

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

© S. M. Sohan 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

Dr. Jonathan P. Sillito
Department of Computer Science

Dr. Xin Wang
Department of Geomatics
Engineering

Date

Abstract

In distributed agile teams, people often use email and instant message as knowledge sharing mediums to clarify the project requirements or user stories. Knowledge about the project included in these private mediums is easily lost when recipients leave the project. This thesis outlines the opportunities and challenges of retaining this knowledge, followed by an overview of the previous work in this area. Next, it presents a new machine learning based auto-tagging technique and its implementation, Taggy, that does the following:

1. Automatically captures an email when a project specific email address is added to it's recipients.
2. Automatically captures an instant message using a plug-in.
3. Finds the potentially relevant user stories for the email and instant message.
4. Tags the email and instant message with the relevant user stories.

The details of this technique is discussed that includes a high level architecture to low level algorithmic constructs. In a quantitative evaluation, Taggy could correctly auto-tag 76% of 4,757 emails from 9 industry based agile teams. Also, in a preliminary user study, the participants have provided encouraging feedback about it's potential use as a lightweight knowledge center. The key limitations of the two evaluations are identified too. Next, this thesis provides insights into the key challenges about dealing with some of the unexpected cases in the data, finding a right balance between auto-tagging accuracy and its coverage, and the trade-offs between optimizing the auto-tagger for a single project vs. all available projects. Based on the design, implementation and evaluation of the auto-tagger, the key shortcomings are identified and grouped into two directions for future work, i) additional feature and ii) improved evaluation.

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.

Publication

The core ideas of this thesis appeared in the following publication:

S. M. Sohan, Michael M. Richter, and Frank Maurer. Auto-tagging emails with user stories using project context. In Agile Processes in Software Engineering and Extreme Programming, volume 48 of Lecture Notes in Business Information Processing, pages 103 - 116. Springer Berlin Heidelberg, 2010.

Table of Contents

Abstract	ii
Acknowledgements	iii
Publication	iv
Table of Contents	v
List of Tables	vii
List of Figures	viii
1 Introduction	1
1.1 User Stories - An Overview	3
1.2 Distributed Agile Projects	6
1.3 Research Motivation	8
1.4 Research Problem	11
1.5 Research Goals	14
1.6 Key Contributions	15
2 Related Work	16
2.1 Communication in Distributed Teams	16
2.2 Capturing Knowledge from Emails	18
2.3 Wiki-Based Knowledge Sharing	22
2.4 Context-Based Knowledge Management	25
2.5 Review of Existing Tool Support for Knowledge Sharing	29
2.6 Summary	32
3 Taggy	33
3.1 Assumptions	33
3.2 The Agile Project Context	36
3.3 High Level Workflow	37
3.4 Architecture	41
3.5 Similarity Computation	44
3.5.1 Local-Global Principle	45
3.5.2 Similarity Function	47
3.5.3 Learning Relative Weights	50
3.5.4 Email Matching	52
3.5.5 Instant Message Matching	56
3.6 Implementation Details	57
3.7 An Illustrative Example	59
4 Quantitative Evaluation	63
4.1 Evaluation Approach	63
4.1.1 Accuracy	65
4.1.2 Statistical Relevance	65
4.2 Evaluation Results	66
4.2.1 Data set# 1: ScrumPad	66
4.2.2 Data set# 2: IBM Jazz	72
4.3 Limitations	74

5	Preliminary Qualitative Evaluation	75
5.1	Participants	75
5.2	Data Collection & Analysis	76
5.3	Findings	77
5.4	Limitations	81
6	Discussion	83
6.1	Dealing with Absence of Necessary Context	83
6.2	The Impact of Incorrect Tagging	84
6.3	Selecting a Threshold Similarity Score	85
6.4	Handling Attachments and Full-Text Search	85
6.5	Project Specific vs. Project Agnostic Learning	86
6.6	Handling Email Replies	86
6.7	Limitations of Taggy	87
7	Conclusion	89
7.1	Research Goals Addressed	89
7.2	Future Work	92
	7.2.1 Extensions with New Features	92
	7.2.2 Improvements through Evaluation	93
8	Ethics Approval	95
	Bibliography	97

List of Tables

3.1	Mapping Between Email and User Story Attributes	46
3.2	An Example Product Backlog	59
3.3	Similarity Scores	61
4.1	Scrumpad Data Set	68
4.2	Evaluation Using Scrumpad Data Set	70
4.3	IBM Jazz Data Set	73
4.4	Evaluation Using IBM Jazz Data Set	73
5.1	Taggy User Study Participants	76

List of Figures

1.1	A Scrum Wall. Source: http://www.xqa.com.ar	5
1.2	A Virtual Scrum Wall. Source: http://www.scrumpad.com	6
1.3	An Example Agile Outsourcing Model. Source: [1]	7
1.4	Jigsaw Puzzle Visualizing the Auto-Tagging Problem	14
2.1	Use of Emails for Different Purposes. Source: [2]	19
2.2	A Wiki-Based Document Structure for Requirements Engineering. Source: [3]	22
2.3	Mylyn Eclipse Plugin - Bugzilla Integration. Source: [4]	27
2.4	Mylyn Eclipse Plugin - Setting source code context of a bug. Source: [4]	28
2.5	Dashboard in ScrumPad, a Commercial Agile Project Management Tool. Source: [5]	29
3.1	Taggy High Level Workflow	37
3.2	Email Auto-tagging Workflow	38
3.3	Taggy Instant Message Intake Process	40
3.4	Taggy Components	42
3.5	CBR System Input and Output	45
3.6	Taggy Screenshot - Showing User Story and Related Emails	53
3.7	Taggy Screenshot - Showing Email and Related User Stories	54
3.8	Taggy Implementation Details by Component	57
4.1	Taggy Empirical Evaluation Steps	64
4.2	A ScrumPad Message Linked to a User Story	67
4.3	Impact of Training on Relative Weights	69
8.1	Ethics Approval	96

Chapter 1

Introduction

In this thesis I have presented a machine learning based technique to build an organic knowledge center for distributed agile teams. The knowledge is captured from emails and instant messages and automatically tagged to the relevant project requirements or user stories. I have discussed about a prototype implementation of the technique named Taggy. Taggy has correctly found the relevant user story for 76% of 4,745 emails from 9 agile project teams. I have also conducted a user study and found encouraging feedback about the technique and its implementation. In a nutshell, this thesis documents the details of my auto-tagging technique, its novelty compared to some of the existing solutions, a prototype implementation and the evaluation results based on real-world data as well as its qualitative feedbacks.

The process of agile software engineering is a collaborative approach which incrementally delivers software in small iterations, preferably of two to four weeks. Agile teams commonly use “user stories”, a small description of a desired feature in everyday language, to capture the software’s 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 on an ongoing basis among the people involved in a project. For example, as the engineering team starts 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 [6, 7, 8, 9]. In collocated setups, where the team members 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 becomes infeasible. To mitigate this difficulties in communication, people use alternatives such as phone calls, emails, instant messaging, wikis, web forums and so on. So, in such setups communication is often text based instead of oral and direct knowledge sharing. In this sense, distribution makes it easy to capture the knowledge about the user stories 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 in the sense that they remain in the inbox unless deleted, one can use this knowledge in future. But, email is also a personal tool and it is hard to transfer a selection of project related emails from the email inbox in a reusable format. The same is true for instant messages as well. So, when someone leaves the team, it becomes hard to access that knowledge if needed later down the road. To retain this useful information, this thesis explored a machine learning based technique. As a proof of concept, the technique is also implemented and evaluated through a prototypical 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 Case Based Reasoning (CBR). For auto-tagging, a CBR system is designed to look at the text, people, and temporal similarity between emails and user stories. The evaluation results show that a well trained software such as Taggy can auto-tag emails with user stories. Also, we found that combining contextual 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 effort required of human users, 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 stories and distributed agile projects to develop an 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 an application. Cohn [10] 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 user stories. Jeffries [11] and Davies[12] also emphasize the role of conversation as a key component of user stories.

Wake, author of [13], suggested a popular acronym, INVEST, that defines good stories being Independent, Negotiable, Valuable to users, Estimable, Small and Testable. However, to hold all these characteristics while remaining 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 the start of every iteration (typically two to four weeks), the team picks the top user stories from the product backlog to create an “Iteration Backlog” with the target to deliver the functionality for each of these user stories at the end of the iteration. A user story is supposed to be limited in scope so that it can be delivered within a single iteration. However, the details about the user stories are laid out on an ongoing basis during the iteration through customer collaboration and feedback.



Figure 1.1: A Scrum Wall. Source: <http://www.xqa.com.ar>

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 team members collaborate to fine tune the user story details. Collocated teams often use sticky pads on a team wall or whiteboard to visualize the iteration backlog and track progress. Figure 1.1 shows a “Scrum Wall”, where user stories are grouped based on their development status.

However, distributed agile teams often use a virtual team walls in place of a physical wall so that it is usable across several geographical locations. Figure 1.2 shows such a virtual wall taken from ScrumPad.

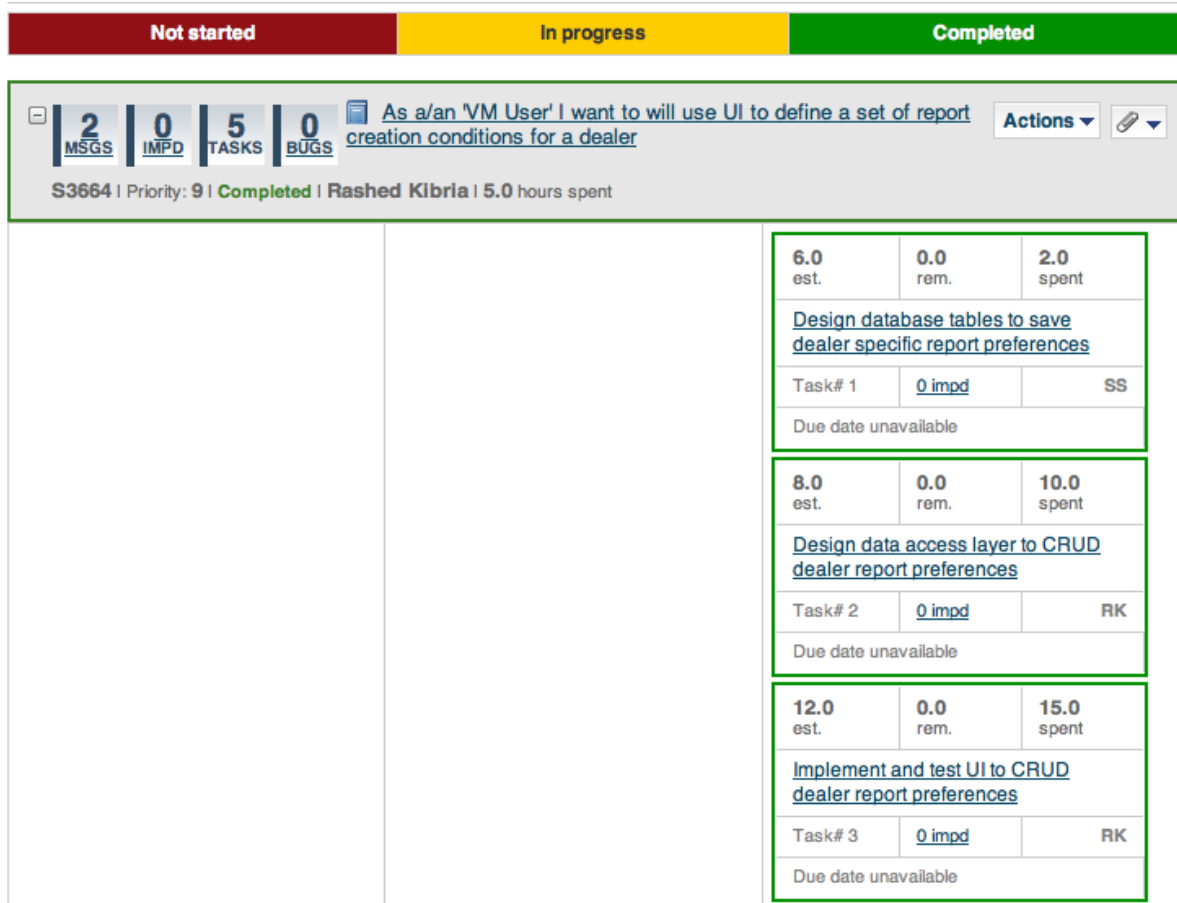


Figure 1.2: A Virtual Scrum Wall. Source: <http://www.scrumpad.com>

1.2 Distributed Agile Projects

The members of a distributed agile project work from different geographical locations using agile principles. In practice, the distribution can take several models such as the following three outlined by Braithwaite et al [14]:

1. Agile outsourcing - the development team is offshore to the customer.
2. Agile dispersed development - developers are spread over different locations, such as open source projects.
3. Distributed agile development - customers are distributed and one development team is distributed evenly so that each customer has a part

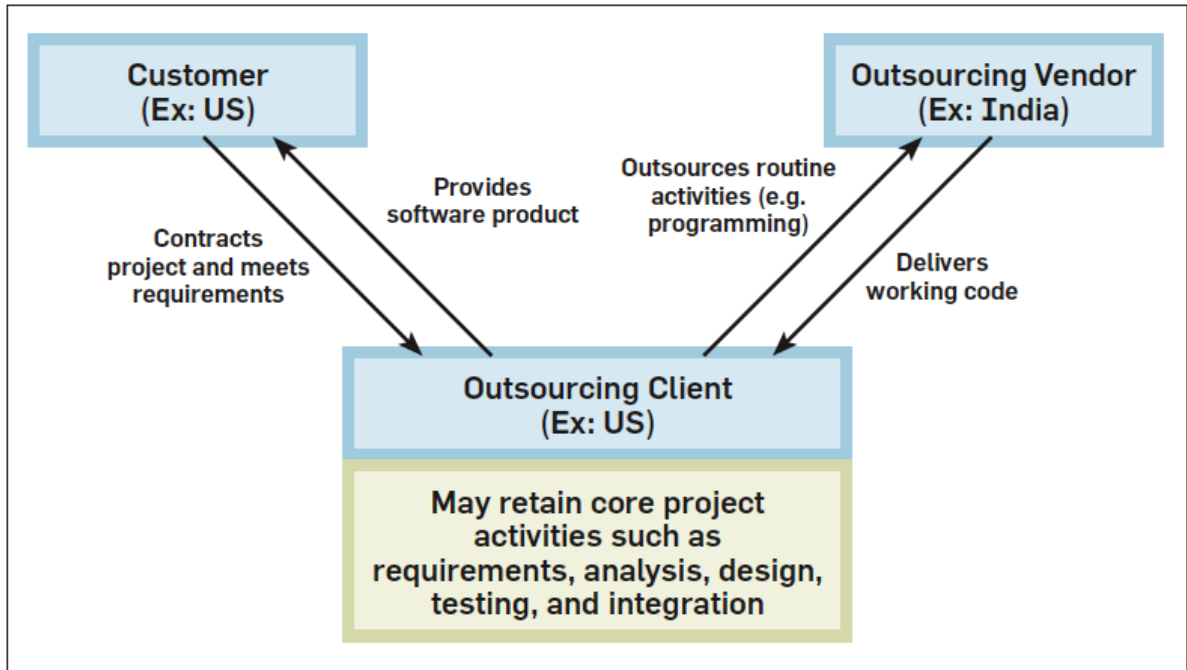


Figure 1.3: An Example Agile Outsourcing Model. Source: [1]

of the team to closely work with.

Again, the geographical distribution can be as near as office across the road or as far as the opposite side of the world. For example, a team may work for a customer within 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, wikis, forums and online project management tools to uncover important details about the user stories. Scrum and XP are two popular agile specifications [8, 7].

Figure 1.3 shows an example of an outsourced agile project taken from [1]. As seen in this example, a distributed agile project may employ people working on different roles across the globe.

Cost savings has been identified as one of the principal deciding factors for distributing projects across the globe [15]. However, projects also go distributed to utilize global talent supply and local knowledge [16, 1]. 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 model [16]. Similarly, Yahoo! had success with distributed teams that valued people over process and exercised the right information sharing formats and timing [17].

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 such as ScrumPad, VersionOne etc. as well as open-source [5, 18] such as XPLanner, Trac etc.[19, 20] Such tools help people working remotely to have a single backlog around which to collaborate.

1.3 Research Motivation

The existing literature about collaboration in distributed agile software projects suggests the use of various electronic mediums to compensate for missing face-to-face communication. Because of time zone differences, such teams often use asynchronous communication tools such as email and wikis in addition to synchronous ones such as telephone calls, instant messaging, etc. For example, Robarts [15] found that teams would follow up with a written email after they had a telephone meeting to ensure the discussion was clearly understood. Moreover, of all the available written communication tools, email has been regarded as the most-widely used by several former studies [21, 22]. Additionally, even in collocated settings, people often use emails to communicate as it doesn’t require an immediate response from the recipient or the overheads of scheduled meeting.

Although having multiple communication channels adds flexibility to collaboration, it also means that knowledge gets fragmented across different channels. This fragmented knowledge may be spread over the project management tool, wiki, emails, instant messages and other places. Basili et al. provided the following list of experiences related to

the lack of access to available information [23]:

- An employee left with short notice. The organization lost all of its experience in a certain area and tries to recover it, but it doesn't even know what experience was lost.
- A consultant spends three weeks developing a course that already exists because he doesn't know that it was done before.
- Someone repeats a \$35,000 mistake for which there is a simple solution.
- A consultant gave a customer a promise, but is now busy with other work. No one else knows about his promise so it doesn't happen.
- An employee learned a lot during a project, but has no time for packaging and dissemination so the knowledge cannot be leveraged.
- A new employee is hired, but is for a long time considered a burden instead of help because he needs detailed support from his coworkers, who do not have sufficient time to help him.
- (continued...)

For agile teams, this situation can be improved with a useful knowledge management solution. To retain the important details about the user stories, one needs to combine these fragmented pieces into a centralized repository. But looking into email inboxes of colleagues to do this may violate their privacy, if it is even available in the first place. Even worse, when a teammate leaves, her emails are lost 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 tagging them with the relevant user stories would be a time consuming task. The following user story and email collaboration shows an example of knowledge fragmentation:

Available in Project Management Tool

User Story

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 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 some of the team members only look at the user story, they will find partial information about it. So, the whole conversion of “region code” may be confusing to them. However, when they can see the emails along with the user story, they 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 may 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 an agile 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, in 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 on the team will be able to easily find the necessary information about a user story when required.

From a team member's 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 an inbox and putting them into a system and linking with a user story. This manual process is cumbersome and time consuming. However, if an application can do this automatically, then it may work as a knowledge-base even after someone leaves the team. Such an application needs to be unobtrusive 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 it ask developers to manually enter their chat logs into the system.

The core technical challenge in devising such a software system 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 proper token 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 email clients including the ones on mobile phones is cumbersome. Also, it imposes a behavioral change or a learning curve for the people at the business end.

To find out the implicit relation between a free-format email and a use story, an application needs to handle similarity between two texts, which is a standard problem. Moreover, 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 exists underneath a written communication. Using context

information has the potential to increase this accuracy of user story-email matching. So, a software system needs to be able to combine both the text and context similarity to auto-tag emails with user stories.

To account for the context, one needs to use the associated meta information that is available. The context of an email and a user story are not the same. 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 contextual details meaningfully so that each component gets an appropriate weighting in the relevance computation. So, it is important to have a formula that produces a similarity ranking which considers both the fuzzy text similarity and the associated context relevance.

The 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 technique to automatically read and relate emails and instant messages with user stories based on agile project context. A proof of concept tool implementation is required to test the technique. The tool needs to minimize human efforts in doing so. Another complementary goal is to evaluate the tool using real world data. The technique and its prototypical implementation are discussed in Chapters 3. The evaluation of the tool is presented in Chapter 4.



Figure 1.4: Jigsaw Puzzle Visualizing the Auto-Tagging Problem

Figure 1.4 visualizes the research goal as it shows the fragmented knowledge from project management tools and email inboxes to be combined in a jigsaw puzzle. The

target is to automatically solve this puzzle so that each user story is surrounded by it's related emails and vice versa.

This research also aimed to investigate the difficulties involved 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 are expected to provide a foundation for next level of research in this area.

1.6 Key Contributions

Results from this research have been published and presented at XP 2010 [24].

The key contribution of this research is that it demonstrates a novel approach to building a knowledge-base for distributed agile teams from emails, instant messages and user stories. This approach has a similar motivation to that found in some previous work, such as clustering email archives[25], tagging forum messages with source code change set[26, 27] and using wikis to share knowledge. However, this research shows a machine learning approach to conglomerate fragmented knowledge from emails, instant messages and project management tools which minimizes human efforts.

Another key contribution is that 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. I have used Case Based Reasoning (CBR) for auto-tagging. Similar to this auto-tagging solution, other CBR systems concerning different entities may also utilize this technique.

Chapter 2

Related Work

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 [6] puts heavy emphasis on customer collaboration. Understandably in distributed projects, geographic and temporal distances create communication challenges not seen in a collocated setup. Existing research focuses on the use and effectiveness of the different communication channels that are used in distributed projects.

Thissen et al. points out that for a distributed team to become successful, it must use appropriate tools to ensure easy flow of information [28]. Lanubile provided a categorization of such collaboration tools [29] 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, they identified email 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 instant message over telephone calls for meeting between two sub-teams at two different locations due to language barriers [30]. 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[29]. Here is an opinion from a customer of their case study about the selection of a medium for 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 acceptance criteria for user stories in the iteration backlog [31]. Also, quick feedback from 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[22]. Although the target was to use groupware for such activities so that the knowledge could be easily retained and 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 [2]. They also found that clarification related emails were the most frequently exchanged of all these categories. Figure 2.1 shows their findings for twelve iterations of a project.

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 into meaningful groups. Also, there has been work

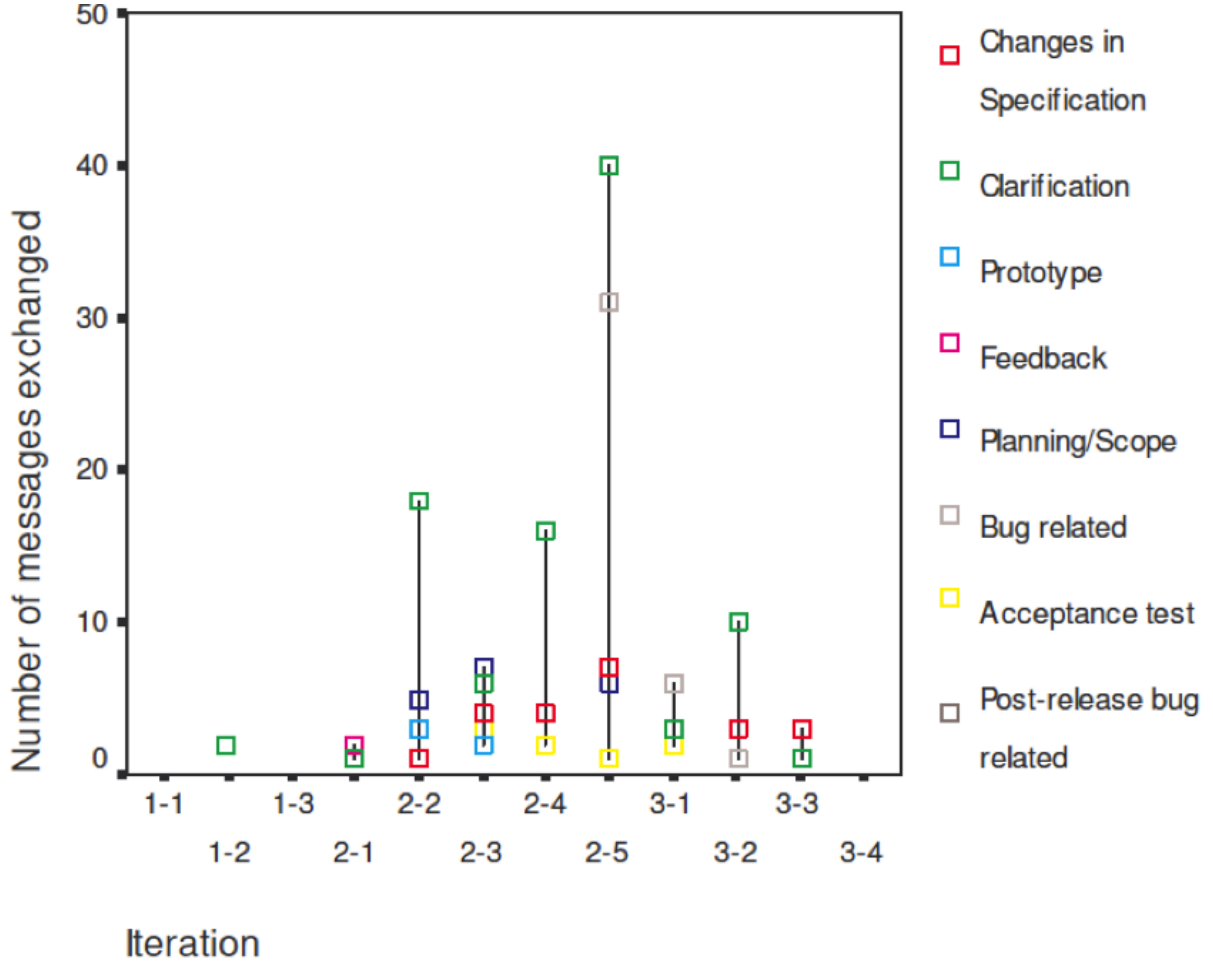


Figure 2.1: Use of Emails for Different Purposes. Source: [2]

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 [32]. TeamInfo let people to use the 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 tries 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, the users of TeamInfo faced challenges to come up with the right 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 requires significant experience as the rules were expressed using a new declarative language. TeamInfo helped its users in retaining and sharing knowledge. Taggy is similar to TeamInfo in that it uses the same CC: feature of emails to get hold of the knowledge. But the key difference is, instead of asking the users to provide rules for classifying emails, Taggy tries to tag emails with relevant user stories. 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 as well.

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 [26]. 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 lookup 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 important patterns. For example, Medynskiy provided a multi-modal visualization of email, instant message and CVS commit logs from the Python project[25]. They mined the textual information to extract meaningful groups and linked among the groups to content contributors. 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 archived data sources, Taggy builds the knowledge-base as it is created.

Another work, The Experience Factory implementation introduces several tools for capturing emails and instant messages [23]. For example, they capture Frequently Asked Questions with expert answers, focused chat sessions, email and project related presentations. However, the management activity in Experience Factory requires human efforts to collect the email contents into the system as well as keep them updated. Taggy addresses this issue by not only automatically grabbing the emails but also using a machine learning approach to tag emails with user stories.

Software requirements traceability helps developers to understand a requirement based on its underlying reasoning [33]. However, for distributed agile projects, this is hard since knowledge is dispersed. Tagging emails with user stories could improve the traceability in such cases. To attain this goal, the approach of Taggy is in alignment with the aforementioned research into 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.

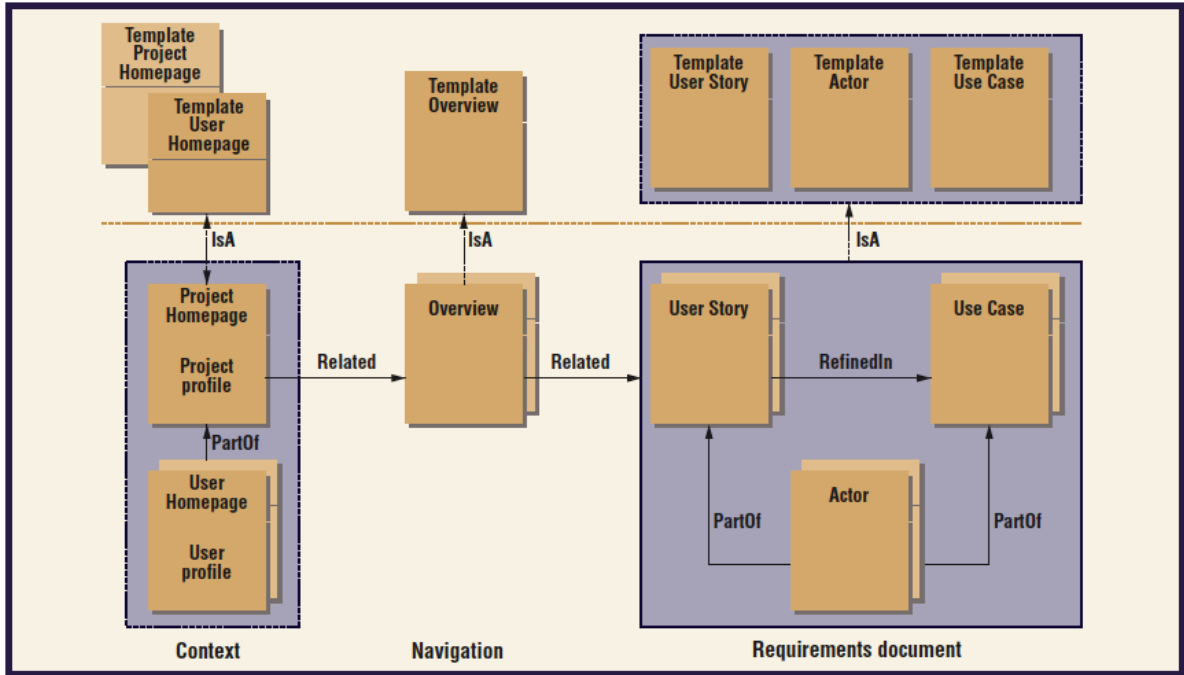


Figure 2.2: A Wiki-Based Document Structure for Requirements Engineering. Source: [3]

2.3 Wiki-Based Knowledge Sharing

The overwhelming success of Wikipedia [34] is a result of collaborative editing. Being inspired by this, several researchers explored the potential of using Wiki for collaborative software knowledge management. For example, Decker et al. have found successful requirement capturing and stakeholder participation through the wiki platform[3]. They proposed a document structure or template for using wikis 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. Figure 2.2 shows their suggested wiki document structure for requirements engineering.

Decker et al. also propose 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 [35]. In this regard they introduced “Wikilogy”, 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, and the like. 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 Collaborative Semantic Web Portal Prototype. This tool utilizes wiki platform for collaborative knowledge acquisition to be used in agile project management [36]. Their solution added access control to the wiki pages and also several indicators for every page such as number of contributors, the creator and so on. 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 incorporates several interesting techniques [37]. 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 content on the wiki and ensure the content is updated accordingly. This may be a barrier, as it requires dedicating considerable human effort to a cause that has

lower immediate business value compared to other tasks, such as developing and testing the software.

Chau et al. studied the contribution to a software wiki made by people working on different roles [38]. They observed the use of MASE, a wiki-based knowledge sharing system, in a medium-sized software company. They discovered a very high usage of MASE (that exceeds 90%) 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 flexibility without a steep 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 [39]. For example, Young et al. mentioned the use of wiki for automated acceptance tests so that the continuous integration system could execute the tests[40]. However, such tabulation of acceptance tests often does not capture underlying knowledge, which may be necessary to understand a user story.

While wiki can work as a knowledge sharing tool, the addition of email discussions offer several unique benefits. First, email is a general purpose communication tool, so most people are already familiar with this. Wikis are 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 easier to follow a discussion than to find the

latest changes to a wiki page. As Chau et al. pointed, wikis and similar centralized knowledge capturing approaches often employ 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[38]. Capturing knowledge from emails and linking it to user stories will complement the collaborative editing benefits of wiki. So, it will add the useful details that are already discussed from emails to any knowledge from the wikis.

2.4 Context-Based Knowledge Management

Maalej et al. provided a context-based solution for lightweight knowledge sharing in distributed software projects [41]. 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 a 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 knowledge. 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 effort from the provider without much immediate benefit.

To minimize this burden, they proposed the use of additional semantic information or an additional ontology of 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 artifacts, 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 [42]. 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 is 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 a 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[43] and Mylyn[4]. 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 through relevant source code for a task. Mylyn also has integrations with several task and bug tracking tools. Figure 2.3 shows a screenshot

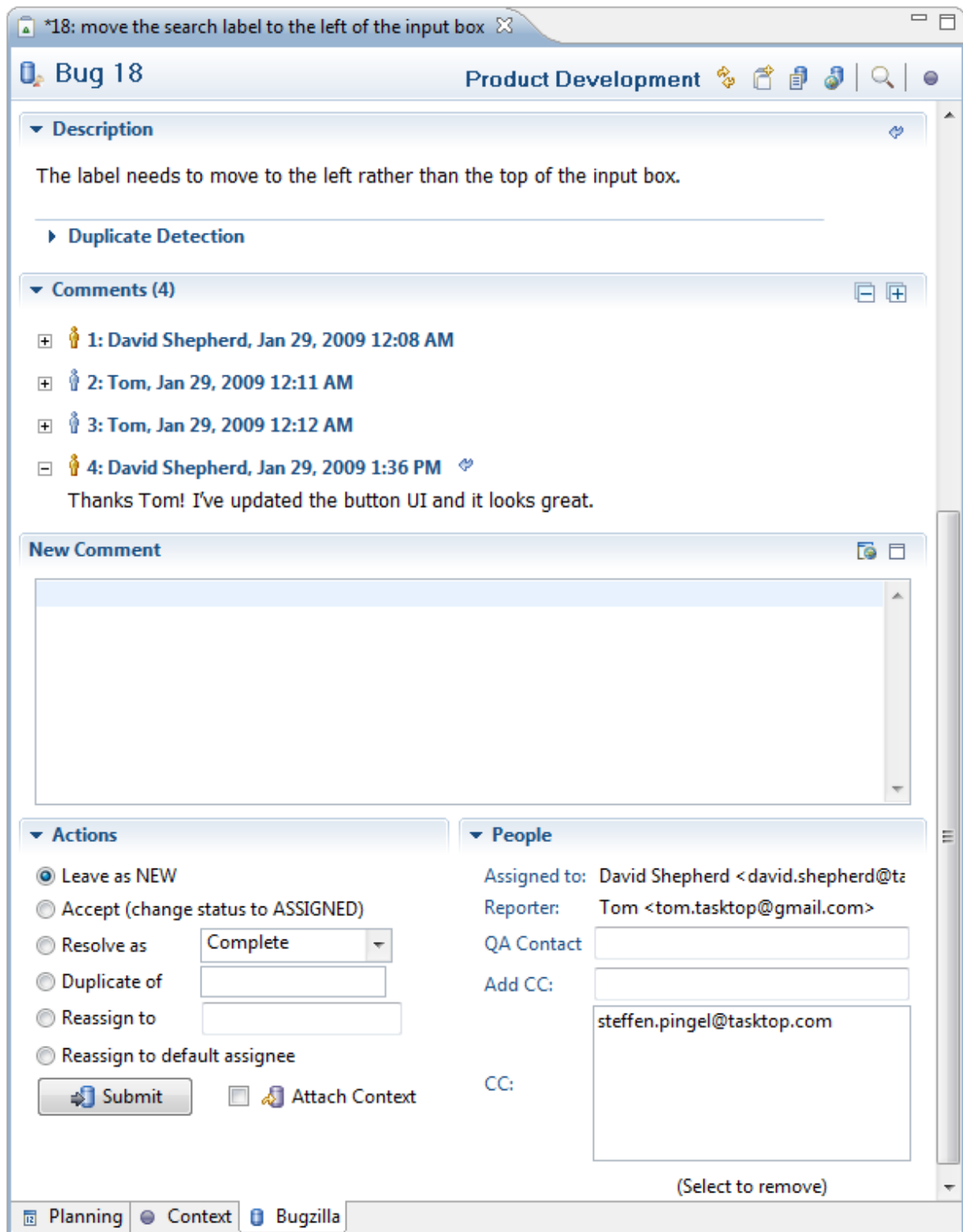


Figure 2.3: Mylyn Eclipse Plugin - Bugzilla Integration. Source: [4]

of Mylyn where a developer can directly put comments on tasks and browse work items within the IDE while making code changes. Also, they can provide the source context against tasks, so that it is possible to locate the source code for a given task as shown in Figure 2.4.

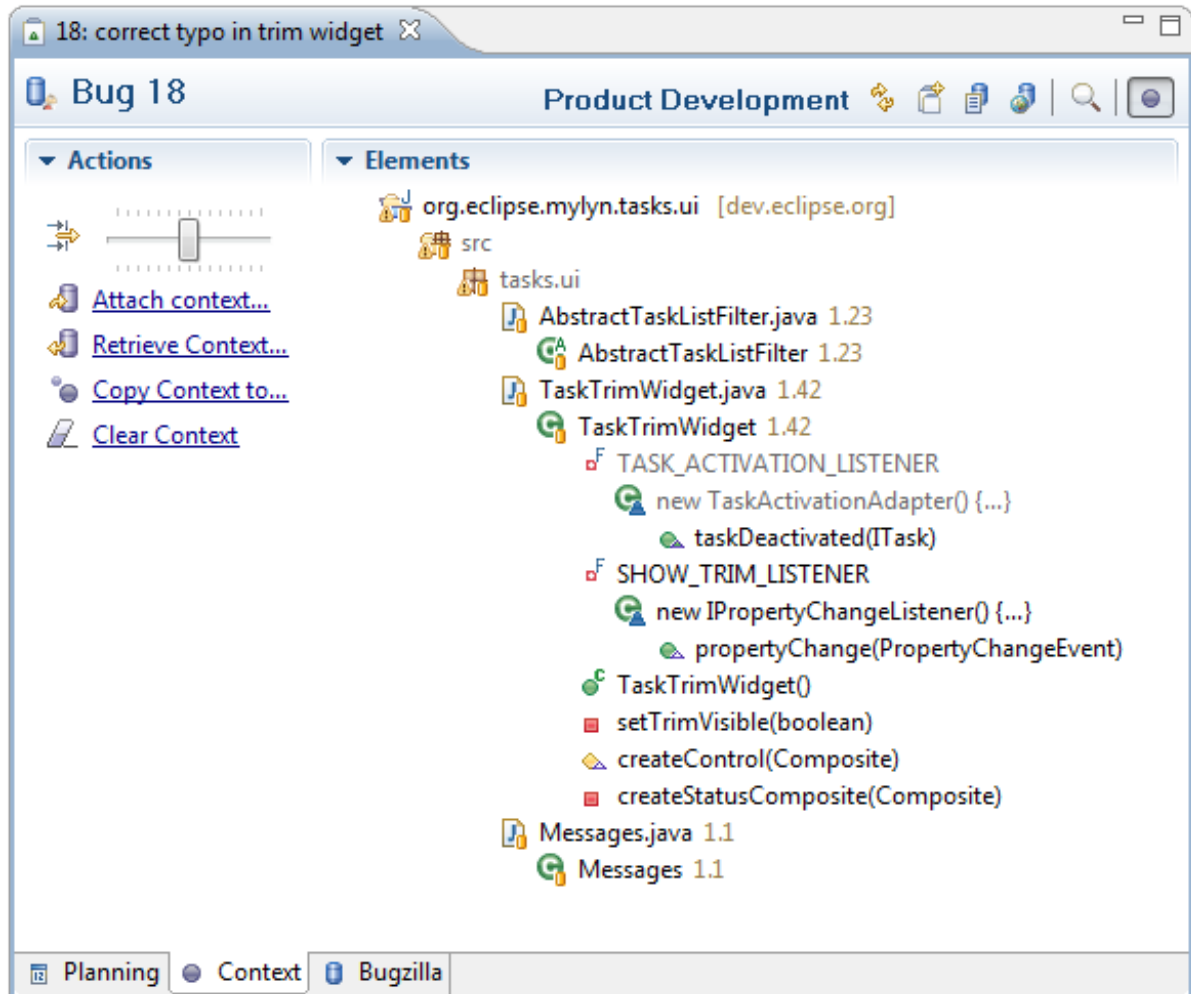


Figure 2.4: Mylyn Eclipse Plugin - Setting source code context of a bug. Source: [4]

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 source code with user stories, Taggy links email discussions. So, the knowledge base produced by Taggy is likely to complement Zelda with relevant

higher-level information from emails that can provide greater insight into user stories. In a sense, Taggy interlinks two high level artifacts, which complements the knowledge found from 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 [2]. 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.

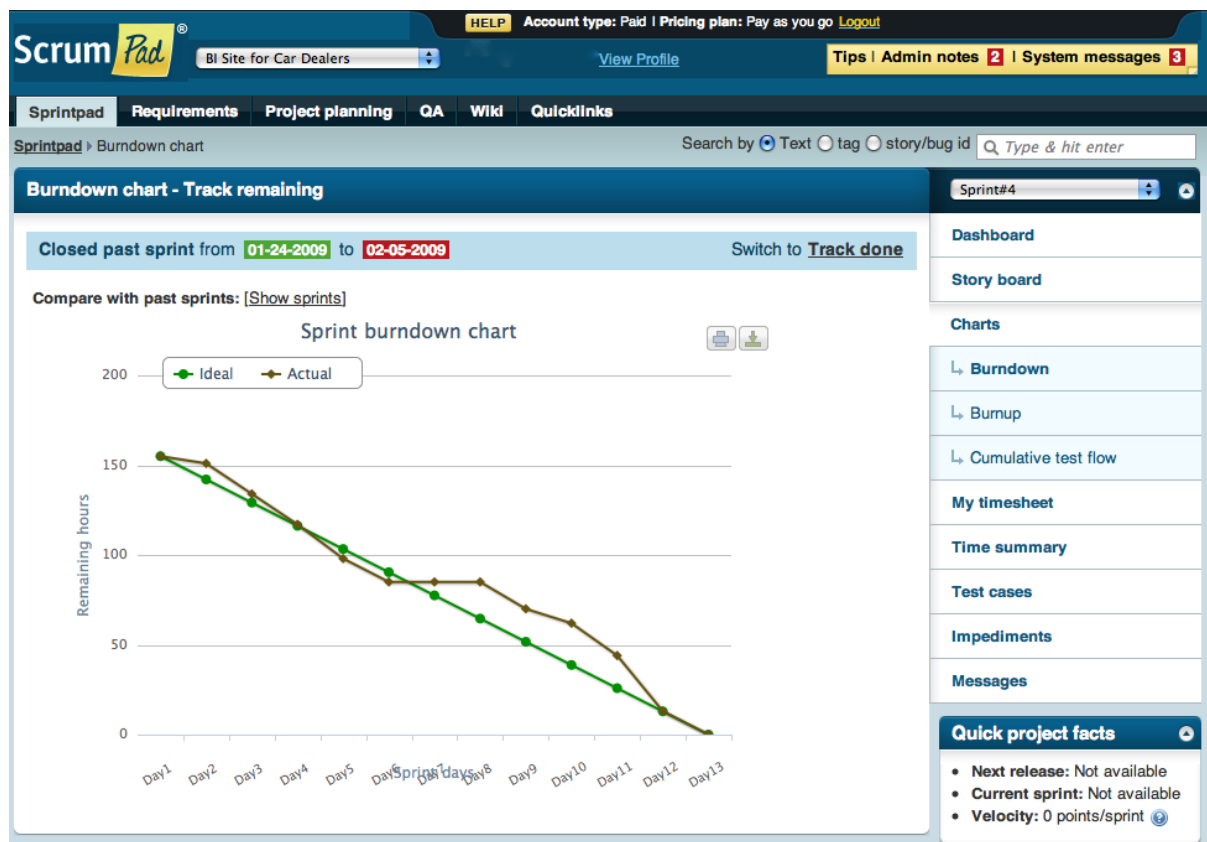


Figure 2.5: Dashboard in ScrumPad, a Commercial Agile Project Management Tool. Source: [5]

There are a number of commercial tools available for agile project management, such as VersionOne[18], Mingle[44], ScrumPad[5], ScrumWorks[45], IBM Rational Team Concert[46]. These tools offer templates for capturing various agile software engineering artifacts such as user stories, backlogs, and acceptance tests. Also, they provide process-specific workflows; for example, a workflow for Scrum includes a virtual story wall and various charts for distributed project tracking. Figure 2.5 shows a screen from ScrumPad. As this screen shows, it is possible to search and browse user stories, messages, wikis, etc. as well as produce visual reports on the project's progress using these tools. 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[19] and Trac[20].

Several issue or bug tracking tools such as Jira[47], Bugzilla[48], and FogBugz[49] 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[50].

Distributed agile teams often use general purpose project management tools such as Basecamp[51] and Teambox[52]. While these tools are not tailored to provide agile-specific artifacts, they are often picked for their simplicity of use. So, instead of providing a template for an user story or an issue, these 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 pur-

pose shareware tools such as Microsoft SharePoint[53]. SharePoint provides an infrastructure for managing websites, communities, content, search and reporting that is mainly configuration-based and does not require coding effort 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[54], CodePlex[55] 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.

Agile teams also use continuous integration tools such as Hudson [56], CruiseControl[57] and Microsoft Team Server[58] 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 includes build stats, test results and commit history. Distributed agile teams often use a separate tool to capture knowledge about the high level artifacts that are not found in the continuous integration tools.

The aforementioned tools help 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 and FogBugz, also accept incoming emails from people. But none of 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.

2.6 Summary

To summarize, the following key points are found from the related work:

1. Distributed teams commonly use emails and instant messages for knowledge sharing. Other popular mediums include wikis, project management tools and continuous integration dashboards etc.
2. A knowledge center needs to capture and serve useful knowledge from different sources without introducing much human effort.
3. It is important to build a web of knowledge that connects different software engineering artifacts such as user stories, emails, source code changes and wiki pages so that one can use all available information when needed.
4. Both the customers and technical team contributes to the knowledge about an agile project. So, a knowledge base needs to allow the flexibility to its end users so that they can utilize their familiar communication mediums.
5. The context of an artifact can be used to relate it with other artifacts. Also, one can browse the content in a knowledge base starting from an artifact and then following the other relevant artifacts for its given context.

These key points contribute to the requirements for Taggy. Taggy shows a novel technique for knowledge acquisition and tagging from emails and instant messages so that the end users can conveniently contribute without spending any significant effort.

Chapter 3

Taggy

In this chapter, I have presented the details of my auto-tagging technique in terms of Taggy. Taggy is a prototype implementation of the technique. 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 to auto-tag the emails with user stories based on the following assumptions:

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 long ago.

- The developers and/or customers of a user story often communicate via 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 clue for 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 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 low-fi 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 Chapter 2, 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 as 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 termed as “project email” 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 line is just text, similar to the body or attachment contents of an email, due to professional etiquette and for the sake of grabbing attention, people write informative subjects when 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 shippable 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 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 the 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 information for 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, Figure 3.1 shows the two high level steps involved in the process of auto-tagging an email with user stories.

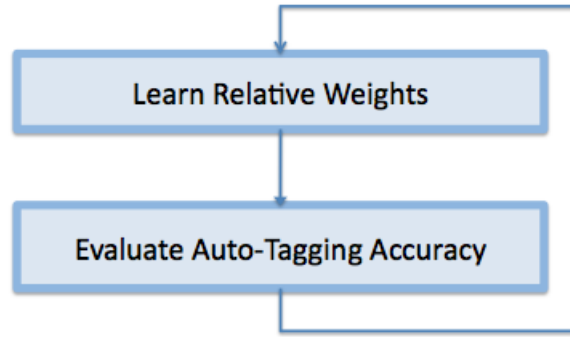


Figure 3.1: Taggy High Level Workflow

As with most other machine learning techniques, Taggy needs to learn its parameters concerning relative weights of similarity measure before making decisions. The details of finding the initial relative weights and learning is discussed later in Section 3.5.3. Once the weights are learned, the system needs to be evaluated to see its accuracy. Based on the findings of evaluation, the learning is reiterated to find a set of relative weights that produces optimum auto-tagging accuracy. This process is discussed in greater detail at Chapter 4.

After learning and evaluation, Taggy can be used to auto-tag emails with user stories. This auto-tagging process is designed to be used in a live system as illustrated in Figure 3.2. The following list explains each step of the workflow:

1. **Copy emails to project email address.** This is the input step for Taggy. As discussed in the 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

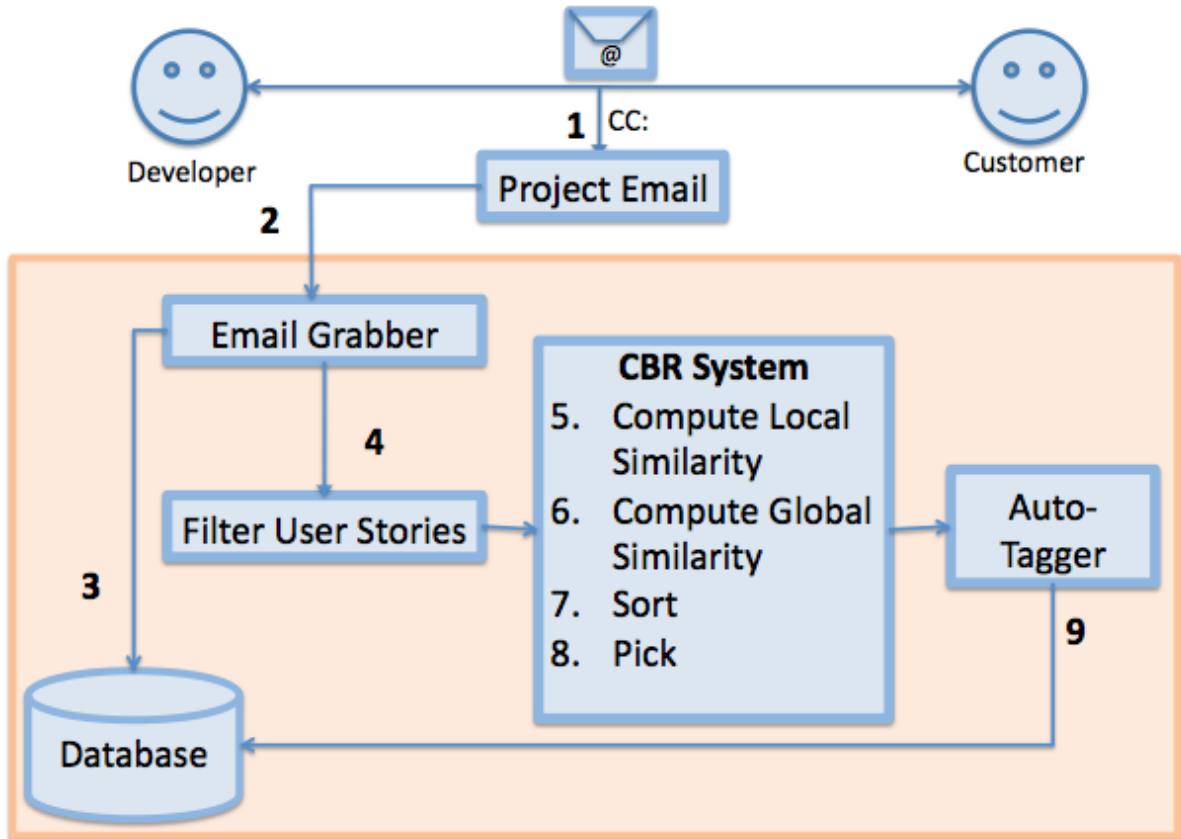


Figure 3.2: Email Auto-tagging Workflow

implemented by the distributed team. This intake process was successfully adapted in previous work [32]. In case someone forgets to do this while sending the email, it is possible to simply forward the email to the “project email” later.

2. **Grab email.** Once 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 still stored under the project. This step makes an email available for search and browsing without the need for looking into other users' inboxes.

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, it 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 discussed in Section 3.5.2.
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 3.5 . 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 that

for some emails no user story may be picked. Instead of picking up all the user stories above threshold, Taggy can be configured to pick up only the top n stories, where n is a number greater than or equal to one. This allows Taggy to limit the auto-tagging to a limited number of user stories per email.

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, steps 1-3 of the aforementioned steps differ significantly. Figure 3.3 shows these three steps. These new steps are explained next:

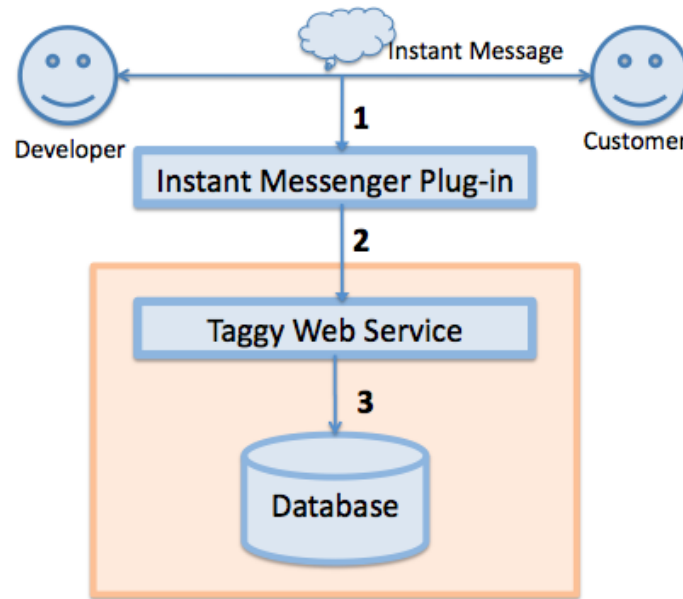


Figure 3.3: Taggy Instant Message Intake Process

1. **Activate instant message plugin.** Instant message clients often allow the use of plugins. For example, Skype [59] 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 involved 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 in 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. Instant messages that are part of a single conversation are stored together in Taggy. The conversation is identified by the supplied identifier from instant messaging client.

Since instant messages don't capture any subject, Taggy cannot produce a subject similarity. As a work around for 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/he 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 key components of Taggy is depicted at Figure 3.4. Next, the details about each component from the figure is discussed following a bottom-up approach.

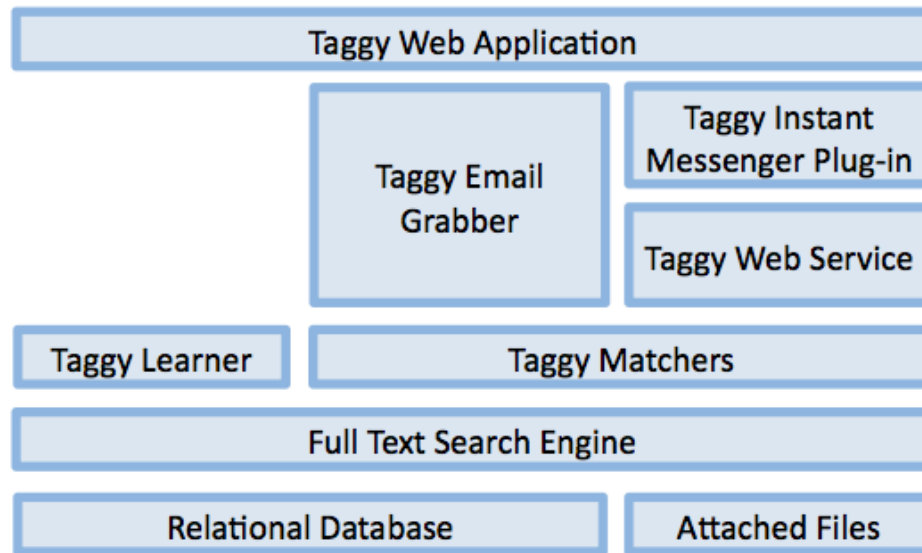


Figure 3.4: Taggy Components

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 a person's 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 would be 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: 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 required data.
5. **Taggy Learner.** As discussed before, Taggy requires training to learn 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 to 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

As I have discussed, the Learner and Matchers are the core components of Taggy. A similarity computation technique forms the heart of these components. In case of Taggy, this is a machine learning technique called Case Based Reasoning (CBR). CBR is used to find out relevant user stories for an email or instant message.

Figure 3.5 shows a high level view of the CBR system. A CBR system uses a library of

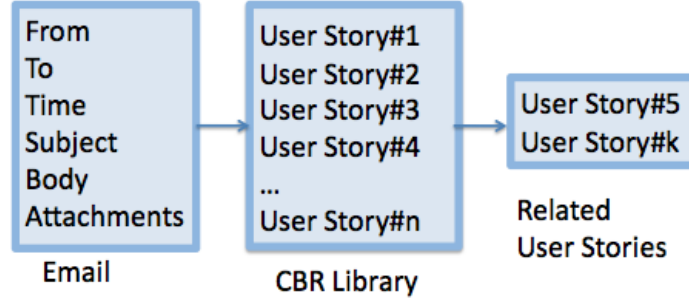


Figure 3.5: CBR System Input and Output

existing cases to match against a new case. For example, Taggy has a library of existing user stories which are treated as existing cases in the CBR system. Each case is a 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 in that they 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.

3.5.1 Local-Global Principle

The actual computation inside the CBR system is done following the local-global principle[60] 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: 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

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, 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:

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

$S_{Temporal}$ 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 negative score 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.

$$S_{People} = \begin{cases} 1 & \text{email people} = \text{user story people} \\ \frac{\#of\ common\ people}{\#of\ user\ story\ people} & \text{email people} \cap \text{user story people} \neq \emptyset \\ -1 & \text{else} \end{cases} \quad (3.2)$$

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: if 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. Please note that, a user story is supposed to have at least one customer or customer's representative in its people context. 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.

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

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

$S_{Subject}$ 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 a Vector Space Model [61] 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, a popular micro-blogging site that serves 12000 searches/second, at the time of writing of this thesis is using this search engine [62].

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:

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

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 a 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 a measure of global similarity [63]. 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 incorrect.

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 in 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 others 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. These numbers 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_weight
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 > 0

  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

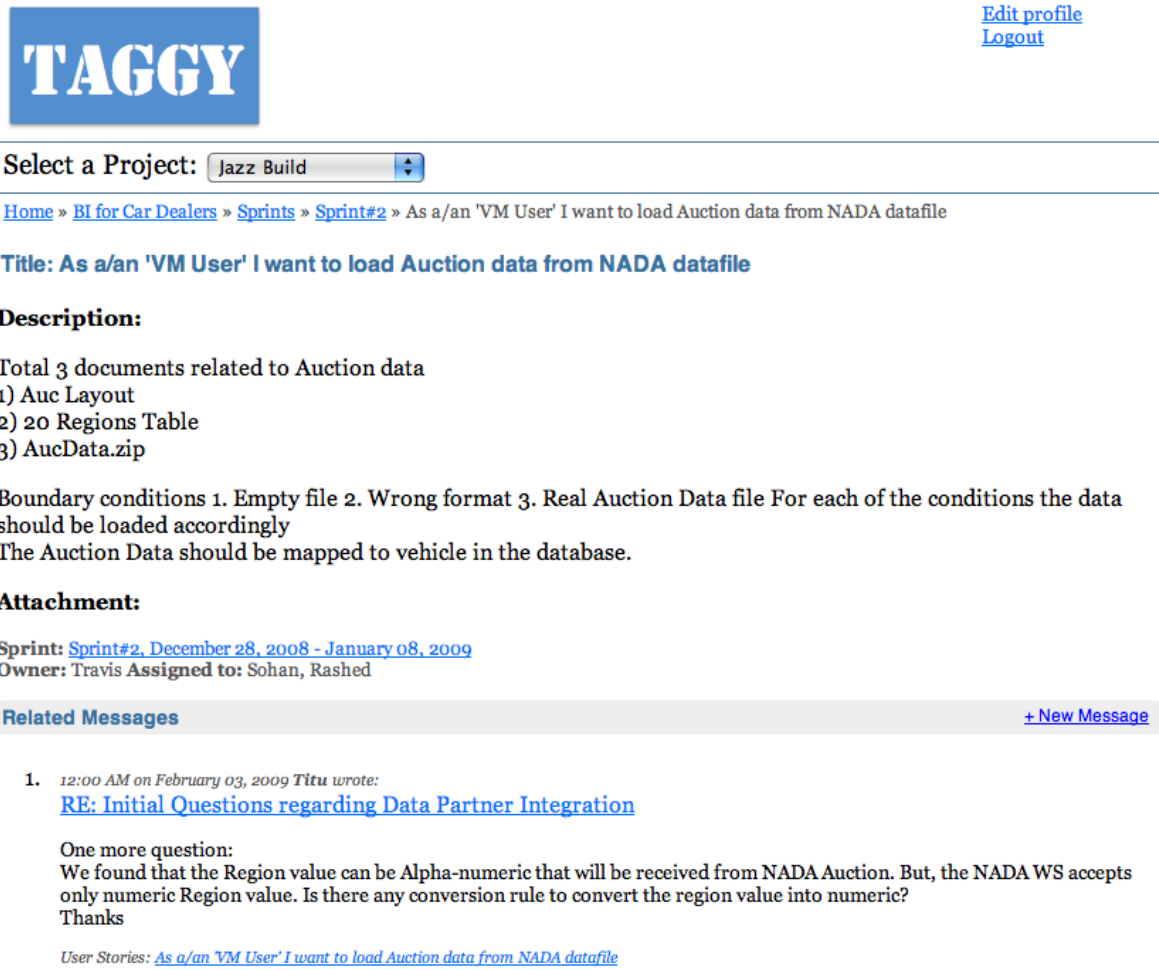
making a correct decision while punishes the one that influences a wrong decision. This reward-punishment continues unless all training emails are seen. Also, it is possible to stop if this converges to a point when adding new emails do not alter the relative weights. However, as with most machine learning techniques, the usefulness of this learning is related to the quality and quantity of the training data. As shown before, Taggy's Learner 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 the 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 since it is a transparent learning approach where one can monitor the changes in relative weights at each step of the learning.

3.5.4 Email Matching

Once the relative weights are learned, Taggy can match emails against user stories based on the trained system. This process essentially starts with grabbing email and saving it. A developer or customer sends out emails about the project and copies to Taggy using CC: feature. While saving, a lookup is performed to see if the email addresses present in the sender, recipients and CC: match against any of the existing users' record in Taggy's relational database for the project. If found, the users are linked with the emails. Next, the email matcher of the Taggy Matchers component is asked to look for relevant user stories in the database.

To an end user of Taggy, the outcome of auto-tagging is as shown in Figure 3.6 and Figure 3.7. So, while browsing user stories, one can see the related emails or vice versa.

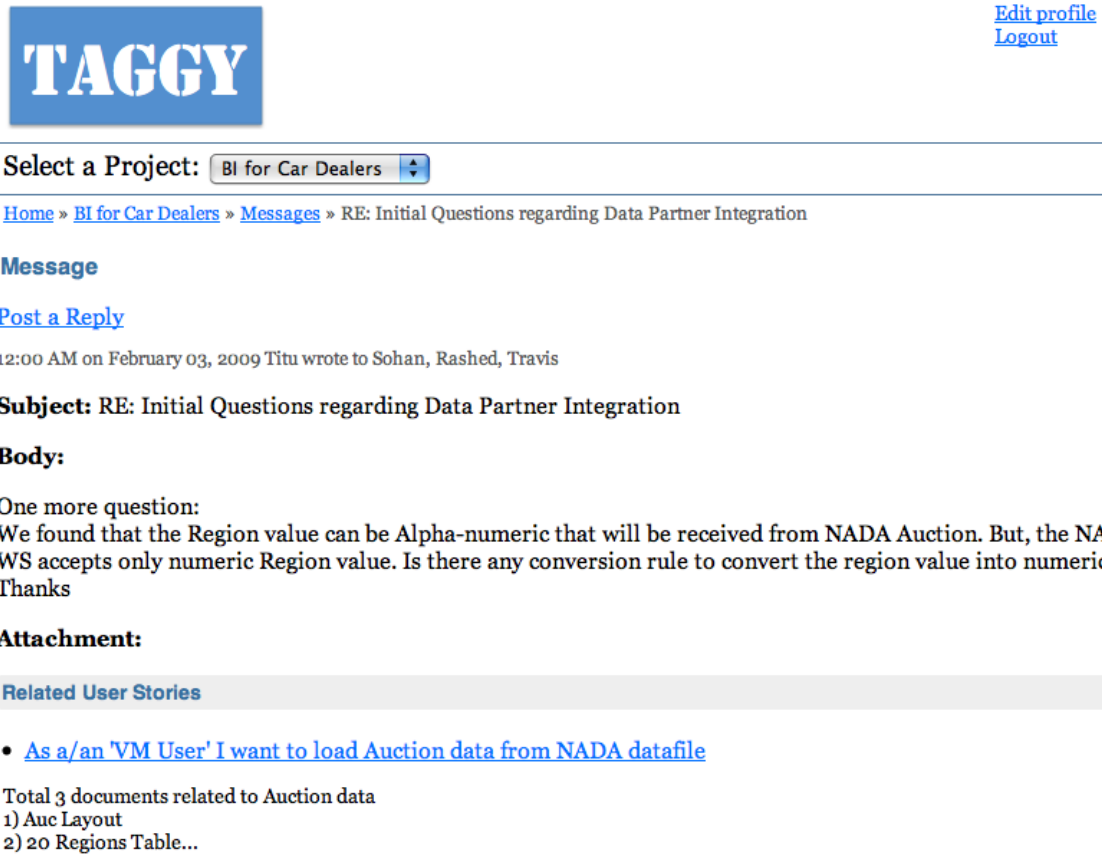


The screenshot shows the Taggy web application interface. At the top left is the 'TAGGY' logo. At the top right are links for 'Edit profile' and 'Logout'. Below the logo is a 'Select a Project:' dropdown menu currently set to 'Jazz Build'. A breadcrumb trail reads: 'Home » BI for Car Dealers » Sprints » Sprint#2 » As a/an 'VM User' I want to load Auction data from NADA datafile'. The main title is 'Title: As a/an 'VM User' I want to load Auction data from NADA datafile'. Under the 'Description:' section, it states 'Total 3 documents related to Auction data' and lists: '1) Auc Layout', '2) 20 Regions Table', and '3) AucData.zip'. It then lists 'Boundary conditions 1. Empty file 2. Wrong format 3. Real Auction Data file' and notes that data should be loaded accordingly and mapped to the vehicle in the database. The 'Attachment:' section shows 'Sprint: Sprint#2, December 28, 2008 - January 08, 2009' and 'Owner: Travis Assigned to: Sohan, Rashed'. A 'Related Messages' section includes a message from Titu dated February 03, 2009, with the subject 'RE: Initial Questions regarding Data Partner Integration'. The message content asks a question about the Region value being Alpha-numeric and whether a conversion rule exists. At the bottom, a 'User Stories:' link points to the current user story.

Figure 3.6: Taggy Screenshot - Showing User Story and Related Emails

This same outcome is found for instant messages as well. So, this email matching is responsible for “matchmaking” among the emails and user stories based on their context and text similarity.

The email matcher filters out user stories so that user stories from far past or future are discarded. Next, it computes the local similarity measures and combines them to produce global similarity values. The ones that show a global similarity above a threshold, survives as the potential candidates for auto-tagging. Based on user preferences, the email is auto-tagged against the top n user stories, where n is any number greater than or equal to 1.



The screenshot shows the Taggy web application interface. At the top left is the 'TAGGY' logo in white text on a blue square. At the top right are links for 'Edit profile' and 'Logout'. Below the logo is a 'Select a Project:' dropdown menu with 'BI for Car Dealers' selected. A breadcrumb trail reads: 'Home » BI for Car Dealers » Messages » RE: Initial Questions regarding Data Partner Integration'. The main section is titled 'Message' and includes a 'Post a Reply' link. The message content shows a timestamp '12:00 AM on February 03, 2009' and the text 'Titu wrote to Sohan, Rashed, Travis'. The subject is 'RE: Initial Questions regarding Data Partner Integration'. The body of the message contains a question about NADA Auction data and a 'Thanks' note. Below the message is an 'Attachment:' section. Underneath is a 'Related User Stories' section with a list item: 'As a/an 'VM User' I want to load Auction data from NADA datafile'. Below this list, it states 'Total 3 documents related to Auction data' and lists '1) Auc Layout' and '2) 20 Regions Table...'.

Taggy

[Edit profile](#)
[Logout](#)

Select a Project: BI for Car Dealers

[Home](#) » [BI for Car Dealers](#) » [Messages](#) » RE: Initial Questions regarding Data Partner Integration

Message

[Post a Reply](#)

12:00 AM on February 03, 2009 Titu wrote to Sohan, Rashed, Travis

Subject: RE: Initial Questions regarding Data Partner Integration

Body:

One more question:
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

Attachment:

Related User Stories

- [As a/an 'VM User' I want to load Auction data from NADA datafile](#)

Total 3 documents related to Auction data

- 1) Auc Layout
- 2) 20 Regions Table...

Figure 3.7: Taggy Screenshot - Showing Email and Related User Stories

The following code fragment shows these steps:

```

def find_most_similar_user_stories(email, n)
  nearby_user_stories = UserStory.planned_nearby(email.date)

  if nearby_user_stories
    temporal_scores = find_temporal_scores(nearby_user_stories, email.date)
    people_scores   = find_temporal_scores(nearby_user_stories, email.people)
    subject_scores  = find_subject_scores(nearby_user_stories, email.subject)
    email_text      = email.body_with_attachment
    body_scores     = find_body_scores(nearby_user_stories, email_text)

    global_scores = compute_global_scores(relative_weights, temporal_scores,
    people_scores, subject_scores, body_scores)

    global_scores_above_threshold = discard_under_threshold_scores(global_scores)

    if global_scores_above_threshold.present?
      return global_scores_above_threshold[0..(n-1)]
    end

  end

  return blank

end

```

The different similarity computations in the email matcher are carried out using the Equation 3.1, Equation 3.2, Equation 3.3, Equation 3.4 and Equation 3.5. As shown

in the code fragment, one can configure Taggy email matcher to auto-tag with the top n number of top matches. Having n greater than 1 helps to identify cases when an email is potentially relevant about two or more user stories. However, in some teams this situation may be different where most of the time the communication is just about a single user story. So, this configuration will help them to tune Taggy according to their communication patterns. Although not implemented, Taggy could also learn this parameter, n , based on the training data. If the training data shows that most emails only discuss a single user story, it may only choose to show the most relevant story or adjust this number based on the trend.

This matching algorithm is run in the same background process as that of the Email Grabber. Because of the filtering step, Taggy has a reduced search space to deal with. Although this improves speed, it may produce a wrong decision if a distributed agile team chooses to discuss about user stories from the distant past in the emails. In such cases, the definition of “distant” needs to be adjusted accordingly. In the implementation of Taggy, anything beyond immediate past and future iteration is treated as “distant” based on the observation from training data.

3.5.5 Instant Message Matching

The instant message matcher is implemented using the same approach, since we can find similar people and temporal context from instant messages as well. As discussed earlier, the intake process is different for instant messages, which uses a Taggy web service to push chat messages from a Taggy Instant Messenger Plug-in. The actual similarity computation in case of instant messages omits the subject similarity and applies a different relative weight scheme. The threshold is also adjusted accordingly.

3.6 Implementation Details

Figure 3.8 shows the implementation frameworks for the different architectural components.

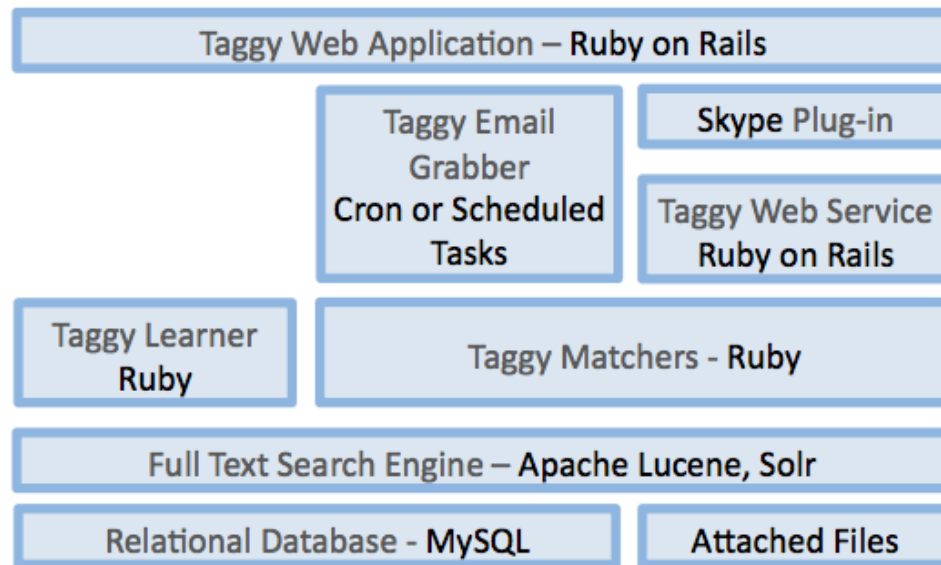


Figure 3.8: Taggy Implementation Details by Component

As with any software solution, the implementation of Taggy was based on available programming languages and platforms. The core Taggy components, the Learner and Matcher, are developed using the Ruby programming language [64]. The web application is also developed using the open-source Ruby on Rails web framework [65]. The choice of this framework was based on my prior experience as well as its support for producing quick data-backed web application development.

The full text search is served by Apache Lucene [66]. This open-source search engine is widely used across the web to facilitate full-text search. In a previous implementation of Taggy, I used Microsoft SQL Server Full-Text Search Engine [24, 67]. But Lucene was chosen because of its ability to index rich text files such as Word Documents, and PDF files. Also, Lucene can be configured to support synonyms, language stems and comes with out-of-the-box support for full-text search using several languages. However,

the architecture of Taggy allows the replacement of this engine by another one without impacting the rest of the system.

To communicate with Lucene, a web service wrapper called Solr is used [68]. Solr's RESTful web API provides easy access to Solr indexing and querying from other languages, such as Ruby. Solr and Lucene can index the contents from rich documents such as PDF files, Word documents and Spreadsheets. This feature was used to index both user story and email attachments alongside their primary contents.

Since most project management tools support full-text search anyway, this indexing is not likely an additional burden applied by Taggy. However, understandably the index size of Taggy may grow if a team frequently uses emails and instant messages, which is otherwise not captured by most of the project management tools.

For capturing instant messages, a Skype plugin is developed using the Skype4Py library [69]. The plugin can be turned on/off by the click of a button. When activated, it copies the chat messages with meta information and sends this data to the web service. Skype was picked because its use as a popular instant messenger service in software knowledge sharing [40]. Also, it offers a suite of features that are used by distributed agile teams such as group chat, group video conferencing, and screen sharing. A similar plugin can be developed for other instant message services if supported by the platform.

The background process for the email grabber can be run using Cron jobs on Unix based platforms or scheduled tasks on Windows based platforms [70, 71]. These scheduling tools can be configured to look for new incoming emails in preferred intervals starting from once every second to any realistic period.

Taggy used a MySQL relational database server to store the data. This can be replaced by any relational database supported by ActiveRecord, which includes MySQL, Oracle, Microsoft SQL Server, SQLite, Derby etc [72]. Also, it is possible to use any data storage without using ActiveRecord at all as long as Taggy gets data with a similar

schema.

As of hardware platform, Taggy does not impose any special requirements compared to other agile project management tools.

3.7 An Illustrative Example

Now that the details about Taggy's internals are provided, the following illustrative example will demonstrate Taggy in action.

First, a project's backlog is populated with user stories. This backlog has stories from different iterations. For brevity, a shortened backlog is given at Table 3.2:

Table 3.2: An Example Product Backlog

#	Iteration	Cust- omer	Deve- loper	Description
1	#1, Dec 14-25, 2008	C1	D1	As a/an 'VM User' I want to add/edit/view dealer account profile so that VM can earn revenue from charging a monthly subscription fee. Dealers will have three contacts: Billing, General Manager and Owner. Also, there will be a primary and secondary contact. More information is attached.
2	#1, Dec 14-25, 2008	C1	D2	As a/an 'VM Admin' I want to add/edit/view VM users. Administrator must be able to see username but not password VM Users will be one of inside rep and outside rep. Required fields and more information are attached...
3	#2, Dec 28, 2008-Jan 08, 2009	C1	D1	As a/an 'VM user' I want to load DMV data from Experian from a CSV/Excel file into VM database. Sample data attached.
4	Not Assigned	C1	D1	As a/an 'VM User' I want to activate and deactivate a dealer

This backlog has 4 user stories. The stories are assigned to two developers, D1 and D2, and only one customer C1. Also, this backlog contains stories from Iteration#1, December 14-25, 2008 and Iteration#2, December 28, 2008 to January 8, 2009. Also,

the user story#4 is not yet planned for any iteration. In an agile team, this list is likely to be longer with more user stories, iterations, developers and customers. However, for the sake of an example, this small backlog serves the purpose.

Now, developer D1 needs to get a clarification from the customer C1 while working on a user story at the start of iteration#1. This is asked using the email as shown below:

To: C1

From: D1

CC: project@Taggy

Date: Dec 14, 2008

Subject: Initial Questions on **Dealer setup**

Body:

Please clarify the following questions regarding dealer setup data-

1. Is it ok to assume that the billing, GM and Owner contacts of a dealer will also contain username/passwords. Note that, the main and secondary contacts as well as the Used car manager contact contain user name/passwords.
2. What data should we collect for Physical Address and Mailing Address of a dealer? Is it free text? Or collected as fields street1, street2, city, state, zip, country?
3. Is it possible that for a dealer both an inside and outside salesman is assigned?
4. Apart from the name and pricing, is there any other field associated with the “program” main/platinum information?
5. What is meant by the “Location” field of the outside VM Rep?
6. Is it possible to provide us with a sample input data that will be used to set up an inside/outside VM user’s commission? Please provide examples of default, bonus, retention and special commissions.

7. What are the required fields of all the dealer setup fields?
8. The Business Address is mentioned twice under the 5.6.1.5 - do we need to capture two business addresses?

Please note that this email has a CC: to project@Taggy. So, Taggy will be able to read the contents of this email. As shown in the email matcher, Taggy will try to auto-tag the email with the existing user stories based on context and text relevance. While doing so, the local and global similarities are computed as shown in Table 3.3:

Table 3.3: Similarity Scores

Story #	Temporal Sim.	Temporal Wgt.	People Sim.	People Wgt.	Subject Sim.	Subject Wgt.	Body Sim.	Body Wgt.	Global Sim.
1	1		1		0.6		0.4		0.76
2	1	28.6	0.5	20.2	0.4	34.4	0.25	16.8	0.57
3	0		1		0		0.2		0.24
4	-1		1		0.3		0.3		0.07

Here, similarity scores are computed based on the similarity equations. For example, user story#1 has a people similarity score of 1, since all people in the story, C1 and D1, are present in the email. On the other hand, user story#2 has a people similarity score of 0.5 since only one (C1) of the two people from the user story is found in the email. The values for other similarity measures can also be traced following the respective equations.

From the above table we see that Story #1 is the most similar story to the given email with a similarity score of 0.76. This is greater than the threshold of 0.58 (details discussed in Chapter 4) and thus the system auto-tags the email with Story #1. However, the other stories are not considered as related as those failed to reach the threshold similarity score.

In this example the email is auto-tagged with user story#1. However, if the threshold is set to a lower value or there was another user story with similar profile to user story#1, Taggy could recommend more than one auto-tagging. On the other hand, it could simply

ignore auto-tagging with any user story if another email was too irrelevant for the given backlog.

Chapter 4

Quantitative Evaluation

As previously mentioned in the research goals, an evaluation is required to measure the accuracy of Taggy in auto-tagging emails with user stories. In this thesis, the evaluation is based on two different real world data sets, collected from two different sources. In total the data represents 4,745 messages created by 9 agile teams. The evaluation shows that for different projects Taggy can correctly auto-tag between 76% to 81% emails with user stories after sufficient training.

This chapter is organized as follows: Firstly, the evaluation approach is discussed. As a part of it, the adapted definition of accuracy is given. Next, a null hypothesis is stated which is used in deducing the statistical relevance of the evaluation results. Then, the two data sets are described and the evaluation results for each are provided in detail. Finally, the limitations of this evaluation are discussed.

4.1 Evaluation Approach

The evaluation steps, as depicted in Figure 4.1, are described below.

1. **Random Select.** Randomly select 10% emails from all available emails for training. Training emails are randomly selected to improve the possibility of encountering different patterns in the data.
2. **Initialize Relative Weights.** Initialize the relative weights for temporal, people, subject and body similarities using the training data from step# 1.
3. **Learn.** Use the reward-punishment approach for each training email from step# 1, adjust the relative weights.



Figure 4.1: Taggy Empirical Evaluation Steps

4. **Evaluate.** Using the adjusted relative weights from step#3, auto-tag the remaining emails that were not included in the training data in step# 1. This ensures the training data and evaluation data contain completely disjoint sets of emails. Compute the accuracy of the auto-tagging.
5. **Iterate.** Repeat steps 1 to 4, with 20%, 30%, 40% and 50% data for training. This step attempts to find an optimum partition of training data set where the relative weights converge to a point so that adding more training data adds little value to the auto-tagging accuracy.

4.1.1 Accuracy

The accuracy of auto-tagging is a ratio between the number of correct tags and the total number of emails presented for evaluation. Since this excludes the training data set, the accuracy only represents the accuracy over the evaluation data set. So, the accuracy is computed using Equation 4.1.

$$Accuracy = \frac{\text{Number of correct autotagging}}{\text{Number of total evaluation emails}} * 100 \% \quad (4.1)$$

As Equation 4.1 suggests, it is necessary to be able to justify whether an auto-tagging is correct or incorrect. So, the evaluation data needs to provide information about correct relationship of an email with user stories so that the auto-tagged results can be compared against known values.

4.1.2 Statistical Relevance

The statistical relevance is found by starting with a null hypothesis and then observing the goodness-of-fit of the evaluation data based on chi-square test [73]. In this case the null hypothesis is:

The auto-tagging accuracy of Taggy is no better than a random decision making.

The outcome of an auto-tagging process is either a correct or a wrong tagging of an email against the user stories in a project. Similarly, a random system will produce one of the two outcomes. The target of this statistical relevance determination is to see if the accuracy found from Taggy's evaluation is likely to be achievable by a random system.

In this case, we have a single degree of freedom, since there are two possible outcomes, correct and incorrect. And the expected value of correct and incorrect decisions is same

for a random system. Based on this information, the chi-square computation is done using Equation 4.2:

$$\chi^2 = \frac{(\text{Count of correct tagging} - \text{half of total evaluation emails})^2}{\text{half of total evaluation emails}} + \frac{(\text{Count of wrong tagging} - \text{half of total evaluation emails})^2}{\text{half of total evaluation emails}} \quad (4.2)$$

This computed χ^2 value is compared against the upper critical chi-square value for probability 0.05 with 1 degree of freedom, that is $\chi_{0.05}^2=3.841$. The probability value of 0.05 is conventionally used in significance finding. A value beyond this threshold indicates disagreement between the observation and the null hypothesis. We also present the corresponding p-values against the computed chi-square values. The lower the p-value, the less likely it is that the observation is true given the null hypothesis holds.

4.2 Evaluation Results

The evaluation results are discussed in two sections for two different data sets.

4.2.1 Data set# 1: ScrumPad

This data set was obtained from an online agile project management tool called ScrumPad. ScrumPad allows users to manage product backlog, perform iteration planning and track progress. Also, it lets users discuss the project's user stories using message threads. Each message thread can contain an explicit link to a user story. While sending a message using ScrumPad, one can link it with a user story. The messages are stored in ScrumPad database and also sent to people via emails. People can directly reply to the notification emails, which goes to ScrumPad and gets saved as a message either in a new or under an existing thread.

The user stories in ScrumPad contain information about its title, description, attachments, customer, developers and iteration. Taggy used these user stories to train and evaluate its auto-tagging accuracy.

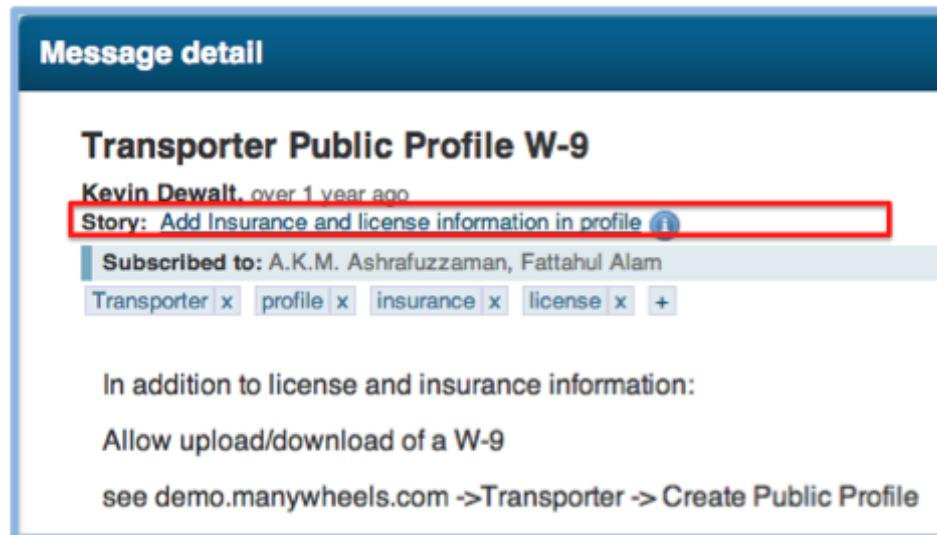


Figure 4.2: A ScrumPad Message Linked to a User Story

Figure 4.2 shows the screenshot of a message at ScrumPad. As this figure shows, the message has a link to the user story. The messages from ScrumPad message threads are derived as emails for the evaluation of Taggy. Messages contain subject, body, sender, recipients and attachments. These fields are mapped against the standard fields of emails. Also, if a message thread in ScrumPad is linked against a user story, all derived emails from the messages of that thread are also treated to be linked against the user story. So, these are essentially the correct tags for the emails since they were all manually provided by the contributors to the message threads. This information helps the training of Taggy using reward-punishment approach with the necessary feedback. Also, the accuracy computation relies on this information to find if an auto-tagging is correct or not.

ScrumPad data set contains data from five distributed agile projects. Four of the five projects, MethodMarketing, BI for Car Dealers, ManyWheels and VarsityDays were

developed by Code71, Inc. (www.Code71.com). Code71 is also the creator of ScrumPad. These four projects employed developers from Dhaka, Bangladesh and Virginia, USA. These projects were developed for four different clients from USA.

The fifth project, MindAndMarket, was distributed among one developer from Bangladesh, and two developers and a client from Belgium. The Belgian development team and the client worked at the same city.

Table 4.1: Scrumpad Data Set

Project Name	Description	Users	Iterations	It. Len. (days)	User Stories	Emails
Method Marketing	Online loan application	7	14	14	86	183
BI for Car Dealers	A decision support tool for vehicle dealers	7	11	14	70	213
Many Wheels	A web application for transporters and shippers	5	12	14	97	158
Varsity Days	A social networking application for school sports teams	5	15	14	88	46
Mind And Market	A project collaboration tool	3	6	7	28	40
Total					369	640

Table 4.1 shows the composition of this data set. In total, this data set contained 640 emails and 369 user stories from 5 distributed agile projects. It is worth mentioning that 640 messages were exchanged using ScrumPad. However, the actual number of project related emails may not be limited to this volume.

Given this data, the aforementioned evaluation approach was followed to train and auto-tag the emails. The iterative approach resulted in an optimum training data partition size of 20%. This size was selected because, a random partition containing 20% of all emails for training produced better auto-tagging accuracy than a smaller sized partition (e.g. 10%) and was as good as a larger sized partition (e.g. 30%).

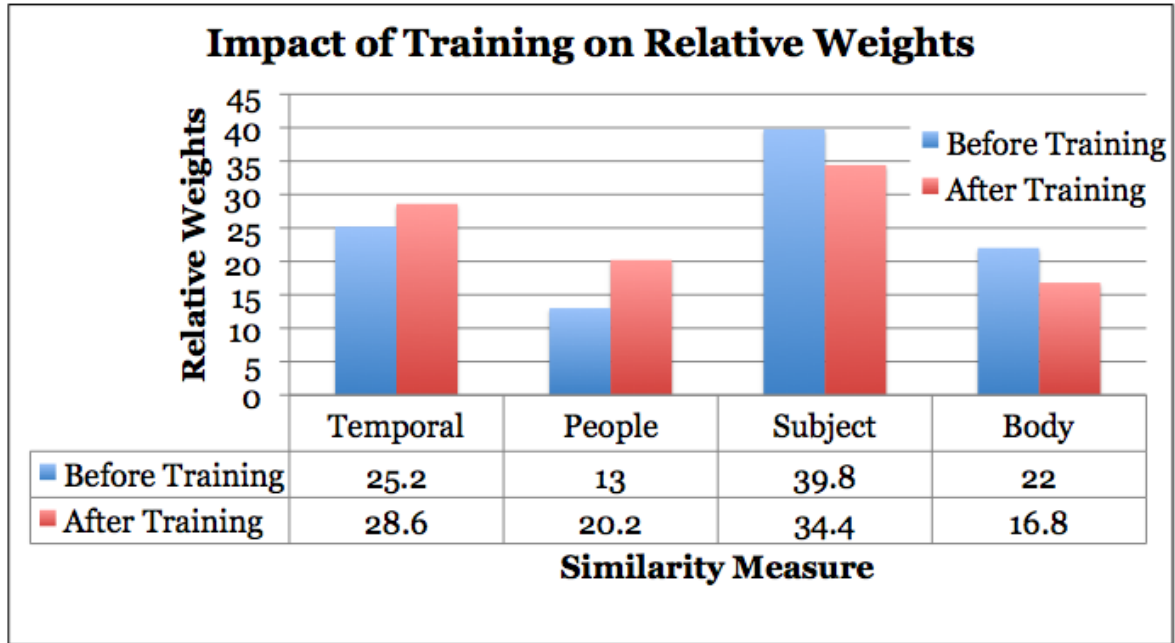


Figure 4.3: Impact of Training on Relative Weights

Chart 4.3 shows the impact of training on the relative weights of different similarity measures. The chart shows that the initial relative weights for the similarity measures of the text components (subject and body) are reduced while the context components (temporal and people) are increased as a result of the training.

Instead of training Taggy on a per project basis, I used 20% data from all the projects. As a result, Taggy could benefit from a large number of training emails. However, it is possible to use a per project learning as well. I have discussed the trade-off regarding learning for specific project vs. a collection of projects at Chapter 6.

The maximum similarity score for a local similarity measure can be 1.0 as shown in the similarity equations at Section 3.5.2. Using the trained relative weight values and a threshold global similarity score of 0.58, an email will be auto-tagged with a user story if one of the following is true:

1. The email has strong text similarity (maximum $0.34 + 0.17 = 0.51$) and some

context similarity with the user story.

2. The email has context similarity (maximum $0.29 + 0.20 = 0.49$) and some text similarity with the user story.

So, setting such a threshold score forces auto-tagging decisions to be based on both context and text relevance. This threshold was found from observing the data. In the training data, the global similarity function produced a score of 0.58 or higher for 90% of the actual email and user story relations. This cut-off point helped to eliminate some false positives and false negatives. The trade-off between selecting this threshold and auto-tagging is discussed later in Chapter 6.

Using this threshold and the trained relative weights, the evaluation yielded the results as shown in Table 4.2.

Table 4.2: Evaluation Using Scrumpad Data Set

Project Name	Total Emails	Training Emails (20%) of Total	Evaluation Emails (80%) of Total	Correct Auto-Tagging	Accuracy	χ^2	p-value
Method Market- ing	183	37	146	111	76%	39.6	0.000000
BI for Car Dealers	213	43	170	133	78%	54.2	0.000000
Many Wheels	158	32	126	93	74%	28.6	0.000000
Varsity Days	46	9	37	29	79%	12	0.000532
Mind And Market	40	8	32	24	74%	8	0.004678
Total	640	129	511	390	76%	141.6	0.000000

As shown in Table 4.2, out of a total of 511 evaluation emails, Taggy could produce correct auto-tagging of 390 emails. This result is found after using 129 emails for training.

The training data was selected randomly from the five projects. The table also shows that the χ^2 values are greater than single degree of freedom $\chi^2_{0.05} = 3.841$, which refutes the null hypothesis as stated before.

The evaluation shows that the accuracy varies between 74% to 79% for different projects with a mean of 76%. However, if only the text similarity measure is used to auto-tag, then Taggy produces 47% accuracy based on the same training and evaluation data set. So, in this evaluation, the addition of context similarity using CBR improves the auto-tagging accuracy by 29%. Some of the user stories in the projects often shared a common vocabulary. For example, the following two user stories are taken from the ManyWheels project:

1. As a Shipper I want to register on ManyWheels website using email, password, name and other common registration fields.
2. As a Shipper I want to login to ManyWheels website using email and password.

Both the stories were developed at Iteration#1 in February 2008. However, two different developers were assigned to implement the stories. In such cases, the text of an email is highly likely to show similar relevance with the text of both the user stories. However, when looked into the people context, the additional information can distinguish the actual relevance of the email with the user stories. As shown in this example, adding context information can help to find the relevant user stories for an email as well as eliminate some of the incorrect auto-tagging that would result if only text relevance was used. It should be noted that the presence of the free text component leads to a fuzzy match and a 100% accuracy may not be achieved as a result of this.

4.2.2 Data set# 2: IBM Jazz

Jazz was a large project at IBM where they built a web-based project management and collaboration tool called Rational Team Concert (RTC). RTC used itself for project management. The RTC project had multiple team areas working on different areas of the application. In evaluation data set# 2, I have used data from 4 key team areas of the project.

For this evaluation, I have used the work items from RTC as user stories. RTC stores the work items with its title, description, owner, assigned developers, and planned iteration name. These fields are same as seen with user stories. Under every work item, there is a message thread, where the team members can discuss about the work item. I have derived emails from the messages in the message threads with an important change: the email subject was formed by adding an “RE:” prefix to the work item title since the messaging system didn’t allow the users to provide a subject line. However, the other fields of messages such as sender, recipients, time stamp, and the body were mapped to the standard email fields.

Taggy was not trained based on data set# 2, rather the trained relative weights from data set# 1 were used to auto-tag the emails. It is clear that training based on data set# 2 would put a very high relative weight to subject similarity since the subjects are identical to the user story titles. Using the training based on data set# 1 reduces this bias.

Table 4.3 shows the composition of this data set. This data set comprises of 4105 emails and 1708 user stories from 4 team areas or sub-projects. As Table 4.3 shows, the total number of users in the teams were different from average active users per iteration. This is because the team areas in the Jazz project were formed as sub-projects and people were allocated based on the iteration targets. So, in this case, a total of 71 users contributed to the Work Item team area in 3 iterations (completed in 5 months).

Table 4.3: IBM Jazz Data Set

Team Area Name	Users	Avg. Active Users	Iterations	It. Len. (days)	User Stories	Emails
Work Item	71	30	3	40-50	695	1643
CC Connector	30	20	5	15-30	518	1330
Agile Planning	53	26	5	30-50	384	840
Build	24	14	3	30	111	292
Total					1708	4105

However, only 30 users were active members since they were either owners or assigned developers of the user stories. The rest 41 users were from the other team areas and contributed with their feedback as end users of the product. The team areas had varying iteration lengths, for example the Agile Planning team spent 50 days for the first iteration but 30 days for the fifth. A large number of discussion went on about the work items since the team members were using the product firsthand while it was being developed.

Based on the training from data set# 1, Taggy produced the evaluation results for data set# 2 as shown in Table 4.4:

Table 4.4: Evaluation Using IBM Jazz Data Set

Team Area Name	Evaluation Emails	Correct Auto-Tagging	Accuracy
Work Item	1643	1273	77%
CC Connector	1330	1011	76%
Agile Planning	840	683	81%
Build	292	235	80%
Total	4105	3203	78%

The evaluation based on data set# 2 shows a 78% accuracy in auto-tagging. Although the training is based on data set#1, this data still carries a bias that the email subjects are identical to their relevant user story titles. So, in the global similarity computation, a score of 0.34 (subject similarity weight is 0.34) is ensured for the emails. To reach the threshold of 0.58 it needs to produce the rest 0.24 of the required score from the body

and context similarity. So, the 22% incorrectly tagged emails had poor context and body similarities with the relevant user stories.

4.3 Limitations

The quantitative evaluation has the following key limitations:

1. For the evaluation, message threads were treated as emails instead of using actual emails. This approach helped in the training and evaluation since the message threads were already linked with the user stories. However, in case of data set# 2, using user story titles were used as email subjects since this information was absent. This may not be the case for an actual email. Similarly, despite having a common structure and intent, message threads and emails may be different in some aspects. An evaluation using actual emails may surface new findings that are not seen when message threads are used.
2. The evaluation data were taken from two sources involving 9 teams. The projects were developed by two companies. This limits the generalizability of the evaluation findings.
3. The evaluation was done based on historical data as opposed to a live system. Since the auto-tagging process needs to work on a live system, evaluating the accuracy on an ongoing basis could unveil new findings that are not found in a historical data.

Chapter 5

Preliminary Qualitative Evaluation

Although the quantitative evaluation serves the purpose of identifying the accuracy of Taggy in auto-tagging, I have conducted a preliminary user study of the tool to get qualitative feedback about it from people involved in software development projects. The feedback comprises of responses from 6 participants. To familiarize them with Taggy, I have demonstrated the key features and explained the underlying technology first. Then, I have invited them to use the Tool to try out the features on their own. Next, I conducted open-ended interviews to elicit feedback around the following four research questions:

1. Is it useful to auto-tag emails?
2. Is it useful to auto-tag instant messages?
3. What are the potential benefits of using Taggy?
4. What are the concerns about Taggy?

The remainder of this chapter provides information about the study participants, data collection, analysis, findings and limitations of the user study.

5.1 Participants

Table 5.1 summarizes the basic information about the study participants. The participants (P1-P6) are from 3 different teams(T1-T3). T1 is a team of 5 working from three different locations. P1 is the product owner for the team who decides the project's user stories, participates in the project planning and regular feedback process.

Table 5.1: Taggy User Study Participants

Team	ID	Role	Years of Exp.
T1	P1	Product Owner	5
T2	P2	Developer	3
T2	P3	Developer	9
T2	P4	Support Specialist	10
T2	P5	Developer	15
T3	P6	Developer	Undergraduate intern

T2 is a Calgary based team of 5 developers. The team develops and provides maintenance support for an end to end recruitment management software. The clientele of T2 includes both local clients as well as remote clients from Asia and Europe.

T3 is a team of 3 Computer Science undergraduate interns and a university professor working from three different universities across Canada. The team T3 develops reusable components to be used by others.

All the teams mentioned here follow iterative incremental process with small iterations (2-3 weeks). Also, they follow a collaborative approach, where the team members and the customers collaborate to define and refine the user stories. Some participants (P1, P2, P3, P5) mostly rely on emails to communicate with remote members while others commonly use instant messages (P4, P6). Participant P4 also relies on an issue tracking tool.

Since the teams are distributed or working for remote clients following some of the agile practices, they are already familiar with the concepts of user stories and iterations. Also, they frequently use text based communication tools. Taggy is essentially a lightweight knowledge management tool for such teams.

5.2 Data Collection & Analysis

The data collection was based on a questionnaire and audio-recording of interview session for each participant. The questionnaire was used to learn about the participant's

background information comprising of the role, number of years of experience, current team composition, tool usage and so on. The Table 5.1 summarizes some of the key information collected from the questionnaires.

Next I gave them a half an hour demonstration of Taggy and discussed about its underlying technique. After this demonstration, the participants tried out the different features of Taggy. They created user stories and tried out auto-tagging with different emails including attachments. Some of them also tried out the instant message auto-tagging using the Skype plug-in. This evaluation was limited to half an hour. Next, I interviewed each of them for about 20 minutes and the interview sessions were audio recorded. A near verbatim text transcript of the audio recordings was extracted and then open coded being inspired by the Grounded Theory Approach [74]. Open coding helps to identify concepts by analyzing data. The codes were then categorized around the four main themes as discussed at the start of this chapter. The findings around the themes are discussed next.

5.3 Findings

Is It Useful to Auto-tag Emails?

The participants were able to send emails to Taggy using the CC feature and found it to be simple and straight forward. They provided encouraging comments about the auto-tagging feature. For example, P6 commented,

“It will be really useful, since it brings information into a single place from different sources”

Another participant P1, mentioned that Taggy adds value to the emailing process by automatically relating with the user stories as he mentioned,

“...It does help to have centralized location where you can communicate and share knowledge easily as done through Taggy. Our suppliers are based on Asia, so its important to relay the information to them seamlessly. Taggy eliminates the clutter from emails as it automatically grabs the emails from different stakeholders and tags with the user stories.”

All the participants stated that remembering the project email is a lot easier than remembering the user story ids or other tokens for both technical and non-technical members. However, participants P4 and P5 brought the fact that at times the clients may forget to put the project email in the recipients list. For example, participant P5 mentioned the following:

“For me it would be really cool if it has a great automatization. However I can see some troubles with clients remembering the email to put CC or me doing it later when I reply. It may be a thing to think about.”

As an alternate route to feed the project related emails to Taggy, P3 suggested the following:

“As a work-around for copying to project email, I can set up email automatic forwards based on rules such as when they are from customers it should auto-forward to Taggy.”

Such a solution may work for advanced level email users who are aware of rules, filters and automatic forwards. However, instead of listening to a single email address for a project, this setup needs to be done at the email account for each of project members. As another alternate, participant P4 suggested the following:

“You can setup a middle man kind of email address so that clients always send the email to the address and then Taggy picks it up and forwards to the project members.”

To summarize, the participants could successfully send emails to Taggy and try out the auto-tagging feature. Their feedback about using different input mechanisms can be part of a future extension of Taggy.

Is It Useful to Auto-tag Instant Messages?

The participants P1, P4 and P5 also tried out the instant message auto-tagging feature of Taggy. Since the Skype plug-in silently sits in the background and sends out the chat messages to Taggy, they found the input process to be simple and unobtrusive. Some participants expressed a greater need for capturing instant messages than the emails, since their principal knowledge sharing medium was instant messaging. P5 mentioned the following:

“its (instant message auto-tagging) what makes it really powerful... for example, sometimes we exchange a quick link or a short note - its a pain to capture that later. You have to save it or send an email, sometimes I forget to do that. Sometimes you close the computer and its gone. It (instant message auto-tagging) will be very useful. ”

Participant P6 mentioned that the plug-in makes it easy and unobtrusive to retain the instant messages. Since he uses two instant messaging service for project collaboration, his instant messages are further fragmented into two places. He mentioned the following as an example where Taggy would be useful for his team:

“I implemented zooming and panning before. So, when my teammate needed help about a similar task, I looked up the code from the project and sent him in Skype. It would be good to have this stored with the task, so that one can easily use this information.”

However, participants expressed the need for instant messenger plug-ins beyond just Skype as some of them mentioned other popular clients such as MSN, Yahoo, Google

etc. Since the auto-tagging is done irrespective of the instant message client type, the same process can be applied to messages from any source that include similar data.

What are the Potential Benefits of Using Taggy?

This part of the study attempted to look for participant responses about who and why someone would use Taggy. New team members were identified as the most common target consumer of this knowledge. Participant P4 mentioned that it could actually help the developer during the development of a user story, since she doesn't need to look into different places to see the discussions about a user story. Participant P6 mentioned that while implementing or testing a user story, at times he refers back to the user story description to check if everything is in the right direction. As a part of description, he also looks into the relevant instant messages. Here is one of his comments about Taggy:

“Instead of looking into several places, Taggy helps me to see all in a single page. And its a shared page as well.”

Participant P4 mentioned that it would help in providing support for customer enquiries about a feature, especially with information that was discussed during its development. The participants wanted to use Taggy mainly to keep their knowledge in a central place without putting much efforts. For example, P1 mentioned the following:

“It will help the new team members. They are able to see the history of user stories... so it will help them up to spec with current development quicker.”

Participant P3 mentioned that it will reduce his documentation efforts since the customer feedbacks from the emails will be automatically captured in a shared location.

What are the Concerns about Taggy?

I also enquired the participants to share their concerns about Taggy. The most common concern was about presenting the archival information. Since, a distributed team may

produce huge amount of content in terms of emails and instant messages, the participants were concerned about information overload. This concern can be addressed by finding an appropriate user interface that is suitable for searching and browsing archived data. This can be a future work on Taggy.

Another concern was about the accuracy of the auto-tagging process. While Taggy’s auto-tagging doesn’t guarantee a 100% accuracy, the quantitative evaluation provides a historical evidence about its accuracy. Also, it is possible to rectify the incorrect decisions made by Taggy, which essentially helps Taggy to adjust its learned parameters.

Participants also mentioned that sometimes people may forget to put the project email in the recipients. In a previous project a similar email intake method was successfully utilized[32]. However, a future work needs to explore if this adds noise to the email communication process and find other alternative email intake methods. The auto-tagging process of Taggy can be invoked through other alternative email intake methods.

5.4 Limitations

This study has several limitations. Firstly, the participants only used Taggy for half an hour. While their feedback based on past experience is valuable, its not clear how the feedback would translate if they used Taggy in a real project for a longer period. Since a knowledge base is essentially targeted to help long term project success, the short evaluation may not reveal some of the important aspects that would matter in the long run.

Secondly, the user study had only 6 participants. Only one of the participants had a role of customer while the rest were technical users. Moreover, the participants were all from small teams of 4-10 members. So, the findings cannot be generalized.

Thirdly, as any other qualitative interviews, the participants and their responses may

carry a bias. A future work may address these limitations by conducting a more detailed qualitative study.

Chapter 6

Discussion

In this chapter, I discuss about some of the findings related to Taggy that led to trade-offs and technical challenges. These findings may be relevant for the next level of work on this topic as well as understanding some of the design choices that I made with Taggy. In particular the following topics are discussed:

1. Dealing with absence of necessary context.
2. The impact of incorrect tagging.
3. Selecting a threshold similarity score.
4. Handling attachments and full text search.
5. Project specific vs. project agnostic learning.
6. Handling email replies.
7. Limitations of Taggy.

6.1 Dealing with Absence of Necessary Context

Taggy produces the auto-tagging based on the context and text relevance of emails with user stories. Email will always contain the context since the sender, recipients and time stamp always come with an email. However, for user stories, some or all of the context may be missing. For example, when a user story is just created, it might not be assigned to any developer or planned for an iteration immediately. So, if someone sends an email about this new user story, it is likely to do bad in the relevancy contest compared to other

user stories with the context. Taggy could still pick up such a user story for auto-tagging if the email text shows strong text similarity. Otherwise, Taggy is likely to produce an incorrect auto-tag. This is the principal reason for the incorrect auto-tagging as seen at the quantitative evaluation results.

However, in an agile environment, the developers and customers mostly share knowledge about the work in current or recent iterations. In the evaluation data sets, I have found 70% of the user story related discussions went on during the iteration of the user stories. And 95% of the discussions took place starting a month before to the following month of the iteration. If a team follows this approach, there will be relatively a few number of emails about the off-context or contextless user stories compared to the ones with context.

6.2 The Impact of Incorrect Tagging

Even when the necessary context and text are present, Taggy has a possibility to produce an incorrect auto-tagging. As with most other machine learning techniques, this may be either a result of insufficient training or the mismatch in the actual data and underlying assumptions of Taggy. In case of an incorrect auto-tagging, a human decision can prevail to correct the tagging with the right user stories. However, if Taggy makes a lot of wrong decisions, it might be infeasible to do this manually since a huge number of emails and instant messages are produced. The aforementioned quantitative evaluation provides information about Taggy's accuracy for the evaluation data sets.

In case a wrong tagging is not corrected and left as is, it might still be useful since the email is captured in a shared archive and one can search the archive without looking into other's email inbox.

6.3 Selecting a Threshold Similarity Score

Since Taggy produces a numerical similarity score between an email and a user story, it needs to use a threshold similarity score to discard the ones that do not show “good enough” relevance. Setting this threshold value presents a trade-off since a high value may result in false negative while a low value may yield false positives in auto-tagging.

This trade-off was solved by looking into the data. For example, the similarity scores between the emails and their related a user stories in the training data were observed. It was found that Taggy put a similarity score above or equal 0.58 (out of 1.00) for 90% of the actual email and user story relations in the training data. A value less than 0.58 would produce a large number of false positives and similarly a value above 0.58 would result a large number of false negatives.

This threshold score may be adjusted for a different data set depending on the same principle. However, like other machine learning algorithms, this parameter will not be available without the necessary training data.

6.4 Handling Attachments and Full-Text Search

Matching the email contents against a user story contents is essentially a full-text search problem. Moreover, these high level artifacts often contain attachments of rich files such as Word documents, PDF Files and Spreadsheets. Taggy looks into the attachment contents of a number of rich file types using the third party full-text search engine called Lucene. The full-text search of Lucene also considers language stems, synonyms and other desired advanced features.

Instead of building the full-text search engine, the third party component was used since it is beyond the scope of my research. However, it might be possible to configure the search engine to recognize the important bits about a text. For example, a project

may have a list of words that could provide useful hints about an email's relation with a user story. Such words can be treated specially so that a presence of such words or phrases is weighted higher than others. This avenue is not explored in Taggy and may be considered for a future extension.

6.5 Project Specific vs. Project Agnostic Learning

Since Taggy learns the parameters based on training data, the learning can be done either by project or as a global learning. Per project learning has the potential to provide a tuned parameter set for the project. But this means, a new project cannot get benefit of the auto-tagging since it doesn't have any historical data for training. Learning globally means, once learned, it can be used for other projects. Also, a global learning may learn more patterns since the training data set is larger when multiple projects are combined. On the other hand, this might fail to address any uncommon pattern specific to a project.

The design of Taggy allows both approaches. In case there is historical data for a project, the training can be specific to that project. Otherwise, a new project can use the learned parameters from several projects. For the quantitative evaluation, the training was done globally. So, there was a single set of parameters for all the projects. The training using data set #1 yielded similar relative weights for the projects when a per project training was used. In this case, combining training data from different projects actually improved accuracy since more training data was used to potentially cover more patterns.

6.6 Handling Email Replies

Taggy does not distinguish between a new email and return to an existing email. So, every incoming email goes through a fresh auto-tagging process irrespective of whether

it is a reply email or not. This is because the return email typically contains the same subject and original email along with the new text in the body. So, whatever decision was applied to the original email is likely to be applied to a return email unless there is a significant change in any or some of people, date, subject or content of the email. In the later case, it is logical that this email indeed is a new one and may be a candidate for completely new auto-tagging with different user stories. This is a design trade-off since identifying return emails and applying the same tags as their original emails would be faster compared to recomputing the auto-tagging. But the auto-tagging is run on a background process. So, the cost of rerunning the auto-tagging is less expensive than following the wrong way in case the previous email had inappropriate auto-tagging or the new reply has significant change compared it is predecessors.

6.7 Limitations of Taggy

A number of limitations were observed from the evaluations of Taggy as discussed below:

1. **Real world use.** Taggy was not used in a real world distributed agile project. Despite the quantitative and qualitative evaluations, it remains unknown how it would impact a real world usage.
2. **Email intake.** The CC: based email intake process requires a customer or developer of a project to do this while initiating an email discussion. Although most email senders are aware of adding multiple recipients, the usability of this extra step is not researched.
3. **User interface.** Auto-tagging emails and instant messages means a lot of archived content. To reduce information overload and present the desired content, a proper user interface needs to be in place. Taggy has a web interface that presents the

user stories and the related discussions underneath them and vice versa. In case there are many such emails, this interface needs to be designed to meet the desired user experience. This thesis didn't explore around the appropriate user interface.

4. **Accuracy.** Taggy has shown 76% accuracy based on the evaluation data. The higher the accuracy the better it is to completely reduce human efforts in knowledge management. The accuracy of Taggy limits its usage as a 100% automated knowledge acquisition process.

Chapter 7

Conclusion

In this chapter I draw a conclusion about the thesis. This provides information about the research goals and challenges that are addressed. Also, I discuss about the potential future research on this topic.

7.1 Research Goals Addressed

I have investigated the existing literature and tool support for communication and collaboration among stakeholders in distributed agile projects. I have found knowledge sharing takes place in one of the two distinct levels, i) knowledge sharing about high level artifacts such as user stories, bugs etc. and ii) knowledge sharing about low level artifacts such as source code, design, build etc. The concentration of this thesis was on the high level artifacts where both customers and developers are involved. Looking into this category, I found the following key points:

- Distributed agile teams commonly use text based knowledge sharing.
- Email is the first preference of the customers.
- Whenever possible, instant message is the dominant synchronous communication tool.
- The knowledge in emails and instant messages are rarely centralized for future reference.

Based on these findings, I identified that if it is possible to capture the fragmented knowledge across the emails and instant messages without much human effort, it might

work as a knowledge center for future reference. To solve this problem, I have researched existing tool support and combined some of the existing techniques and designed a new machine learning based solution to automatically tag the emails and instant messages with the user stories. The technique in short employs the following:

- Learns the parameters of a similarity function based on training sample.
- Automatically grabs project related emails and instant messages.
- Attempts to tag the emails and instant messages with the user stories based on context and text relevance.

To find the technical feasibility of this technique, I have also implemented a proof of concept tool called Taggy. The implementation demonstrates an end-to-end auto-tagging solution. This prototype implementation helped me to fine tune the auto-tagging technique. In the end I identified the following attributes to be significant in deciding the auto-tagging relevance:

- Associated people in the artifacts.
- The temporal relevance.
- The subject or heading.
- The text content.

Taggy was trained and evaluated using real life software project data. The data were collected from two different sources. In total the evaluation data consists of 4,745 emails from 9 real life agile project teams. This helped me to identify the accuracy and statistical significance of the auto-tagging performance. Using this data, Taggy has shown an average accuracy of 76% and a chi-square test shows this result is statistically significant.

In addition to the quantitative evaluation, I have also conducted a preliminary user study of Taggy. A total of 6 participants from the industry and academia evaluated the tool after trying its features themselves. They provided encouraging feedbacks and suggestions about Taggy. The ability to centralize the knowledge from different sources was identified as the most potential value addition of Taggy. They identified new team members as the key consumers of the auto-tagged information from emails and instant messages. Some of their suggestions are listed as potential future work on this topic.

This thesis also unveiled some of the technical challenges and design trade-offs associated with auto-tagging. In a nutshell, the auto-tagging process needs to deal with the standard challenges of information retrieval and the limitations of machine learning. I have discussed a few approaches that can be leveraged to support rich attachments, decide a threshold similarity score and using auto-tagging for projects without historical data. I have also provided the list of limitations, some of which can form a future research on this topic.

Essentially, the concept of email auto-tagging with user stories can be extended to other domains with similar characteristics. In this thesis I have discussed about the architectural and some of the key algorithmic structure of Taggy. This information can be used to reproduce the same solution or adapt the solution to a problem in a different domain based on the same core concepts.

Overall, a distributed agile team can utilize the auto-tagging technique of Taggy to retain informal but important knowledge from the emails and instant messages in the organic form with little human effort. This organic knowledge center can support the long term success of a team as it needs to undergo changes in team composition to maintain and extend its product.

7.2 Future Work

While working on Taggy and conducting the evaluations, I have explored several opportunities that might be addressed in a future research. The two core directions that I have identified are, i) extensions with new features and ii) improvements through evaluation and fine tuning. A few potential future work on the two directions are discussed next.

7.2.1 Extensions with New Features

A future research taking my work as a foundation can lead to a number of extensions. Here I provided a short list of such potential extensions:

- **Auto-tag other artifacts.** While email and instant messages are the two most utilized knowledge sharing mediums, they are mostly about the high level artifacts. An extension might look into auto-tagging of the low level artifacts, such as source code, automated build and test results etc. Engineers and technical people will be the potential consumers of this knowledge. Some of the core concepts presented in Taggy, such as combining context with text relevance, can also be applied to the low level artifacts.
- **User interface.** The user interface of Taggy is similar to the ones used in most online collaboration tools, where a list of messages appears underneath the user story. However, when emails and instant messages are used, there is a potential of producing a huge volume of content. To ensure a seamless user experience, a suitable user interface needs to be explored. Otherwise, it might be the cause of information overload.
- **Multiple intake methods.** Presently the only email intake process is via a project email inbox. Similarly, the only intake process for instant messages is a Skype

plugin. A future work may explore other viable intake methods and possibly use a combination of multiple channels to ease the process of knowledge feeding. It is apparent that, a fluid intake process will encourage the end users to utilize the tool.

- **Pluggable solution.** Presently Taggy is a standalone system. However, the people in distributed agile teams use different tools from different vendors. To reach the potential users, Taggy needs to offer a pluggable solution so that, if desired, teams can easily plug-in Taggy to their existing project management tools. In a sense, the auto-tagging solution needs to seamlessly fit in with the existing project management and collaboration tools to allow people to use it without abandoning their favorite tools.

7.2.2 Improvements through Evaluation

A greater level of evaluation is needed to discover the concerns and potential improvements of the Taggy features. Here I provide a list of potential evaluation scenarios:

- **Real life use.** The tool needs to be used in real life projects for a period of time. While using the tool on a day to day basis, it is possible to find out the usability of the approach as well as social impact of the tool. Once a team uses it for a while, it is possible to collect data about why, who, when and how they use the archived emails and instant messages. Based on the findings, the approach can be adapted to better suit the usage pattern.
- **Quantitative evaluation:** With a wider variety of data sources, the auto-tagging process can be adjusted to support different communication patterns that may not be observed with the existing data sets. Also, my current work did not evaluate the accuracy of instant message auto-tagging. A future work may focus on improving the accuracy of the auto-tagging process using data from more diverse sources.

To conclude, this thesis presents a technique with an implementation of a lightweight organic knowledge center for distributed agile teams. This can be used as a foundation for future work in this research area.

Chapter 8

Ethics Approval

The ethics approval for conducting the qualitative evaluation of Taggy is given at Figure 8.1:



UNIVERSITY OF
CALGARY

MEMO

CONJOINT FACULTIES RESEARCH ETHICS BOARD
c/o Research Services
Main Floor, Energy Resources Research Building
3512 - 33 Street N.W., Calgary, Alberta T2L 1Y7
Telephone: (403) 220-3782
Fax: (403) 289 0693
Email: csjahrau@ucalgary.ca
Thursday, August 19, 2010

To: Frank Maurer
Computer Science

From: Dr. Glen Bodner, Acting Chair
Conjoint Faculties Research Ethics Board (CFREB)

Re: Certification of Institutional Ethics Review: Effectiveness of Auto-Tagging Emails, IM, Wiki and Files with User Stories Based on Project Context

The above named research protocol has been granted ethical approval by the Conjoint Faculties Research Ethics Board for the University of Calgary. Enclosed are the original, and one copy, of a signed **Certification of Institutional Ethics Review**. Please note the terms and conditions that apply to your Certification. If the research is funded, the sponsor should be notified, and the original certificate sent to them for their files. The copy is for your records. The Conjoint Faculties Research Ethics Board will retain a copy of the Certification on your file.

Please note, an annual/progress/final report must be filed with the CFREB twelve months from the date on your ethics clearance. A form for this purpose has been created, and may be found on the "Ethics" website, <http://www.ucalgary.ca/research/compliance/ethics/renewal>

In closing let me take this opportunity to wish you the best of luck in your research endeavor.

Sincerely,

Cari Jahraus
For:
Glen Bodner, Ph.D., Department of Psychology and
Acting Chair, Conjoint Faculties Research Ethics Board

Enclosures(2)

Figure 8.1: Ethics Approval

Bibliography

- [1] Dinesh Batra. Modified agile practices for outsourced software projects. *Commun. ACM*, 52(9):143–148, 2009.
- [2] Lucas Layman, Laurie Williams, Daniela Damian, and Hynek Bures. Essential communication practices for extreme programming in a global software development team. *Information and Software Technology*, 48(9):781–794, September 2006.
- [3] Bj Decker, Eric Ras, J Rech, Pascal Jaubert, and Marco Rieth. Wiki-Based stakeholder participation in requirements engineering. *IEEE Software*, 24(2):28–35, 2007.
- [4] Eclipse mylyn open source project. <http://www.eclipse.org/mylyn/>.
- [5] An agile / scrum project management tool | ScrumPad. <http://www.scrumpad.com/>.
- [6] Manifesto for agile software development. <http://agilemanifesto.org/>.
- [7] Kent Beck. *Extreme programming eXplained : embrace change*. Addison-Wesley, Reading MA, 2000.
- [8] Ken Schwaber. *Agile software development with Scrum*. Prentice Hall, Upper Saddle River NJ, 2002.
- [9] Scott Ambler. *Agile modeling : effective practices for eXtreme programming and the unified process*. J. Wiley, New York, 2002.
- [10] Mike Cohn. *User stories applied : for agile software development*. Addison-Wesley, Boston, 2004.
- [11] Ron Jeffries. *Extreme programming installed*. Addison-Wesley, Boston, 2001.

- [12] Rachel Davies. *Agile coaching*. Pragmatic Bookshelf, Raleigh N.C., 2009.
- [13] William Wake. *Extreme programming explored*. Addison Wesley, Boston, 2002.
- [14] Hubert Baumeister, Michele Marchesi, Mike Holcombe, Keith Braithwaite, and Tim Joyce. XP expanded: Distributed extreme programming. In *Extreme Programming and Agile Processes in Software Engineering*, volume 3556 of *Lecture Notes in Computer Science*, pages 180–188. Springer Berlin / Heidelberg, 2005.
- [15] Jane M. Robarts. Practical considerations for distributed agile projects. In *AGILE Conference*, volume 0, pages 327–332, Los Alamitos, CA, USA, 2008. IEEE Computer Society.
- [16] J. Sutherland, G. Schoonheim, E. Rustenburg, and M. Rijk. Fully distributed scrum: The secret sauce for hyperproductive offshored development teams. In *Agile, 2008. AGILE '08. Conference*, pages 339–344, 2008.
- [17] Brian Scott Drummond and John Francis "JF" Unson. Yahoo! distributed agile: Notes from the world over. In *AGILE Conference*, volume 0, pages 315–321, Los Alamitos, CA, USA, 2008. IEEE Computer Society.
- [18] Agile project management tool - VersionOne.
http://pm.versionone.com/Trial_AgileProjects.html?c-aws=p&gr-versionone&v-004&gclid=CIbC9bjhy6QCFQwDbAodgnTfEQ.
- [19] XPlanner home. <http://www.xplanner.org/>.
- [20] The trac project. <http://trac.edgewall.org/>.
- [21] Andrea De Lucia, Filomena Ferrucci, and Filippo Lanubile. Collaboration in distributed software development. In *Software Engineering*, volume 5413 of *Lecture Notes in Computer Science*, pages 174–193. Springer Berlin / Heidelberg, 2009.

- [22] Marcelo Cataldo, Matthew Bass, James D. Herbsleb, and Len Bass. On coordination mechanisms in global software development. In *Global Software Engineering, International Conference on*, volume 0, pages 71–80, Los Alamitos, CA, USA, 2007. IEEE Computer Society.
- [23] B. Chatters. Implementing an experience factory: maintenance and evolution of the software and systems development process. In *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*, pages 146–151, 1999.
- [24] S. M. Sohan, Michael M. Richter, and Frank Maurer. Auto-tagging emails with user stories using project context. In *Agile Processes in Software Engineering and Extreme Programming*, volume 48 of *Lecture Notes in Business Information Processing*, pages 103–116. Springer Berlin Heidelberg, 2010.
- [25] Yevgeniy "Eugene" Medynskiy, Nicolas Ducheneaut, and Ayman Farahat. Using hybrid networks for the analysis of online software development communities. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 513–516, Montral, Qubec, Canada, 2006. ACM.
- [26] Davor ubrani and Gail C. Murphy. Hipikat: recommending pertinent software development artifacts. In *Proceedings of the 25th International Conference on Software Engineering*, pages 408–418, Portland, Oregon, 2003. IEEE Computer Society.
- [27] Davor Cubranic, Gail C. Murphy, Janice Singer, and Kellogg S. Booth. Hipikat: A project memory for software development. *IEEE Transactions on Software Engineering*, 31(6):446–465, 2005.
- [28] M. Rita Thissen, Jean M. Page, Madhavi C. Bharathi, and Toyia L. Austin. Communication tools for distributed software development teams. In *Proceedings of the*

- 2007 ACM SIGMIS CPR conference on Computer personnel research: The global information technology workforce*, pages 28–35, St. Louis, Missouri, USA, 2007. ACM.
- [29] M. Korkala and P. Abrahamsson. Communication in distributed agile development: A case study. In *Software Engineering and Advanced Applications, 2007. 33rd EUROMICRO Conference on*, pages 203–210, 2007.
 - [30] Rory V. OConnor, Nathan Baddoo, Kari Smolander, Richard Messnarz, Steinar Hole, and Nils Brede Moe. A case study of coordination in distributed agile software development. In *Software Process Improvement*, volume 16 of *Communications in Computer and Information Science*, pages 189–200. Springer Berlin Heidelberg, 2008.
 - [31] T. Hildenbrand, M. Geisser, T. Kude, D. Bruch, and T. Acker. Agile methodologies for distributed collaborative development of enterprise applications. In *Complex, Intelligent and Software Intensive Systems, 2008. CISIS 2008. International Conference on*, pages 540–545, 2008.
 - [32] Lucy M. Berlin, Robin Jeffries, Vicki L. O’Day, Andreas Paepcke, and Cathleen Wharton. Where did you put it? issues in the design and use of a group memory. In *Proceedings of the INTERACT ’93 and CHI ’93 conference on Human factors in computing systems*, pages 23–30, Amsterdam, The Netherlands, 1993. ACM.
 - [33] A. Egyed and P. Grunbacher. Automating requirements traceability: Beyond the record & replay paradigm. In *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on*, pages 163–171, 2002.
 - [34] Wikipedia. <http://www.wikipedia.org/>.
 - [35] W. Lohstroh, R. J. Westerwaal, B. Noheda, S. Enache, I. A. M. E. Giebels, B. Dam,

- and R. Griessen. Self-Organized layered hydrogenation in black Mg₂NiH_x switchable mirrors. *Physical Review Letters*, 93(19):197404, November 2004.
- [36] M. Tasic, V. Milicevic, and M. Stankovic. Collaborative knowledge acquisition for agile project management. In *Computer as a Tool, 2005. EUROCON 2005. The International Conference on*, volume 2, pages 1081–1084, 2005.
- [37] Isabel Cruz, Stefan Decker, Dean Allemang, Chris Preist, Daniel Schwabe, Peter Mika, Mike Uschold, Lora Aroyo, Sren Auer, Sebastian Dietzold, and Thomas Riechert. OntoWiki a tool for social, semantic collaboration. In *The Semantic Web - ISWC 2006*, volume 4273 of *Lecture Notes in Computer Science*, pages 736–749. Springer Berlin / Heidelberg, 2006.
- [38] Thomas Chau and Frank Maurer. A case study of wiki-based experience repository at a medium-sized software company. In *Proceedings of the 3rd international conference on Knowledge capture*, pages 185–186, Banff, Alberta, Canada, 2005. ACM.
- [39] Fitnesse Acceptance Testing Tool. <http://fitnesse.org/>.
- [40] Cynick Young and Hiroki Terashima. How did we adapt agile processes to our distributed development? In *AGILE Conference*, volume 0, pages 304–309, Los Alamitos, CA, USA, 2008. IEEE Computer Society.
- [41] Takahira Yamaguchi, Walid Maalej, and Hans-Jrg Happel. A lightweight approach for knowledge sharing in distributed software teams. In *Practical Aspects of Knowledge Management*, volume 5345 of *Lecture Notes in Computer Science*, pages 14–25. Springer Berlin / Heidelberg, 2008.
- [42] Jonathan I. Maletic and Andrian Marcus. Supporting program comprehension using semantic and structural information. In *Proceedings of the 23rd International Con-*

- ference on Software Engineering*, pages 103–112, Toronto, Ontario, Canada, 2001. IEEE Computer Society.
- [43] Martin P. Robillard and Gail C. Murphy. FEAT: a tool for locating, describing, and analyzing concerns in source code. In *Proceedings of the 25th International Conference on Software Engineering*, pages 822–823, Portland, Oregon, 2003. IEEE Computer Society.
- [44] Agile project management tool - mingle - for agile software development | agile ALM | ThoughtWorks studios. <http://www.thoughtworks-studios.com/mingle-agile-project-management>.
- [45] ScrumWorks - CollabNet, inc. <http://danube.com/scrumworks/pro>.
- [46] IBM software - rational team concert. <http://www-01.ibm.com/software/awdtools/rtc/>.
- [47] Bug, issue and project tracking for software development - JIRA. <http://www.atlassian.com/software/jira/>.
- [48] Bugzilla :: bugzilla.org. <http://www.bugzilla.org/>.
- [49] Fog Creek Software and Happy Cog Studios <http://www.happycog.com>. FogBugz - bug & issue tracking, project management, help desk software. <http://www.fogcreek.com/fogbugz/>.
- [50] Dane Bertram, Amy Voids, Saul Greenberg, and Robert Walker. Communication, collaboration, and bugs: the social nature of issue tracking in small, collocated teams. In *Proceedings of the 2010 ACM conference on Computer supported cooperative work*, pages 291–300, Savannah, Georgia, USA, 2010. ACM.

- [51] Project management software, online collaboration: Basecamp.
<http://basecamphq.com/>.
- [52] Teambox - collaboration software. <http://teambox.com/>.
- [53] Business collaboration platform for the enterprise and the internet - SharePoint 2010. <http://sharepoint.microsoft.com/en-us/Pages/default.aspx>.
- [54] Secure source code hosting and collaborative development - GitHub.
<http://github.com/>.
- [55] CodePlex - open source project hosting. <http://www.codeplex.com/>.
- [56] Hudson CI. <http://hudson-ci.org/>.
- [57] CruiseControl home. <http://cruisecontrol.sourceforge.net/>.
- [58] Team foundation server 2010. <http://msdn.microsoft.com/en-us/vstudio/ff637362.aspx>.
- [59] Skype. <http://www.skype.com/intl/en-us/home>.
- [60] M. M. Richter. *Knowledge Containers*. Morgan Kaufmann Publishers, 2003.
- [61] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620, 1975.
- [62] The twitter engineering blog: Twitter’s new search architecture.
<http://engineering.twitter.com/2010/10/twitters-new-search-architecture.html>.
- [63] Richard Sutton. *Reinforcement learning : an introduction*. MIT Press, Cambridge Mass., 1998.
- [64] Ruby programming language. <http://www.ruby-lang.org/en/>.

- [65] Ruby on rails. <http://rubyonrails.org/>.
- [66] Apache lucene - overview. <http://lucene.apache.org/java/docs/index.html>.
- [67] Microsoft sql server. <http://www.microsoft.com/sqlserver>.
- [68] Apache solr. <http://lucene.apache.org/solr/>.
- [69] Skype4Py. <http://skype4py.sourceforge.net/doc/html/>.
- [70] crontab.org - CRONTAB. <http://crontab.org/>.
- [71] Windows scheduled tasks. <http://support.microsoft.com/kb/308569>.
- [72] Active record – object-relation mapping put on rails. <http://ar.rubyonrails.org/>.
- [73] Herman Chernoff and E. L. Lehmann. The use of maximum likelihood estimates in 2 tests for goodness of fit. *The Annals of Mathematical Statistics*, 25(3):579–586, September 1954. ArticleType: research-article / Full publication date: Sep., 1954 / Copyright 1954 Institute of Mathematical Statistics.
- [74] Anselm Strauss. *Basics of qualitative research : techniques and procedures for developing grounded theory*. Sage Publications, Thousand Oaks CA, 2nd ed. edition, 1998.