.4 Email Matching

3.5.5 Instant Message Matching

3.6 Implementation Details

3.7 An Illustrative Example

Chapter 4

EMPIRICAL EVALUATION

4.1 Evaluation Approach

4.1.1 Accuracy

4.1.2 Statistical Relevance

4.2 Evaluation Results

4.2.1 Data set# 1: ScrumPad

4.2.2 Data set# 2: IBM Jazz Rational Team Concert

4.3 Limitations

Chapter 5

PRELIMINARY QUALITATIVE EVALUATION

5.1 Goals

5.2 Methodology

5.3 Study Setup

5.4 Findings

5.5 Limitations

Chapter 6

DISCUSSION

6.1 Handling Attachments

6.2 Selecting Threshold Similarity Score

6.3 The Impact of Incorrect Tagging

6.4 Dealing with Absence of Necessary Context

6.5 Project Specific vs. Project Agnostic Learning

6.6 Dealing with Undesired Input

6.7 Limitations of Taggy

Chapter 7

CONCLUSION

7.1 Research Goals Addressed

7.2 Future Work