Chapter 6
Leveraging Web 2.0 for software evolution

Yuan Tian and David Lo

Summary. In this era of Web 2.0, much information is available on the Internet. Software forums, mailing lists, and question-and-answer sites contain lots of technical information. Blogs contain developers' opinions, ideas, and descriptions of their day-to-day activities. Microblogs contain recent and popular software news. Software forges contain records of socio-technical interactions of developers. All these resources could potentially be leveraged to help developers in performing software evolution activities. In this chapter, we first present information that is available from these Web 2.0 resources. We then introduce empirical studies that investigate how developers contribute information to and use these resources. Next, we elaborate on recent technologies and tools that could mine pieces of information from these resources to aid developers in performing their software evolution activities. We especially present tools that support information search, information discovery, and project management activities by analyzing software forums, mailing lists, question-and-answer sites, microblogs, and software forges. We also briefly highlight open problems and potential future work in this new and promising research area of leveraging Web 2.0 to improve software evolution activities.

Much of the content of this chapter is a summary of, and is based on, the following articles having either one or both of the chapter authors as co-authors: [3, 343, 703, 805, 806, 841, 845]. We would like to thank the other co-authors of these articles: Swapna Gottipati, Jing Jiang, Ferdian Thung, Lingxiao Jiang, Didi Surian, Nian Liu, Hanghang Tong, Ee-Peng Lim, Christos Faloutsos, Hong Cheng, Achananuparp Palakorn, and Philips Kokoh Prasetyo.

T. Mens et al. (eds.), *Evolving Software Systems*,
DOI 10.1007/978-3-642-45398-4_6, © Springer-Verlag Berlin Heidelberg 2014

6.1 Introduction

Web 2.0 has revolutionized the use of web sites [667]. Prior to Web 2.0, most web sites were simply static pages that did not support much user interactions. With Web 2.0, users can post new content dynamically, update a long list of friends in real time, collaborate with one another, and more. This has changed the paradigm of how users use the Web. Web 2.0 sites include but are not limited to blogs, microblogging sites, social networking sites, and sharing and collaboration sites. Currently, most web users spend a substantial amount of time in Web 2.0 sites and the adoption of Web 2.0 sites is growing [235]. Much knowledge is shared in these Web 2.0 sites.

Web 2.0 also affects software developers. Currently, developers share a lot of information in Web 2.0 sites. These resources could be leveraged to help developers perform their tasks better. It is common for developers to consult information sharing sites like software forums, e.g., CNET[1], Oracle OTN forum[2], SoftwareTipsandTricks[3], and DZone[4]. Developers often encounter the same problems, and solutions found by one developer are disseminated to others via these sites. Developers of open source projects such as GNOME[5] use mailing lists as discussion forums to support communication and cooperation in project communities. Besides software forums and mailing lists, developers often turn to question-and-answer sites, such as StackOverflow[6], to seek help from other expert developers about software-related problems that they face. Blogging services are also popular among software developers. Developers use blogs to record knowledge including their ideas and experience gained during software evolution tasks. This knowledge could be discovered by other developers through web search. In the recent few years, with the advent of social media, developers have yet another means to share and receive information. Much information is shared via microblogging sites such as Twitter [863] and Sina Weibo [7]. Information shared in microblogging sites is often recent and informal. The unique nature of information in these sites makes them interesting resources that augment other Web 2.0 resources. Developers also use many collaboration sites to jointly work together to complete various software projects. These collaboration sites, also referred to as software forges, can host tens of thousands of projects or even more. Examples of these collaboration sites (or software forges) are GitHub[8] and SourceForge[9].

Web 2.0 can be leveraged to assist developers find information that help them in software evolution tasks. Developers often want to find answers to various develop-

[1] forums.cnet.com
[2] forums.sun.com/index.jspa
[3] www.softwaretipsandtricks.com/forum
[4] java.dzone.com
[5] www.gnome.org
[6] stackoverflow.com
[7] www.weibo.com
[8] github.com
[9] sourceforge.net

ment questions. Software forums often contain such answers. However, it can take much time for developers to sift the mass of contents shared there to find desired answers. Developers also often want to reuse programs satisfying some properties. Information stored in Web 2.0 resources can be leveraged for this task. For the above mentioned tasks, automation is needed to bring relevant pieces of information to developers.

Developers can also leverage Web 2.0 to discover new and interesting knowledge. For example, developers often want to be notified of new patches to security loop holes, new useful libraries, new releases of some libraries, new programming tips, and many other pieces of information. Today, developers often need to manually search for such new information by leveraging search engines or reading slightly outdated information from magazines or books. The more recent these pieces of information are, the more useful they are to developers, e.g., developers could quickly patch new security loop holes before they get exploited. For these purposes, microblogging sites are promising sources of information as information shared there is often recent and informal, providing timely and honest feedback on many recent issues that various developers find interesting.

Developers often work together in various open source projects hosted in many software forges.[10] Based on projects that developers have worked on, we can build various support tools that improve the management of projects in these software forges [805, 806]. These project management support tools can help to better allocate resources (i.e., developers) to tasks (i.e., projects) [805] or predict bad situations (e.g., project failures) such that appropriate mitigation actions can be taken [806]. In practice, often such project management tasks are done manually and thus support tools could reduce the amount of manual effort needed.

In this chapter, we first present information available in various Web 2.0 resources in Section 6.2. We then describe how developers use these Web 2.0 resources in Section 6.3. Next, we review recently proposed automated techniques that leverage Web 2.0 resources for information search, information discovery, and project management. We start by summarizing past studies that leverage Web 2.0 for information search including studies on answering software development questions and searching for similar projects [343, 841] in Section 6.4. We then summarize past studies that leverage Web 2.0 sites for information discovery, e.g., [3, 703], in Section 6.5. We also present past studies that leverage Web 2.0 to support project management, e.g., [805, 806], in Section 6.6. We describe open problems and future work in Section 6.7. Finally, we conclude in Section 6.8.

[10] Please refer to Chapter 10 of this book

6.2 Web 2.0 Resources

There are various Web 2.0 resources that software developers often use to learn and exchange information and knowledge. In this section, we highlight several of these resources.

6.2.1 Software Forums, Mailing Lists and Q&A Sites

Developers often ask and discuss questions in software forums. Other more experienced developers that face the same problems can reply with some answers. These exchanges of information are documented and thus other developers facing similar problems in the future can also benefit. There are many software forums available online. Some forums are specialized such as Oracle OTN forum and DZone. Some others are general purpose such as CNET and SoftwareTipsandTricks. Figure 6.1 shows the Oracle OTN Forum where many people discuss Java programming.

A software forum is often organized into categories. For example, Figure 6.1 shows that inside Oracle OTN forum, questions and answers are categorized into: Java Essentials, Java API, etc. Inside Java Essentials, there are more sub-categories: New to Java, Java Programming, and Training / Learning / Certification. Within each sub-category, there are many threads. Within each thread, there are many posts. At times a thread can contain even hundreds of posts. The posts contain questions, answers, or other pieces of information (e.g., positive feedbacks, negative feedbacks, junk, etc).

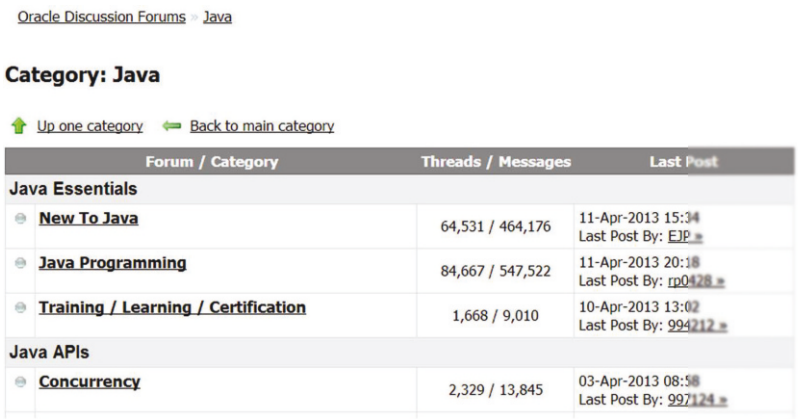

Fig. 6.1: **Oracle OTN Forum**

Developers of open source projects use mailing lists to communicate and collaborate with one another. A *mailing list* works as a public forum for developers or users who have subscribed to the list. Anyone in the list can post messages to other people in the list by sending emails to a public account. Contents of these messages are usually related with changes made by developers or problems faced by developers or users during software development or product usage. For example, from *GNOME* website developers and users can get information about each mailing list and decide whether to subscribe to one or more lists.[11] These lists are created for various purpose: some are created for messages related to particular modules (e.g., "anjuta-devel-list" is for messages related to Anjuta IDE), some are created for messages related to special events (e.g., "asia-summit-list" is for messages related to GNOME.Asia Summit organization), some are created not for developers but for end users (e.g., "anjuta-list" is for messages from users of Anjuta IDE), etc.

Developers also can seek answers for questions from question-and-answer sites. In general question-and-answer sites like Yahoo! Answers, people can ask questions about various domains including news, education, computer & internet, etc. StackExchange[12] is a fast-growing network which contains 104 domain specific question-and-answer sites focusing on diverse topics from software programming to mathematics and IT security. Among the 104 question-and-answer sites, *StackOverflow* is the biggest and most famous one. It has more than 5 millions questions (most of them are related to software development) and more than 2 millions users since it was launched in 2008.

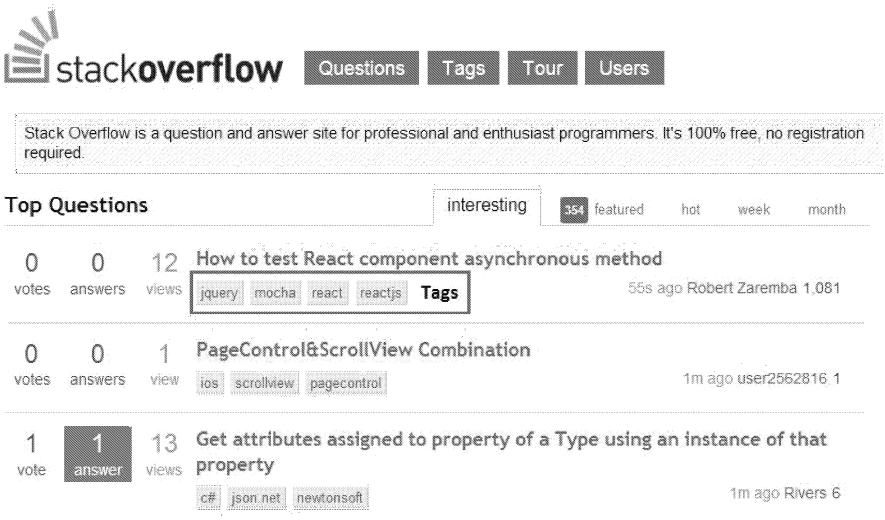

Fig. 6.2: **Question-and-Answer Threads in Stack Overflow**

[11] mail.gnome.org/mailman/listinfo

[12] stackexchange.com

Figure 6.2 shows question-and-answer threads extracted from StackOverflow. Each question in StackOverflow has a short title that briefly summarizes the question. A user who asks the question can assign several *tags* like "java" according to the topics of the question. These tags are used to help users to search for similar questions and their answers. A user who views a question can vote up the question if he/she thinks the question is useful or vote down it if he/she thinks the question is unclear. The sum of votes, the number of people who has viewed the question, and the total number of provided answers are recorded for each question.

6.2.2 Software Blogs & Microblogs

Blogging is one of the typical features of the Web 2.0 era. Similar with home pages, *blogs* are created for individuals. They record personal thinking, experience and stories in a diary-like format. Different from home pages, blogs' contents change more often and blogs support discussions by allowing others to post comments. In addition, the *RSS (Really Simple Syndication)* technology allows people to not only link to a page containing a blog but also to subscribe to it. People who have subscribed to a blog will be informed if the blog's content has been modified.

Developers are using blogs to share their ideas, knowledge, and experiences on software development. Usually people find others' blogs through web search. For instance, a developer has encountered a problem but lacks experience to solve the problem; he or she might go to Google Search and seek for solutions using some keywords. Some of the top returned search results may link to other developers' blogs where the ways to solve the same or similar problem are presented. By this means, blogs provide knowledge for the whole community of software developers.

In the recent years, microblogging services are getting popular. Millions of people communicate with one another by broadcasting short messages which are made public to all. Different from traditional blogging services and other social media, microblogs are usually short and often contain information of very recent news and events; microblogs are also informal in nature and microbloggers are often unafraid to express their honest opinions about various topics. Software developers also make use of this new social media trend. Thus, one could potentially discover various information from microblogs, e.g., new features of a library, new methodologies to develop software systems, new conferences, new security loop holes, etc. The informal and timely nature of microblogs suit software development well. In software development, many new "events", e.g., releases of new libraries, etc., occur from time to time. Developers could learn from the wisdom of the masses that are available in the millions of microblogs about various software relevant topics. Furthermore, a number of studies have shown the important role of informal communication [108, 343, 675]. Microblogging is yet another kind of informal communication. Microbloggers can express honest opinions about various libraries, programming languages, etc. through their microblogs that are available for all to see.

Fig. 6.3: **Microblogs in Twitter**

Figure 6.3 shows some sample microblogs (a.k.a. tweets) from Twitter, which is arguably the largest microblogging site. Microbloggers can post short contents (at most 140 characters in Twitter) that would then be broadcasted to those that subscribe to it. These microblogs are also publicly available for all to see. A microblogger can subscribe to (i.e., follow in Twitter) other microbloggers and get notified whenever new microblogs are generated. A microblogger can also forward microblogs to (i.e., retweet in Twitter) others, as well as reply to others' microblogs. Microblogs can be tagged with particular keywords (i.e., *hashtags* in Twitter). For instance, the microblogs in Figure 6.3 are all tagged with hashtag #csharp.

Developers can include various contents in their microblogs. For example, from Figure 6.3, the first microblogger shares a tip on visitor pattern. The second microblogger asks a question, while the third microblogger broadcasts a personal message on what he is currently doing.

6.2.3 Software Forges

With the advent of Web 2.0, developers can choose to work on various projects with many collaborators across the globe. Software forges provide additional support for this. A software forge, e.g., Google Code, SourceForge, GitHub, etc., hosts hundreds or even hundreds of thousands of projects. Developers can view these projects, be aware of development activities happening in them, download and use the projects,

and also contribute code, test cases, bug reports, etc. to the projects. Figure 6.4 shows some projects that are hosted in SourceForge.

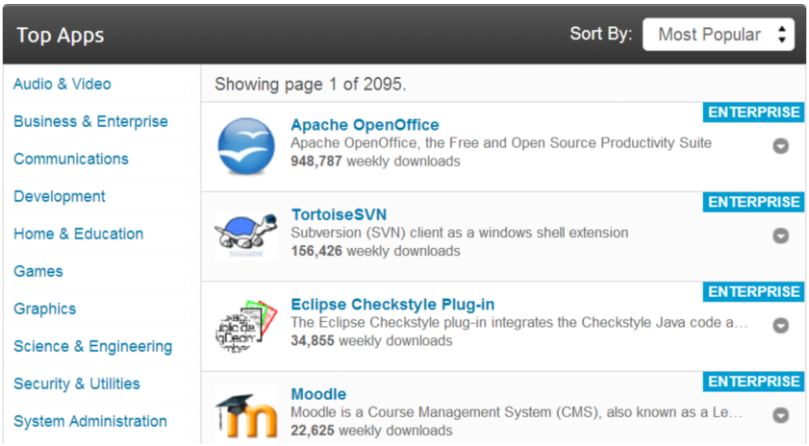

Fig. 6.4: **Projects in SourceForge**

Software forges often provide *APIs* for others to query activities that happen within them. With these APIs, much information can be gathered. We can track how developers work with others across time. We can track the number of downloads. We can also track new projects and new collaborations that are created over time. Chapter 10 of this book describes the evolution of projects in software forges.

6.2.4 Other Resources

There are also other Web 2.0 resources like LinkedIn[13], Facebook[14], Wikipedia[15], Academic.edu[16], Foursquare[17], Google Talk[18], and many more. Many of these resources often contain information about subgroups devoted to specific topics including software evolution. For example LinkedIn and Facebook contain profiles of many software developers. Wikipedia defines many software evolution related terms. Academic.edu shares research studies including those related to software evolution. Foursquare provides geospatial locations of people including those of

[13] www.linkedin.com
[14] www.facebook.com
[15] www.wikipedia.org
[16] academia.edu
[17] foursquare.com
[18] support.google.com/talk/?hl=en

software developers. These resources also provide a wealth of information that can potentially be leveraged to improve software evolution activities.

6.3 Empirical Studies

In this section, we review several empirical studies that investigate how Web 2.0 resources have been used by software developers. We categorize these studies according to resources that they target, namely: software forums, mailing lists & question-and-answer sites, software blogs & microblogs, and software forges.

6.3.1 Software Forums, Mailing Lists and Q&A Sites

Rupakheti and Hou analyzed 150 discussion threads from Java Swing forum [738]. They found that *API problems* recur in software forums. This phenomenon was leveraged to design an *API critic* which advises how an API should be used. They manually categorized the threads into unclear threads, other threads (not related to layout and composition), application specific requirement threads, and threads that would benefit from the automated API critic.

Bird et al. constructed a *social network* from a mailing list [108]. Each node in the network is an individual and there is an edge between a to b if b replied to a message that is generated by a. They analyzed more than 100,000 messages from *Apache HTTP Server's* developer mailing list. They found that the out-degree (i.e., the number of people that replies to a person) and in-degree (i.e., the number of people to whom a person has replied to) follow *power-law distributions*. They also found that the level of activity in a mailing list is strongly related to the level of activity in the source code.

Sowe et al. investigated knowledge sharing interactions among knowledge providers and knowledge seekers in the mailing lists of the Debian project [789]. A knowledge provider is an expert that helps other participants in the project. A knowledge seeker refers to any participant who asks questions related to software development or software usage. They collected messages and replies generated by 3735 developers and 5970 users. They found that knowledge providers and knowledge seekers interact and share knowledge a lot. Developers generate more replies than messages while users generated more messages than replies.

Treude et al. analyzed how developers use question-and-answer sites like *StackOverflow* [854]. The authors collected 15 days of question-and-answer threads and manually analyzed a smaller set of 385 questions that are randomly sampled from the collected threads. They divided these questions into different groups and found that questions that ask for instructions are the most popular questions. They also found that questions that ask for code review, the questions made by novices, and abstract questions are answered more frequently than other types of questions.

Nasehi et al. compared high quality and low quality answers on *StackOverflow* to learn important attributes of good code examples [634]. An answer is of high quality if it has been accepted by the asker or it has a relatively high voting score. The authors sampled 163 question-and-answer threads. Each of them has at least one answer that contains a code example and receives 4 or more points. They summarized that high quality answers usually contain concise code example, use the context of the question, highlight important elements, give step-by-step solutions, and provide links to extra resources.

6.3.2 Software Blogs & Microblogs

Pagano and Maalej investigated how software developers use blogging services [675]. They collected blog posts generated by 11,00 developers from four open source project communities, namely Eclipse, *GNOME*, PostgreSQL, and Python. The authors matched the bloggers' identities to source code committers. They found that blogggers who are also committers post more frequently than single bloggers who never commit changes to the source code. They reported that commits are frequently, short, and precise while blog posts are less frequent, longer (14 times longer than commit), and contain less source code. They also found that developers are more likely to post blogs after corrective engineering or management tasks than after forward engineering or re-engineering tasks.

Parnin and Treude studied blog posts that are related to *API documentations* [680]. In their work, they collected developers' blog posts by performing Google searches for all methods' names in the jQuery API. They then manually categorized the top-10 returned search results and found that blog posts cover 87.9% of the API methods. They also found that tutorials and experience reports are the most common types of blog posts. A tutorial often describes problems to be solved and shows solutions in detailed steps. An experience report describes the experience gained from handling a problem. Different from the findings reported by Pagano and Maalej that only 1.8% of blog posts contain source code, this work found that 90% of the blog posts mentioning API methods contain code snippets. The authors concluded that these API related blog posts are used to: describe a philosophy of a design approach or a problem, support a niche community, and store information for bloggers' future personal use.

Parnin et al. investigated the motivation and challenges of blogging developer knowledge [681]. They extracted 55 blogs by performing Google searches using keywords related to three technology areas: *IDE plugin* development, mobile development, and web development.[19] They sent a survey to each of the authors of the 55 blogs and collected 30 responses in the end. They found that developers blog because it can help them educate employees, gain personal reputation, document their experiments, and get feedback that can be used to improve their code or product.

[19] They choose Eclipse and Visual Studio plugins, Android and iPhone development, and Django and jQuery development as representatives.

They also summarized that the biggest challenges for developers to use blogging services are time and the lack of a reward system. For instance, it takes much time for authors to write a high quality blog; it is also time consuming to manage all blog posts, such as porting blogs between systems and filter spam comments.

Bougie et al. conducted an empirical work to understand how developers use Twitter to support communication in their community and what they talk about on Twitter [136]. They sampled 68 developers from three project communities: Linux, Eclipse, and MXUnit. By analyzing 600 microblogs generated by these 68 microbloggers and comparing them with microblogs generated by normal Twitter users, they found that microblogs generated by sampled developers contain more conversations and information sharing. They categorized these 600 microblogs into four categories: software engineering-related, gadgets and technological topics, current events outside technical topics, and daily chatter.

We and a few others extended the work by Bougie et al.'s [845]. We analyzed 300 microblogs (a.k.a. tweets) that are tagged with software related *hashtags* (i.e., #csharp, #java, #javascript, #dotnet, #jquery, #azure, #scrum, #testing, and #opensource). Compared with Bougie et al's sampling method that extracts all microblogs generated by a special group of developers, these 300 microbloggs are more relevant to software development. We manually analyzed the contents of the 300 microblogs and categorized them into ten categories: commercials, news, tools&code, q&a, events, personal, opinions, tips, jobs, and miscellaneous. We found that jobs, news, and q&a are the top 3 most popular categories. We also calculated the percentages of microblogs that are retweeted for each category and found that the most diffused microblogs are from events and commercials categories. Some examples are: "... vote for Superdesk in Ashoka Changemakers Global Innovation Contest. ..." (events), "... GlobalStorage for #dotnetnuke 6 #azure, ... is 15% OFF ..." (commercials). Personal microblogs also get retweeted. The least diffused categories, aside from miscellaneous, are: tools&code, jobs, and q&a. Although these tweets are many in number, they are not widely diffused in the Twitter network.

6.3.3 Software Forges

Madey et al. analyzed open source projects that are hosted in SourceForge [552]. They analyzed 39,000 projects which are developed by more than 33,000 developers. They created a collaboration *social network* where developers are nodes and collaborations among developers (i.e., two or more developers work on the same project) are edges. A modified spanning tree algorithm was used to extract clusters (i.e., groups) of connected developers. Based on this collaboration social network they found that *power-law relationships* exist for project sizes (i.e., number of developers in a project), project memberships (i.e., number of projects that a developer joins), and cluster sizes (i.e., number of developers in a cluster).

Xu et al. investigated *social network properties* of projects and developers in SourceForge [932]. They found that the networks exhibit small world phenomena

and are scale free. Small world phenomenon refers to a situation where each node in a network is connected to other nodes in the network by a small number of intermediary nodes. Scale free network refers to a situation where degree distribution of nodes follows a *power-law distribution*. For scale free networks, preferential attachment (i.e., probability of a new node to link to an existing node is proportional to the degree of the existing node) exists.

Ricca and Marchetto investigated 37 randomly selected projects in Google Code and SourceForge [716]. They investigated "heroes" in these projects; heroes refer to important developers that have critical knowledge on particular parts of a software system. They found that heroes are a common phenomenon in open source projects. They also reported that heroes are faster than non-heroes in completing change requests.

Dabbish et al. investigated a different software forge namely GitHub [210]. Different from SourceForge, GitHub is more transparent, i.e., other developers can track and follow the activities of other developers or changes made to a project. They interviewed a set of GitHub users to investigate the value of transparency. They found that transparency is beneficial for various reasons including: developer recruitment, identification of user needs, management of incoming code contributions, and identification of new technical knowledge.

6.4 Supporting Information Search

In this section, we describe several studies that leverage Web 2.0 to support information search. We first describe two of our previous studies that consider two information search scenarios, namely searching for answers in software forums [343], and searching for similar applications in software forges [841]. We then highlight other studies.

6.4.1 Searching for Answers in Software Forums

Motivation. A thread in a software forum can contain a large number of posts. Our empirical study on 10 software forums found that a thread can contain up to 10,000 posts [343]. Scanning for relevant posts in these threads can be a painstaking process. Likely many posts are irrelevant to a user query. Some posts answer irrelevant questions. Some other posts are relevant but do not provide an answer to the problem that a developer has in mind. Furthermore, even after an exhaustive investigation, there might be no post that answers relevant questions or a correct answer might not have been provided in the forum.

To aid in searching for relevant answers, developers typically make use of general purpose *search engines* (e.g., Google, Bing, etc.) or customized search engines available in software forums. General purpose search engines return many web-

pages. Often many of them are not relevant to answer the questions that developers have in mind, e.g., searching for Java might return the island Java in Indonesia or the Java programming language. Customized search engines are likely to return more relevant results however the number of returned results can still be too many. For example, consider searching the Oracle forum with the following question: "How to get values from an arraylist?". Figure 6.5 shows the returned results. There are 286 threads returned and some threads contain as many as 30 posts. Developers would then need to manually investigate and filter returned results to finally recover posts that answer the question. This could be very time consuming. Thus, we need a more advanced solution to help find relevant posts from software forums.

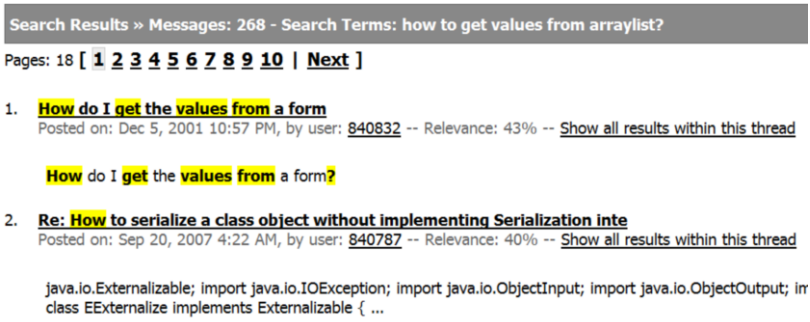

Fig. 6.5: Search results (268 of them) from Oracle forum for query: " How to get values from arraylist?"

Approach. Our proposed approach first labels posts in software forums with predefined *tags*; it then uses these tags to return relevant answers from threads in software forums. It utilizes two main components: tag inference engine and *semantic search engine*. Our tag inference engine automatically classifies posts in software forums with one of the following categories: answers, clarifying questions, clarifying answers[20], positive feedback, negative feedback, and junk (e.g., "today is Friday"). With the inferred tags, developers could focus on the answers that can be hidden deep inside long threads rather than investigating all the posts. With the inferred tags, questions with correct answers (identified based on the corresponding positive feedback) can also be identified. Our semantic search engine enhances standard search engine by making use of the inferred semantic tags to return more relevant answers.

To build a tag inference engine that classifies posts into the seven categories, we follow these steps:

1. We represent each post as a feature vector. To do this, we extract the text in the post and record the author of the post. The textual content of the post is then subjected to the following pre-processing steps: *stopword removal (i.e., removal of*

[20] Answers to clarifying questions.

non-descriptive words) and *stemming (i.e., reduction of a word to its root form)*. For example, the words "reads" and "reading" are reduced to "read". The resultant words are then weighted based on their *term frequency (i.e., the number of times the words appear in the post)*. These together with the author information are used as features (a.k.a. a feature vector) that represent a post.

2. Given a set of posts and their labels, we train a *machine learning* model that discriminates posts belonging to each of the 7 categories using Hidden Markov Support Vector Machine (SVM^{HM}) [443]. We take the representative feature vectors that characterize the training set of posts to train this machine learning model. SVM^{HM} classifies a post not only based on its content and author but also the previous few posts. This is particularly effective as the category of a post is often dependent on the category of the previous few posts, e.g., if the previous post is a question, the next post is likely to be an answer or a clarifying question rather than a feedback.

The learned categories could be used to help conventional search engines. A conventional search engine takes in a set of documents (i.e., forum posts in our settings), pre-processes each document into a bag of words, and indexes each document. When a user enters a query, the index is used for fast retrieval of relevant documents in the document corpus. We enrich conventional search engines by leveraging the semantic information available from the inferred tags. To create this semantic search engine, we embed our tag inference engine to infer tags of the posts in the document corpus. These tags are then used to filter irrelevant posts, e.g., junk. Only documents that are potentially relevant would be returned.

Experiments. Three different forums are analyzed in our experiments: SoftwareTipsandTricks[21], DZone[22], and Oracle[23]. We infer the labels of 6068 posts from the forums manually - 4020, 680, and 1368 posts are from SoftwareTipsandTricks, DZone, and Oracle, respectively. Approximately half of the posts are used for training (i.e., 2000, 300, 620 posts from SoftwareTipsandTricks, DZone, and Oracle, respectively) and the remaining posts are used for testing. We build a search engine corpus using the same sets of posts and consider a set of 17 software queries.[24] We compare our semantic search engine with a standard information retrieval toolkit [25]. We consolidate results returned by the standard information retrieval toolkit and our semantic search engine. The consolidated results are then given to five human evaluators who would give a rating of 2, for correct answers, 1, for partially correct answers, and 0, for irrelevant answers.

We first evaluate the accuracy of our tag inference engine in terms of *precision*, *recall*, and *F-measure* (i.e., the harmonic mean of precision and recall) [557]. We use the manually inferred tags as the ground truth. The results are tabulated in Table 6.1. We can achieve an F-measure of 64-72%. Next, we evaluate the usefulness of our

[21] www.softwaretipsandtricks.com

[22] forums.dzone.com

[23] forums.sun.com/index.jspa

[24] E.g., "How to read files in Java?", please refer to [343] for detail.

[25] www.lemurproject.org

semantic search engine that leverages inferred tags. We compare our approach with a conventional search engine in terms of *mean average precision (MAP)* over a set of 17 queries [557]. The mean average precision a the set of queries is the mean of the average precision per query; the average precision of a query is computed by averaging the precision at the top-k positions, for different values of k. With our semantic search engine we can accomplish an MAP score of 71%, while the conventional search engine can only achieve an MAP score of 18%.

Table 6.1: Precision, Recall, and F-measure Results of Our Proposed Tag Inference Engine

Dataset	Precision	Recall	F-measure
SoftwareTipsandTricks	73%	71%	72%
DZone	68%	61%	64%
Oracle	71%	67%	69%

6.4.2 Searching for Similar Applications in Software Forges

Motivation. Web search engines allow users to search for similar webpages (or documents in the Web). Similarly, developers might want to find similar applications (i.e., applications that serve similar purposes). Finding similar applications could help various software engineering tasks including rapid prototyping, program understanding, plagiarism identification, etc. There have been a number of approaches that could retrieve applications that are similar to a target application [455, 581]. McMillan et al. proposed JavaClan [581] which has been shown to outperform MUDABlue [455]. JavaClan leverages similarities of API calls to identify similar applications. API calls within applications are treated as *semantic anchors* which are used to identify similar applications to a target application. However, the accuracy of these approaches can still be improved. In their user study, JavaClan only achieved a mean confidence score of around 2.5 out of 4.

Recently, many developers tag various resources with labels. This phenomenon is referred to as *collaborative tagging*. Many software forges allow users to tag various applications based on their perceived functionalities. In this study, we leverage collaborative tagging to find similar applications. Our goal is to improve the accuracy of the state-of-the-art approach.

Approach. Our approach, shown in Figure 6.6, consists of several steps including: data gathering, importance weighting, and similar application retrieval. We describe these steps as follows:

1. Data Gathering. We download a large number of applications as the base corpus to detect similar applications. In this study we invoke the API that comes with

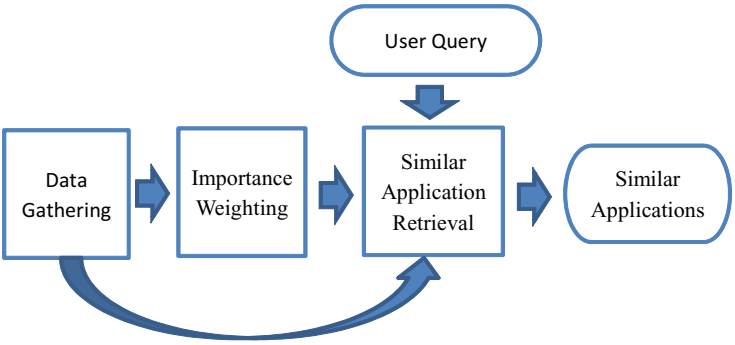

Fig. 6.6: Similar Application Retrieval: Block Diagram

Categories	**License**
Communications, Graphics, Internet, Office Suites, Security, System	Apache License V2.0, BSD License, GNU General Public License version 2.0 (GPLv2), GNU Library or Lesser General Public License version 2.0 (LGPLv2), MIT License, Mozilla Public License 1.1 (MPL 1.1)
Languages	**Intended Audience**
Czech, Dutch, English, French, Hungarian, Italian, Norwegian, Polish, Portuguese, Russian, Slovene, Spanish, Thai	Developers, Education, End Users/Desktop, Science/Research
Programming Languages	
AutoIt, Delphi/Kylix, Java	

Fig. 6.7: Example Tags from SourceForge

SourceForge[26] to collect tags from a large number of applications hosted there. An example of tags given to an application in SourceForge is shown in Figure 6.7. In this study, we treat each tag as a distinct entity and we ignore the semantic relationships between tags.

2. Importance Weighting. Not all tags are equally important. Some tags are very general and are used to label a large number of applications. These tags are not very useful for the retrieval of similar applications as otherwise all applications

[26] sourceforge.net/apps/trac/sourceforge/wiki/API

would be considered similar. On the other hand, tags that are only used by a few applications are more important for retrieval as they can help to differentiate one application from the others.

Based on the above rationale, we assign importance weights to tags based on applications tagged by them. If a tag is used by many different applications, it is assigned a low weight. On the other hand, if a tag is used by only a few applications, it is assigned a high weight. We use the concept of *inverse document frequency* first proposed in the information retrieval community [557] to assign weights to tags. Formally, the weight of a tag T is given by Equation 6.1 where $Applications(T)$ refers to the size of the application set tagged by T.

$$weight(T) = \frac{1}{Applications(T)} \tag{6.1}$$

3. Similar Application Retrieval. Each application is represented as a vector of its tags' weights. The similarity between two applications can then be measured based on the similarities of their representative vectors. Various *similarity measures* can be used. We use *cosine similarity* which is a standard similarity metrics in information retrieval [557]. The cosine similarity of two applications A and B is given by Equation 6.2 where $A.Tags$ and $B.Tags$ refer to the tags for application A and B respectively. From the numerator of the above formula, the cosine similarity of A and B is higher if they share many common tags that have high importance weights. The denominator of the formula normalizes cosine similarity to the range of zero to one. If an application is tagged with many tags, the chance for it to coincidentally share tags with other applications is higher. To address this, the denominator considers the number and weights of the tags that are given to each application.

$$CosSim(A,B) = \frac{\Sigma_{T \in (A.Tags \cap B.Tags)}.weight(T)^2}{\sqrt{\Sigma_{T \in A.Tags}.weight(T)^2} \times \sqrt{\Sigma_{T \in B.Tags}.weight(T)^2}} \tag{6.2}$$

Given a target application A, our system returns the top-n applications in our corpus that are most similar to A based on their cosine similarities.

Experiments. To investigate the effectiveness of our approach, in our data gathering step we collect 164,535 applications (i.e., projects) from SourceForge. These applications form our corpus to recommend similar applications. We use the following 20 queries: bcel, bigzip, certforge, chiselgroup, classgen, color-studio, confab, drawswf, genesys-mw, javum, jazilla, jsresources, opensymphony, psychopath, qform, redpos, sqlshell, tyrex, xflows, and yapoolman. Each of the above queries is an application. The 20 queries were also used in evaluating JavaClan which is the state-of-the-art approach to recommend similar applications [581].

We compare our approach with JavaClan. We use our approach and JavaClan to recommend 10 applications. We then perform a user study to evaluate the quality of the recommendations. We ask users to rate each recommendation using a 5-point

Likert scale [19]: 1. strongly disagree (i.e., the query and recommended applications are very dissimilar), 2. disagree, 3. neither agree or disagree, 4. agree, and 5. strongly agree (i.e., the query and recommended applications are very similar). Based on user ratings, we use the following three metrics to measure the effectiveness of our approach and JavaClan (the last two metrics have been used to evaluate JavaClan):

1. Success Rate. We deem a top-10 recommendation to be successful if at least one of the recommendations is given a rating 3 or above. The success rate is given by the proportion of top-10 recommendations that are successful for the queries.
2. Confidence. The confidence of a participant to a recommendation is reflected by his/her rating. We measure the *average confidence* which is the average of the ratings given by the participants for the top-10 recommendations.
3. Precision. The precision of a top-10 recommendation is the proportion of recommendations in the top-10 recommendation that are given ratings 4 or 5. We measure the average precision across the top-10 recommendations for the queries.

Table 6.2 shows the success rate, average confidence, and average precision of our proposed approach and JavaClan. In terms of success rate, our approach outperforms JavaClan: the success rate is increased by 23.08%. Our approach also achieves a higher average confidence than JavaClan. A *Mann-Whitney U test*, which is a non-parametric test to check the significance of a difference in means, shows that the difference in average confidence is significant (with a p-value of 0.001). Furthermore, out of the 20 queries, in terms of average confidence per query, our approach outperforms JavaClan in 13 queries and is equally as effective as JavaClan in 5 queries. Furthermore, our approach achieves a higher average precision score than JavaClan. We have also performed a A Mann-Whitney U test. The result shows that the difference in mean is not significant (with a p-value of 0.488). Furthermore, out of the 20 queries, in terms of precision per query, our approach outperforms JavaClan in 7 queries and is equally effective as JavaClan in 8 queries.

Table 6.2: Effectiveness of Our Proposed Approach and JavaClan: Success Rate, Confidence, and Precision

Approach	**Success Rate**	**Avg. Confidence**	**Avg. Precision**
Proposed Approach	80%	2.02	0.115
JavaClan	65%	1.715	0.095

6.4.3 Other studies

Aside from our studies, there are a number of other studies that also leverage Web 2.0 resources to help various software evolution activities. We highlight some of these studies in brief in the following paragraphs.

Thummalapenta and Xie proposed Parseweb which helps developers to reuse open source code [839]. Parseweb accepts as input a source object type and a destination object type. It then generates a sequence of method invocations that can convert the source object type to the destination object type. To realize its function, Parseweb interacts with a software forge namely Google Code and leverages the search utility available in it.

Thummalapenta and Xie proposed a technique named SpotWeb that detects hotspots and coldspots in a framework or *API* [838]. Hotspots refer to parts of the framework or API that are frequently used. Coldspots refer to parts of the framework or API that are rarely used. Their proposed technique works on top of Google Code search. It works by analyzing framework code and the framework usages among the projects in Google Code software forge. Experiments were conducted on a number of frameworks with promising results.

McMillan et al. proposed Portfolio which is a *search engine* that finds functions in a large code base containing numerous projects [582]. Their proposed search engine takes in user inputs in the form of free form natural language descriptions and returns relevant functions in the code base. Two information retrieval solutions are used to realize the proposed approach namely page rank and spreading activation network. Their search engine analyzes a software forge containing hundreds of projects from FreeBSD.

McMillan et al. proposed a tool named Exemplar (EXEcutable exaMPLes ARchive) which takes high-level concepts or descriptions and returns applications that realize these concepts [579]. The proposed approach ranked applications in a large application pool by considering several sources of information. These include textual description of the application, list of *API methods* that the application calls, and dataflow relations among the API method calls. Examplar had been evaluated on a set of 8,000 projects containing more than 400,000 files that are hosted in SourceForge with promising results.

Ponzanelli et al. proposed a technique that leverages crowd knowledge to recommend code snippets to developers [690]. Their tool named Seahawk is integrated to the *Eclipse IDE* and recommends code snippets based on the context that a developer is working on. To achieve this, Seahawk generates a query from the current context in the IDE, mines information from *StackOverflow* question-and-answer site, and recommends a list of code snippets to developers.

6.5 Supporting Information Discovery

In this section, we describe studies that develop tools that facilitate developers in discovering new information from Web 2.0 resources. We first highlight our visual analytics tool that supports developers in navigating through the mass of software-related microblogs in Twitter [3]. We also present our analytics tool that can automatically categorize microblogs to support information discovery [703]. We then highlight other studies.

6.5.1 Visual Analytics Tool for Software Microblogs

Motivation. Although microblogging is becoming a popular means to disseminate information, it is challenging to manually discover interesting software related information from microblogs. The first challenge comes from the sheer size of microblogs that are produced daily. Storing *all* microblogs is not an option. Second, many microbloggers do not microblog about software related topics. Indeed only a minority of microbloggers are software developers. Thus there is a need to filter many unrelated microblogs to recover those that are relevant to software development. Third, the large number of microblogs might make it hard for developers to "see" trends in the data. Motivated by these challenges, there is a need to develop an approach that can harvest and aggregate thousands or even millions of microblogs. It should also allow developers to perform *visual analytics* such that various kinds of trends and nuggets of knowledge can be *discovered* from the mass of microblogs. In this study, we propose such an approach.

Approach. We propose a visual analytics platform that filters and aggregates software related microblogs from Twitter. Our proposed platform identifies *topical* and *longitudinal* trends. Topical trends capture relative popularity of similar topics, e.g., relative popularity of various libraries. Longitudinal trends capture the popularity of a topic at various time points, e.g., the number of times people microblog about PHP at various time points. These trends can provide insight to developers, e.g., developers can discover popular programming languages to learn, or discover interesting events (e.g., notification of important security holes, etc.) in the past 24 hours.

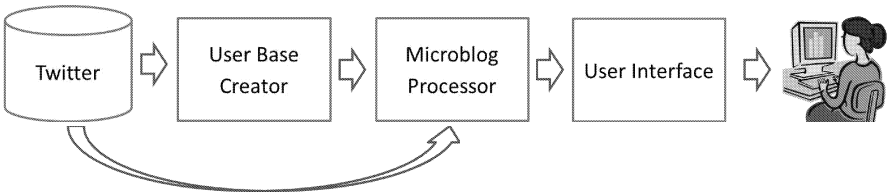

Fig. 6.8: **Proposed Approach: Visual Analytics Platform**

Our platform, illustrated in Figure 6.8, has 3 blocks: *User Base Creator*, *Microblog Processor*, and *User Interface*. *User Base Creator* recovers microbloggers that are likely to microblog about software related topics. *Microblog Processor* downloads and pre-processes microblogs from Twitter. It also identifies topical and longitudinal trends from the microblogs. *User Interface* presents the trends to end users as a web interface which allows users to analyze the trends and the underlying microblogs.

1. *User Base Creator* first processes a set of seed users which are well-known microbloggers that often microblog about software topics. We take the list of seed users available in [43]. In Twitter, a user can *follow* other users and receive up-

dates on microblogs made by the other users. Using these follow links, we expand the seed users to include microbloggers that follow at least n seed users (by default we set the value n to 5). We consider this user base as those that are likely to microblog about software related topics.

2. *Microblog Processor* uses Twitter REST API to continually download microblogs. We then perform standard text pre-processing including tokenization, stopword removal, and stemming. Some technical jargons, e.g., C#, C++, etc. are manually identified. These jargons are not stemmed. Since the identification of jargons is done manually, we focus on jargons corresponding to topics whose trends we would like to visualize. There are jargons that we do not identify and they are treated as regular words and are stemmed. We then index the resultant set of microblogs using *Apache Solr*.[27]

   Next, we perform trend analysis and compute both topical and longitudinal trends. To compute topical trend, we manually select a set of 100 software-related topics, e.g., JavaScript, Scrum, etc., from relevant Wikipedia pages and popular StackOverflow's tags. We then compute for each topic the number of microblogs mentioning the topic at a specific time period. Topics that are more frequently mentioned are more popular than others. To compute the longitudinal trend of a particular topic or keyword, we compute the number of tweets containing it at various points in time. We thus could compute the popularity of various topics and the popularity of a topic at various time points.
3. *User Interface* presents the resultant topical and longitudinal trends. To present topical trends, we display various topics using fonts of various sizes. The size of the font depends on the popularity (i.e., frequency) of the corresponding topic in the microblogs. To present longitudinal trends, for each topic, we plot a line graph that shows the popularity of the topic at various time points.

Experiments. Our dataset consists of approximately 58,000 microbloggers, 76 million microblogs, and 18 million follow links.

With topical trend analysis, popular topics of interest can be highlighted to users. Figure 6.9 shows our topical trend user interface. It shows the relative popularity of various topics. From the interface, users can find out that *JavaScript*, *Ruby*, and *Java* are the most popular programming language topics in the microblogs that we collected in a 24-hour period ending on the 25th of November 2011. For framework, libraries, and systems, *Apple*, *COM*, and *JQuery* are the most popular topics.

With longitudinal trend analysis, the popularity of a topic across different time points can be captured and shown to users. Figure 6.10 shows our longitudinal trend user interface for "JavaScript". We can notice that the number of microblogs related to JavaScript varies over time. We also notice a number of peaks. The highest peak is for the 10th of October 2011. At this date, Google released a new programming language called *Dart* [829]. Programs written in *Dart* can be compiled into JavaScript. We also notice that the number of microblogs related to JavaScript changes periodically - people tend to microblog more about JavaScript on some days than other

[27] lucene.apache.org/solr

days. Figure 6.11 shows another longitudinal trend for "Scrum". We note that, similar to the popularity of JavaScript, the popularity of Scrum is also periodic. We do not notice much anomaly in the Scrum longitudinal trend though. In the future, it is interesting to develop approaches that can automatically highlight anomalies and recover important events.

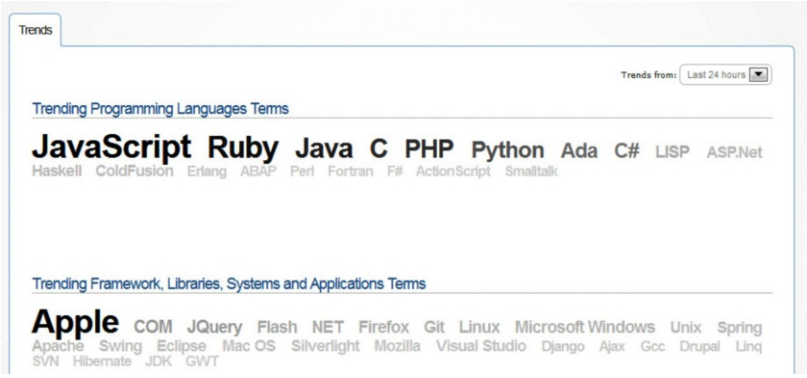

Fig. 6.9: **Topical Trend User Interface**

Fig. 6.10: **Longitudinal Trend User Interface for "JavaScript"**

6.5.2 Categorizing Software Microblogs

To leverage microblogging in software evolution tasks, we need to first understand how microblogging is currently used in software related contexts. One way to do this is to categorize software related microblogs. We present our *machine learning approach* that automatically assigns category labels to microblogs.
Motivation. By subscribing to and reading microblogs written by other developers, a developer can discover much information, e.g., a new programming trick, a new

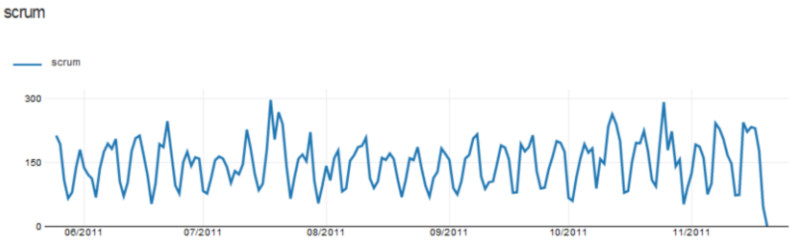

Fig. 6.11: **Longitudinal Trend User Interface for "Scrum"**

API, etc. Unfortunately, most of the microblogs are not informative [629]. Even if they are informative, they might not be relevant to *engineering* software systems. Our manual investigation on a few hundreds microblogs tagged with software related hashtags (see Section 6.3.2) shows that most of the microblogs belong to the category: jobs. They are job advertisements and are not relevant to engineering software systems. Thus, many interesting microblogs *relevant* to engineering software systems are buried in the mass of other *irrelevant* microblogs. In this work, we build a machine learning solution that can automatically differentiate *relevant* and *irrelevant* microblogs.

Approach. The framework of our proposed approach is shown in Figure 6.12. It works in two phases: training and deployment. In the training phase, the goal is to build a *machine learning model* (i.e., a discriminative model) that can discriminate relevant and irrelevant microblogs. In the deployment phase, this model is used to classify an unknown microblog as relevant or irrelevant. Our framework consists of 3 main blocks: webpage crawler, text processor, and classifier. A microblog can contain a URL; for such microblogs the webpage crawler block downloads the content of the webpage pointed by the URL. Our text processor block converts textual content in the microblogs and downloaded webpages' titles into word tokens after standard information retrieval pre-processing steps. These word tokens become features for our classifier which constructs a discriminative model. The model is then used to predict if a microblog is relevant or not.

We elaborate the webpage crawler, text processor, and classifier blocks as follows:

1. Webpage Crawler. A microblog in Twitter contains a maximum of 140 characters. To express longer contents, microbloggers often include a URL to a webpage, containing expanded content, in the microblog. Services like *bit.ly* are often used to shorten the URL. Information contained in the webpages pointed by these URLs can help to classify the relevance of the microblog. Thus, we want to download these external webpages. Our webpage crawler block performs this step by first checking if a microblog contains a URL. It uses a regular expression to detect this. It then expands any shortened URL into the original URL by checking the HTTP header. Finally, it downloads the webpages. It then extracts the titles of these webpages as they provide succinct yet very informative con-

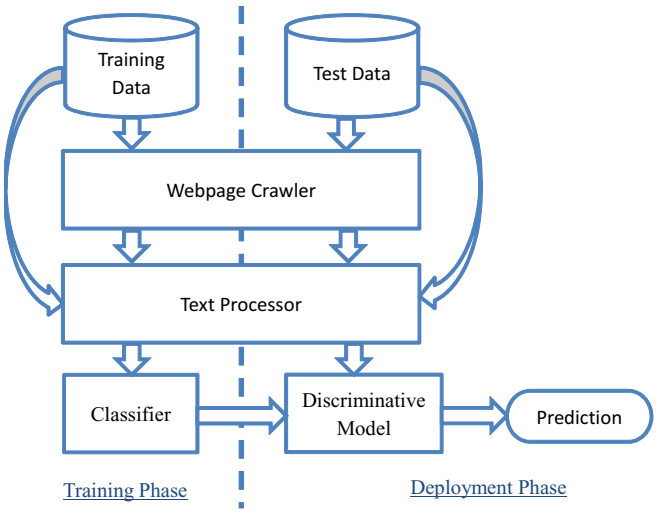

Fig. 6.12: Proposed Approach: Microblog Relevancy Categorization

tents. The body of a webpage is often long and contain extraneous information (e.g., advertisements, navigation links, etc.).

2. Text Processor. This block processes the text contents of the microblogs and webpage titles. It first removes stop words based on *Natural Language Toolkit (NLTK)'s* stopword list.[28] Next, it reduces each word to its root form (i.e., stemming) by using Porter stemmer [692]. Finally, each pre-processed word is treated as a feature and we combine these words to form a feature set that characterize a given microblog.
3. Classifier. This block takes in the feature sets, produced by the text processor block, of a set of microblogs whose relevancy label is known (i.e., relevant or irrelevant). It then constructs a discriminative model that differentiates relevant from irrelevant microblogs. We make use of *support vector machine (SVM)* [612] to construct the discriminative model. SVM has been widely used in past studies on software mining [489, 802]. SVM views a microblog as a point in a multi-dimensional space where each feature is a dimension. It then creates a hyperplane that best separates feature sets of the relevant microblogs with those of the irrelevant microblogs. This hyperplane is the discriminative model which is used in the deployment phase to assign relevancy labels to other microblogs.

Experiments. We use a dataset consisting of 300 microblogs which are tagged with either one of the following 9 *hashtags*: #csharp, #java, #javascript, #.net, #jquery, #azure, #scrum, #testing, and #opensource. Although the dataset does not cover all kinds of microblogs and hashtags, it is a good starting point to test the effectiveness of our proposed approach. These microblogs have been grouped into 10 categories

[28] nltk.org

listed in Table 6.3 (see [845]). Here, to create the ground truth data to evaluate the effectiveness of our approach, we manually re-categorize these 300 microblogs into 2 classes: relevant and irrelevant. The distribution of relevant and irrelevant microblogs across the 10 categories is shown in Table 6.4.

Table 6.3: Microblog Categories

	Category	Details
1.	Commercials	Advertisements about a commercial product or a company
2.	News	Objective reports
3.	Tools & Code	Sharing of code and/or links to open source tools
4.	Q&A	Questions or links to questions in Q&A sites
5.	Events	Notification of particular events or gatherings
6.	Personal	Personal messages, e.g., ramblings about programming, etc.
7.	Opinions	Subjective expressions of likes or dislikes
8.	Tips	Advice about a particular problem, e.g., how to do a particular programming task, etc.
9.	Jobs	Job advertisements
10.	Misc.	Other kinds of microblogs. This includes microblogs whose contents are unclear.

Table 6.4: Relevance Per Microblog Category

Category	Proportion of Relevant Microblogs
Tools & Code	100%
Tips	100%
Q&A	86.4%
Events	45.5%
Opinions	42.9%
Commercials	40%
News	29.5%
Personal	0%
Jobs	0%
Misc.	0%

Using the above data, we perform a 10-fold cross validation, and measure the precision, recall and F-measure of our proposed approach. In 10-fold cross validation, 90% of the data is used for training and only 10% is used for testing. We would like to investigate the sensitivity of our approach on the amount of training data.The experiment shows that we can predict the relevancy of a microblog with 74.67% accuracy, 76% precision, 67.38% recall, and 71.43% F-Measure.

Next, we investigate the effectiveness of our approach for each of the ten microblog categories. The result is shown in Table 6.5. It shows that we can more accurately predict relevancy labels of jobs, personal, Q & A, tools & code, opinions, and misc categories. Our approach needs to be further improved for tips category

(low precision), and events category (low precision and recall). For the events category, the microblogs are more ambiguous and it is harder to predict if a microblog is relevant or not. In the future, we plan to use other approaches including sentiment analysis [677] to improve the accuracy of our proposed approach.

Table 6.5: Effectiveness Per Microblog Category

Category	Accuracy	Precision	Recall	F-Measure
Jobs	100%	0%	0%	0%
Personal	93.8%	0%	0%	0%
Q&A	79.6%	84.2%	91.4%	87.7%
Tools & Code	79.5%	79.5%	100%	88.6%
Opinions	76.2%	55.6%	83.3%	66.7%
Misc.	72%	0%	0%	0%
Tips	48.5%	48.5%	100%	65.3%
Commercials	60%	50%	50%	50%
News	54.5%	61.5%	34.8%	44.4%
Events	45.5%	20%	33.3%	25%

For the above results, we make use of 10-fold cross validation. Then we would like to investigate the sensitivity of our approach on the amount of training data. For this, we perform *k-fold cross validation*, where k is less than 10. We vary k from 2 to 9 and show the resulting accuracy, precision, recall, and F-measure for these values of k in Table 6.6. We notice that the F-measure scores do not vary much, this shows that our framework is effective enough on different amount of training data.

Table 6.6: Results using Different Amount of Training Data

k	Accuracy	Precision	Recall	F-Measure
9	75.43%	75.19%	70.92%	72.99%
8	74.29%	74.62%	68.79%	71.59%
7	73.98%	74.05%	68.79%	71.32%
6	74.33%	73.88%	70.21%	72%
5	73.67%	74.22%	67.38%	70.63%
4	75%	75.78%	68.79%	72.12%
3	74.67%	74.44%	70.21%	72.26%
2	75%	75.78%	68.79%	72.11%

6.5.3 Other studies

There are a number of other studies that leverage Web 2.0 resources for information discovery. We highlight a few of them in brief in the following paragraphs.

Hens et al. extracted frequently asked questions (FAQs) from mailing lists and software forums [389]. They employed a text mining approach that utilizes text pre-processing techniques and *Latent Dirichlet Allocation (LDA)* which is a topic modeling technique. After a topic model was learned from the mailing lists and software forums, several processing phases were employed to identify question and answer pairs that are associated with a topic, discard topics that are unfocused, and process the remaining question and answer pairs to improve their readability. They had investigated their proposed approach on mailing lists of 50 popular projects listed in ohloh.net.

Lungu et al. proposed a visualization tool named Small Project Observatory that analyzes projects in a software forge [548]. With their visualization tool, developers can investigate the evolution of project size (in terms of the number of classes), the level of activity (in terms of the number of commits) occurring within a repository over time, the dependencies among projects, the collaborations among developers, and many more. They have deployed their visualization tool on a software forge owned by Soops b.v, which is a Dutch software company, with promising results.

Sarma et al. proposed Tesseract which is a visualization tool that enables one to explore socio-technical relationships in a software project [746]. Tesseract simultaneously shows various pieces of information to users including: developers, their communications, code, and bugs. Tesseract also supports interactive explorations - it allows users to change various settings, filter information, highlight information, and link information in various ways. Tesseract had been evaluated on the *GNOME project* via a user study and the result is promising.

6.6 Supporting Project Management

In this section, we highlight how Web 2.0 resources could be leveraged to aid project management activities. Project management activities (e.g., planning, organizing, and managing resources) need to be performed repeatedly as software evolves over time. We first highlight studies that leverage software forges for the recommendation of developers to a project [805] and prediction of project success [806]. We also describe other related studies.

6.6.1 Recommendation of Developers

Motivation. It is a challenge to find compatible developers as not everyone works equally well with everyone else. Often there are hundreds or even thousands of developers. It is hard for a manager to know everyone well enough to make good recommendations. Past studies only recommend developers from a single project to fix a particular bug report [819]. Thus there is a need for a tool that can help recommend

developers based on their past socio-technical behaviors and skills. In this work we focus on recommending developers from a software forges (i.e., SourceForge).

Approach. Our approach consists of 2 main steps: *Developer-Project-Property (DPP) graph construction*, and *compatibility scores computation*. In the first step, we represent the past history of developer interactions as a special Developer-Project-Property (*DPP*) graph. In the second step, given a developer we compute compatibility scores of the developer with other developers in the *DPP* graph. We propose a new compatibility metric based on *random walk with restart (RWR)*. We elaborate the above two steps in the following paragraphs.

Given a set of developers, their past projects, and the project properties, we construct a *DPP* graph. There are three node types in a *DPP* graph: developers, projects, and project properties. We consider two project properties: project categories and project programming languages. There are two types of edges in a *DPP* graph: one type links developers and projects that the developers have participated in before, another links projects and their properties. A developer can work on multiple projects. A project can have multiple properties: it can be associated with multiple categories and/or multiple programming languages. For forges where only one programming language is supported, other properties aside from programming language can be considered, e.g., tags [855], libraries used, etc.. Figure 6.13 gives an example of a *DPP* graph which is a tripartite graph.

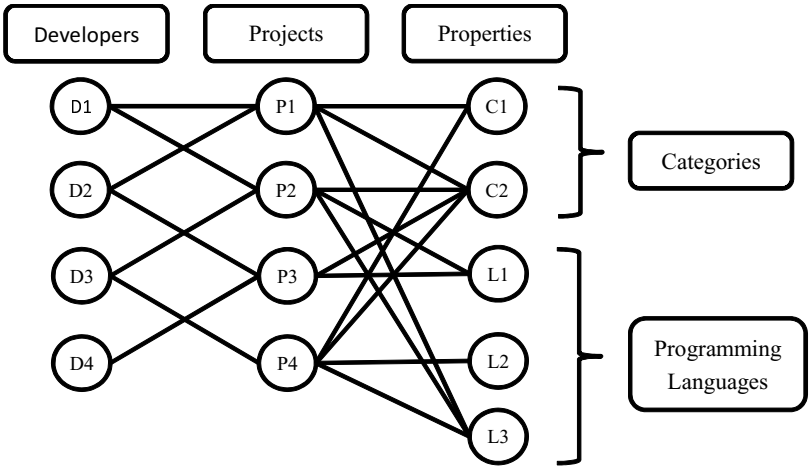

Fig. 6.13: **Example Developer-Project-Property (*DPP*) Graph**

After a *DPP* graph is constructed, we can compute compatibility scores between each pair of developers. A more compatible pair of developers should be assigned a higher score than a less compatible pair. Intuitively, a good compatibility metric should satisfy the following:

1. A pair of developers that have worked together in many joint projects are more likely to be compatible than another pair that have not worked together before.
2. A project is characterized by its properties: categories and programming languages. Intuitively, developers that have worked on similar projects (i.e., different projects of the same/similar properties) are more likely to be more compatible than those that have worked on completely unrelated projects.
3. Developers might not have worked together before. However, they might have a common collaborator. Developers with many common collaborators developing similar projects are more likely to be more compatible than "complete strangers". The same is true for collaborators of collaborators, albeit with lower impact on compatibility.

The above describes three qualitative criteria for a good compatibility metric. We find that computing node similarity using *random walk with restart (RWR)*, which was first proposed for web search engines in 1998 [676], fits the three criteria. Given a developer node d in the *DPP*, by performing many random walks with restart starting from developer node d, many nodes are visited. Some nodes are visited more often than other nodes. RWR assigns scores to these other nodes based on the probability that these nodes are visited during RWR starting from node d. After RWR, developers with higher scores are more likely to have worked with developer d on many common projects, or they have worked on projects with similar properties, or they share many common collaborators or collaborators of collaborators. Given the target developer d, we sort the other developers based on their RWR scores, and return the top-k most compatible developers.

Experiments. To evaluate the effectiveness of our proposed developer recommendation approach, we analyze projects in SourceForge. We make use of the curated data collected by Van Antwerp et al. [40].[29] We analyze the curated data collected from May 2008 until May 2010. Each month, Antwerp et al. release a snapshot of the curated data in the form of SQL tables. From these snapshots, we extract information about developers, projects that these developers work on, and project categories as well as programming languages. To recommend developers, we need sufficient information of developers' past activities. Thus, we only include developers that have worked on at least p projects. SourceForge contains many trivial projects; to filter these projects, we only include projects that have at least n developers. In this study, we set the value of p and n to be 7 and 3 respectively.

A good recommendation eventually leads to a collaboration. To evaluate our approach, we take multiple consecutive monthly snapshots of SourceForge. We consider new collaborations created between these consecutive snapshots. We then apply our approach and investigate if we can accurately predict these new collaborations. We consider a recommendation is successful if at least one of the recommended developer collaborates in the next snapshot. The accuracy of our approach is defined as the proportion of recommendations that are successful. If many new collaborations do not follow our recommendations then the accuracy would be low. This measure is also often referred to as recall-rate@k and has been used in many

[29] www3.nd.edu/~oss/Data/data.html

past studies [645, 737, 800]. This is a lower bound of the accuracy of our proposed approach. In practice, our approach would *actively* recommend developers and more collaborations could have been created.

Given a target developer d, our approach would recommend k developers with the highest RWR scores. Using k equals to 20, for the new collaborations created from May 2008 to May 2010, we find that our recommendation success rate is 83.33%. We also vary the value k and investigate the behavior of our approach. We find that the success rate (or accuracy) varies from 78.79% to 83.33% when k is varied from 5 to 20. Thus there is only a minimal change in accuracy (i.e., 4.54% reduction) when we drop k from 20 to 5. This shows that our top few recommendations are accurate. The runtime of our approach is 0.03 seconds for training (i.e., creating *DPP* and pre-computing internal data structures) and less than a second for query (i.e., running RWR with the pre-computed internal data structures). This shows that our approach is efficient and could be used to support interactive query.

6.6.2 Prediction of Project Success

Motivation. Project success is the eventual goal of software development efforts, be it open source or industrial. There are many projects that are successful - they get released, distributed, and used by many users. These projects bring much benefit in one form or another to the developers. However, many other projects are unsuccessful, they do not get completed, not used by many (if any at all), and bring little benefit to the developers despite their hard work. Investigating failed and successful projects could shed light on factors that affect project outcome. These factors can in turn be used to build an automated machine learning solution to predict the likelihood of a project to fail or be successful. Predicting project outcome is important for various reasons including planning, mitigation of risks, and management of resources.

With the adoption of Web 2.0, much data is available to be analyzed. We can collect information on various successful and failed projects. We can trace various projects that developers have worked on before. In this work, we leverage socio-technical information to differentiate successful and failed projects. Our goal is to find relevant socio-technical patterns and use them to predict project outcome.

Approach. Figure 6.14 illustrates the framework of our proposed approach. It has two phases: training and deployment. In the training phase, our framework processes a set of projects along with developers that work in them. The goal of the training phase is to build a discriminative model that can differentiate successful and failed projects. This model is then passed to the deployment phase to predict the outcomes of other projects. It has several processing blocks: socio-technical information extraction, discriminative graph mining, discriminative model construction, and outcome prediction. The following elaborates each of these processing blocks:

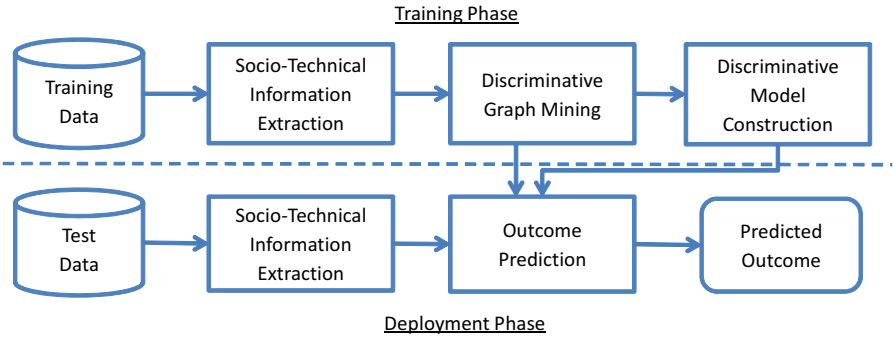

Fig. 6.14: Overall Framework

1. The socio-technical information extraction block processes each of the projects and for each of them extracts socio-technical information in the form of a rich (i.e., multi-labeled) graphs. The nodes in the graph are developers that work on the project. The edges in the graph correspond to the relationships among the various developers. Multiple labels are attached to the nodes and edges to capture the socio-technical information about the developers and their relationships. For each node, we attach the following pieces of information:

   a. Past Successful Projects. This is the number of successful projects that a developer has participated before he joins the current project.
   b. Past Failed Projects. This is the number of failed projects that a developer has participated before he joins the current project.
   c. Length of Membership. This is the period of time that has passed since a developer has joined the software forges before he joins the current project.

   For each edge that links two developers, we attach the following pieces of information:

   a. Past Successful Collaborations. This is the number of successful projects that the two developers have worked together before.
   b. Past Failed Collaborations. This is the number of failed projects that the two developers have worked together before.
   c. Length of Collaboration History. This is the period of time that has passed since the two developers collaborated for the first time to the time they collaborate in the current project.

2. The discriminative graph mining block takes as input the graphs capturing the socio-technical information of the successful and failed projects. The goal of the block is to extract subgraphs that appear often in the socio-technical graphs of the successful projects but rarely in the graphs of the failed projects (or vice versa). These discriminative graphs highlight peculiar socio-technical patterns

that differentiate successful from failed projects. We propose a new discriminative subgraph mining algorithm that analyzes rich graph where each node and edge have multiple labels. We can assign a score $S(g)$ to evaluate the discriminativeness of a subgraph g. The goal of a discriminative graph mining algorithm is then to return top-k subgraphs g that have the highest $S(g)$ scores. Variosus measures of discriminativeness have been proposed in the literature. In this work, we make use of *information gain* [612]. Our algorithm extends the work by Cheng et al. [179] that works on a set of simple graphs (i.e., graphs where nodes and edges have one label each) by a translation-and-reverse-translation based approach:

   a. We translate the set of rich graphs to their equivalent simple graph representations.
   b. We mine discriminative subgraphs from the simple graphs using the algorithm proposed by Cheng et al. [179].
   c. We reverse translate the resultant discriminative simple subgraphs back to their corresponding rich graphs.

3. The discriminative model construction block takes as input the set of discriminative subgraphs. Each of the subgraphs form a binary feature. A socio-technical graph is then represented by a vector of binary features. Each binary feature is assigned a score of 1 if a discriminative subgraph appears in it. The score would be 0 otherwise. Using this representation the successful and failed projects become points in a multi-dimensional space. We use *support vector machine (SVM)* to create a discriminative model which is a hyperplane that best separates the two sets of points.
4. The outcome prediction block takes as input the discriminative model learned in the training phase and vector representations of projects whose outcomes are to be predicted. These vector representations are generated by first extracting socio-technical graphs. Each of these graphs are then compared with the discriminative subgraph patterns extracted during the training phase. Each of the patterns form a binary feature that collectively characterize each of the socio-technical graphs. These features are then input to the discriminative model and a prediction would be outputted.

Experiments. We analyze successful and failed projects in SourceForge. We use the monthly database dumps created by Antwerp et al. [40] from February 2005 to May 2010. Projects that have been downloaded more than 100,000 times are deemed to be successful. On the other hand, those that have been downloaded less than 100 times are considered to have failed. Other definitions of success and failure can also be considered; we only investigate one definition in this work. We extract 224 successful projects and 3,826 failed projects. Using this dataset, we want to investigate if socio-technical information of participating developers (which could be gathered even when a project is at its inception) could be used to predict project success using our proposed approach. In the experiments, we first analyze the efficiency, followed by the effectiveness of our approach.

We find that our translation and reverse translation processes can complete in a short amount of time. The translation process only takes less than 15 seconds to translate the successful and failed projects. After translation the sizes of the graphs grow, however their growth is linear to the number of node labels and edge labels. The average sizes of the translated graphs are 31.54 nodes and 287.25 edges (for successful graphs) and 23.93 nodes and 204.68 edges (for failed graphs). The most expensive operation in our framework is to run the algorithm of Cheng et al. [179] which completes within 4 hours. We mine the top-20 most discriminative rich subgraph patterns.

To measure the effectiveness of our approach we use two measures: accuracy and *area under the ROC curve (AUC)* [362]. The *ROC curve (Receiver Operating Characteristic)* plots the false positive rate (x-axis) against the true positive rate (y-axis) at various settings. AUC is more suitable to be used than accuracy for skewed datasets. For our problem, we have a skewed dataset as there are more failed projects than successful projects. The maximum AUC score is 1.0. Using ten-fold cross validation, our proposed approach can achieve an accuracy of 94.99% and an AUC of 0.86.

6.6.3 Other studies

There are a number of other studies that leverage Web 2.0 resources for software project management to reduce the amount of wasted effort, to better manage resources (i.e., developer time and effort), and to coordinate activities. We highlight a few of them in brief in the following paragraphs.

Guzzi et al. combined microblogging with *IDE interactions* to support developers in their activities [356]. Guzzi et al. noted that developers often need to go through program understanding phase many times. This is a time consuming activity. To address this problem Guzzi et al. proposed a tool named James that integrates microblogging with interaction information that is automatically collected from an IDE. Developers can then share their program understanding experience to their colleagues using James. Thus with James, wasted effort can be reduced and developer resources can be better spent on more useful activities.

Ibrahim et al. investigated factors that encourage developers to contribute to a mailing list discussion [418]. There are numerous threads in a mailing list and developers can easily miss relevant threads to which he/she can contribute ideas and expertise. To address this problem, Ibrahim et al. proposed a personalized tool that recommends threads that a developer is likely to contribute to based on the developer past behaviors. The proposed tool combines two *machine learning algorithms* namely: Naive Bayes and Decision Tree. The proposed tool has been evaluated on mailing lists of three open source projects, Apache, PostgreSQL and Python, with promising results.

Carter and Dewan proposed a tool that is integrated with Google Talk [167]. This tool could highlight remote team members in a distributed development team

who are having difficulty in their tasks, and thus foster more collaborations among developers. Expert developers or project managers could be aware of team members that require help. The proposed tool logs developer interactions with a development environment. A classification algorithm was then employed to infer a developer status based on his/her interaction log. A user study was conducted to evaluate the effectiveness of the proposed approach with promising results.

6.7 Open Problems and Future Work

In the previous sections, we have highlighted a number of studies that analyze how developers use Web 2.0 resources and how automated tools can leverage these resources for information search, information discovery and project management. Albeit the many existing work in this area, we believe much more work can be done to better leverage Web 2.0 resources for software evolution. We highlight some of the open problems and potential future work in this section.

There are many Web 2.0 resources that have not been tapped to improve software evolution. In Section 6.2 we highlighted resources such as directories of developers in LinkedIn, public profiles of developers in Facebook, definitions of software engineering terms in Wikipedia and geolocation coordinates of developers in Foursquare. To the best of our knowledge, there have not been any study that utilize these resources to help software evolution. Furthermore, various web systems evolve (see Chapter 7); thus, many additional functionalities and services are introduced to existing Web 2.0 resources regularly. Many innovative applications can potentially be built leveraging these resources and additional functionalities. For example, one can imagine a tool that enables one to search for a potential employee by leveraging information in LinkedIn and Facebook and correlating the information with the kinds of software evolution tasks that the future employee is supposed to perform. One could also better enhance many information retrieval based tools, e.g., [800, 846, 906, 949], by leveraging domain specific knowledge stored in Wikipedia. One can also imagine an application that tries to recommend more interactions among developers that live in a nearby area by leveraging geolocation coordinates in Foursquare. Thus there is plenty of room for future work.

Combining *many* different sources of information and leveraging them to improve software evolution activities is another interesting direction for future work. Most studies so far only focus on one or two Web 2.0 resources. Each Web 2.0 resources provides an incomplete picture of an entity (e.g., a developer). By combining these Web 2.0 resources, one can get a bigger picture of an entity and use this bigger picture to support various software evolution activities, e.g., recommend a fix/a developer to a bug in a corrective software evolution activities by leveraging information in multiple software forums, question-and-answer sites, software forges, etc.

Another interesting avenue for future work is to improve the effectiveness and efficiency of machine learning solutions that analyze and leverage Web 2.0 resources.

As highlighted in previous sections, the accuracy of existing techniques is not perfect yet. Often the accuracy (measured either in terms of precision, recall, accuracy, or ROC) is lower than 80%. Thus there is much room for improvement. Many new advances in machine learning research can be leveraged to improve the accuracy of these existing techniques. One can also design a new machine learning solution that is aware of the domain specific constraints and characteristics of software engineering data and thus could perform better than off-the-shelf or standard solutions. It is also interesting to investigate how search-based algorithms (described in Chapter 4) and information retrieval techniques (mentioned in Chapter 5) can be utilized to improve the accuracy of existing techniques that leverage Web 2.0 resources.

6.8 Conclusions

Web 2.0 provides rich sources of information that can be leveraged to improve software evolution activities. There are many Web 2.0 resources including software forums, mailing lists, question-and-answer sites, blogs, microblogs, and software forges. A number of empirical studies have investigated how developers contribute information to and use these resources. Automated tools can also be built to leverage these resources for information search, information discovery, and project management which are crucial activities during software evolution. For information search, we have highlighted some examples how Web 2.0 resources can be leveraged: tags in software forums can be used to build a semantic search engine, tags can also be used to recover similar applications, code fragments of interest can be extracted from Web 2.0 sites, etc.. For information discovery, we have also highlighted some examples how Web 2.0 resources can be leveraged: users can find interesting events by navigating through the mass of software microblogs using a visual analytics solution, users can be notified of relevant microblogs using a classification-based solution, frequently asked questions can be extracted from Web 2.0 sites, etc.. For supporting project management activities, Web 2.0 resources can also be leveraged in several ways: appropriate developers can be recommended to a project based on their socio-technical information stored in software forges, potentially unsuccessful projects can be highlighted early using developer socio-technical information stored in software forges, better collaboration among developers can be achieved by integrating microblogging with IDEs, etc.. Much more future work can be done to better leverage Web 2.0 and even Web 3.0 resources in various ways to improve many software evolution activities.