Automated Tagging of Software Projects using Bytecode and Dependencies

Santiago Vargas-Baldrich
Universidad Nacional de Colombia
svargasb@unal.edu.co

Mario Linares-Vásquez
The College of William and Mary
mlinarev@cs.wm.edu

Denys Poshyvanyk
The College of William and Mary
denys@cs.wm.edu

***Abstract*—Several open and closed source repositories group software systems and libraries to allow members of particular organizations or the open source community to take advantage of them. However, to make this possible, it is necessary to have effective ways of searching and browsing the repositories. Software tagging is the process of assigning terms (i.e., tags or labels) to software assets in order to describe features and internal details, making the task of understanding software easier and potentially browsing and searching through a repository more effective. We present Sally, an automatic software tagging approach that is able to produce meaningful tags for Maven-based software projects by analyzing their bytecode and dependency relations without any special requirements from developers. We compared tags generated by Sally to the ones in two widely used online repositories, and the tags generated by a state-of-the-art categorization approach. The results suggest that Sally is able to generate expressive tags without relying on machine learning-based models.**

I. Introduction

Several open- and closed-source software repositories are available for developers to take advantage of preexisting software assets that address particular needs that their projects may have. However, in order to be able to take advantage of a repository, there should be ways for efficiently locating assets in it [1]. Assigning *tags* or *categories* to projects, that describe features of the projects such as application domain, programming language, operating system, etc., is a commonly used approach for improving the browsing, searching and retrieval processes in large repositories. **Categories** represent high-level concepts that help group similar assets together and apply to a wide range of them, (e.g. *testing, framework, bytecode analysis, etc.*) while **tags** represent more specific concepts that often relate directly to implementation details (e.g., *xml, grammar, servlet, java, etc.*).

The assignation of tags and categories could be performed manually, for example, by asking developers to categorize their projects upon uploading them to a repository by selecting from a predefined list of categories or by allowing them to enter free text. However, both cases could lead to misclassification or the developer can just avoid the category selection. The categorization task could be delegated to repository administrators or curators, however, the size and rapid pace at which repositories keep growing can render this manual approach impractical.

Automatic software categorization/tagging has progressively gained importance because of the benefits that can span from its use. Research progress has been made on various approaches and methods that rely on *machine learning* techniques for classification. Most of these approaches utilize identifiers extracted from source code, assuming there is full access to the repository [2]–[4]. This situation is not common in closed-source environments; it is known that many companies often work under high security practices to protect organizational secrets, which restricts the access to source code, thus limiting applications of such a categorization model that relies on sensitive information to work. Also some libraries are published as JAR files without the source code, as in the case of some Maven-based projects. To address this concern, a number of approaches relying on Application Programming Interface (API) calls and identifiers extracted from bytecode have been proposed and evaluated [5]–[7].

In software categorization, the category labels are mostly created manually by domain experts [8] or selected from a set of previously defined categories [5], [6]. These approaches rely mostly on *supervised learning* [9], thus, the approaches require a previously categorized set of projects to be used as training data. Supervised categorization assumes that the set of predefined categories is sufficient to classify any new project that enters the repository although that may not necessarily be the case. Also, a predefined set of categories is limited by the knowledge of available domain experts or by the decisions made by repository administrators.

In some cases, category labels are created automatically by analyzing the information in hosted projects [2], [3], [10]. This automatic generation of category labels has the advantage of relying only on information mined from the source code, thus decoupling category names from specific knowledge of the available domain experts. However, closed source repositories can not be categorized using approaches relying on source code [5], [6], since the source code may not always be available as in the case of projects in Maven repositories.

With those issues in mind, we developed *Sally*, a novel, multi-label and unsupervised approach for automatic tagging of closed-source projects, in particular Maven-based software. By extracting identifiers from bytecode and harnessing the *dependency relations* between projects, *Sally* is able to produce a set of expressive tags and present them in a useful way to users. Additionally, it is is capable of dealing with common problems that previous work on automatic software categorization have faced. In particular, *Sally* has the following contributions over competitive approaches:

1) *Sally* does not need access to source code in order to work. This makes the approach a feasible alterna-

tive for closed-source and organizational repositories where access to information is restricted due to security reasons;
2) Tags are not predefined but rather obtained from identifiers and project dependencies. This way we reduce the possibility of missclasification that could be derived from predefined tags;
3) Besides generating tags based on identifiers and dependency relations, *Sally* is able to produce a measure of how relevant is a tag for a given project;
4) By relying on filtering process based on tags from StackOverflow, *Sally* is able to provide descriptive information about the projects in a repository. Additionally, by mining information from widely used sources, definitions for the presented labels can be obtained;
5) By extracting information from dependency relations among projects, *Sally* is able to produce tags that not only describe projects by themselves, but the context on which they are used.

II. Related Work

Most of the previous work on software categorization has predominantly relied on machine learning algorithms and mainly differ in the way how and which features are extracted and the specific classification algorithm employed.

Source code [2], [3], [10], comments [10], readme files [4], online repository profiles [7], [11] and API calls [5], [6] have been used as input for feature extraction. Besides source code, Ugurel *et al.* [4] include comments and readme files, finding that comments can have a negative effect on the classification for some languages. Kawaguchi *et al.* [2], [3] extract identifiers only from source code arguing that the use of design documents, build scripts or other software artifacts is not convenient because although these artifacts can contain highly abstracted information, their quality can vary greatly from project to project thus affecting the categorization results. Later on, Tian *et al.* present an approach that also takes into account comments in source code and has a more strict filtering scheme for identifiers [10].

In [5], [6], the authors ignore source code and make use of API calls for feature extraction acknowledging the cases in which the availability of source code cannot be counted on. By using their approach, closed-source and organizational software repositories can be subject to automatic software categorization. More recently, bytecode was used in conjunction with online software profiles to extract data from projects and construct and unsupervised categorization model based on a Dirichlet Process clustering algorithm [7]. Approaches that make use of inputs different to source code are relevant since it is known that several companies work under highly restricted environments to protect organizational secrets, a practice that limits access to source code. Software tags and online profiles have been used to perform tasks such as finding similar applications given a query [12] and perform hierarchical categorization of software projects [11]. Al-Kofahi *et al.* [13] explore automatic tagging using fuzzy sets theory and propose TagRec, a tag recommendation tool that also uses evolutionary information extracted from the systems' history.

For classification, most of the approaches use textual categorization, which considers software projects as documents. After extracting identifiers, Ugurel *et al.* [4] use Expected Entropy Loss to select the most important features, Support Vector Machines are used to categorize projects by programming language and application topic. In [2], the authors use Decision Trees for categorization with a reported error rate of less than 5%. To deal with the need of predefined categories required to use Decision Trees (and in general any supervised learning algorithm), the authors propose the use of LSA as a way to obtain similarities between software systems and to use this information for classification. Their work is extended in [3] where MUDABlue is proposed: after obtaining similarities between projects, MUDABlue uses cluster analysis to find sets of related software systems. Tian *et al.* propose in [10] a similar approach that uses LDA as a way to obtain topics and also applies cluster analysis to determine the software clusters. In [6], the authors use Decision Trees, Naive Bayes and Support Vector Machines (SVM) for classification, finding the SVM approach to be the most effective. Wang *et al.* also propose in [14] an approach based on similarity of software systems and clustering using these similarities to build a taxonomy for 40,744 projects from Freecode.

III. Our Approach

Sally is composed of four main steps depicted in Figure 1. First, identifiers are extracted from bytecode and filtered using a scheme that leverages tags from **StackOverflow**. In parallel, dependency relations between projects are analyzed in order to obtain metrics that serve as input in the tagging process. The filtered identifiers and dependency metrics are then used to generate primary and secondary tags for each of the projects. The final step is to produce a tag cloud that gives visual clues about how relevant each tag is for the projects they have been assigned to as well as easy access to the definition of the concept described by the tag.

A. Identifier Extraction and Filtering

Extraction: The *ASM Bytecode Manipulation Framework* is used to obtain class names, class fields, method names and method arguments from bytecode [15]. To prepare the obtained identifiers for further computations, they are splitted by camel case and stemmed using Apache Lucene [16].

Filtering: The first part of the filtering process removes identifiers that appear in more than 50% of the projects, identifiers comprised of less than three characters (to keep important concepts such as `XML`, `POM`, `RSS`) and stop-words. Then, identifiers are filtered by using tags from StackOverflow (SO), a popular *Q&A* site where programmers with a wide range of experience share knowledge by asking and answering questions. It means that identifiers that are not in the tags list from SO are discarded. Tags in the SO site represent concepts related to *software development*, *computer science*, *programming languages*, *algorithms*, *etc.*, that allow to group similar questions together and are maintained by the community, which makes them a diverse and curated set of categories, concepts and terms that we consider to be valid descriptors for the projects to be tagged. In the same way a tag on a question in SO can give a reader an idea about the subject of the question,

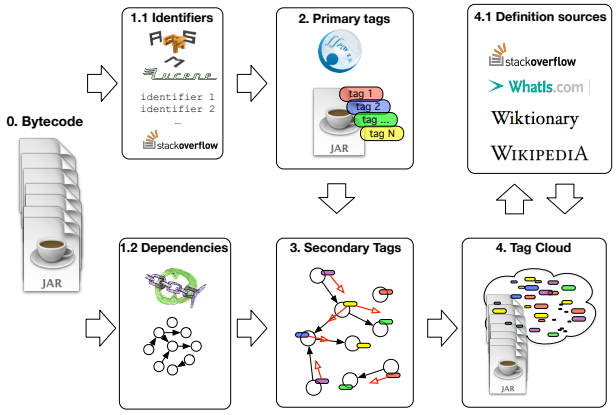


Fig. 1: The Sally approach for automatic tagging of software

we expect it to give information about the software projects it is assigned to.

B. Resolving Dependencies

Software reuse can be seen as the inclusion/adaptation of previously developed software artifacts into new projects with the intent of using their functionality; this inclusion creates a dependency relation between projects. More formally, it is considered that a project u depends on a project v when there is at least one method invocation, object instantiation or inheritance relation from classes in u to classes in v. We refer to these cases that create dependencies as *dependency calls*.

If projects are modeled as nodes and dependency relations as edges, a dependency graph can be obtained from a repository. Furthermore, a weighted graph can be obtained if each edge is labeled with the number of dependency calls between the projects connected by it. The dependency resolution process uses **DepFind** [17] to find dependency relations between projects and creates a weighted dependency graph from the repository. Finally, a *Dependency Metric* for each pair of connected vertices is computed; the metric is defined as:

$$D_{uv} = \frac{dc(u,v)}{\sum_{i=1}^{i=n} dc(u,i)}$$

where $dc(x,y)$ is the number of dependency calls made from project x to y and n is the number of projects x depends on.

C. Assigning Tags to Projects

For each project, a set of tags is found and assigned to it. Each tag is defined as a tuple (*name*, *relevance*) where *name* is the name of the tag and *relevance* is a value between 0 and 1 that describes how relevant is the tag for the particular project. For example, if we were examining the project `JUnit`, we would expect to have tags whose names are related to *testing* with high relevance measures. *Sally* uses identifiers and dependencies extracted from bytecode to identify two types of tags: *primary* and *secondary*.

Primary Tags: These tags stem from the analysis of a particular project by itself, i.e., only taking into account the identifiers extracted from the bytecode and ignoring dependencies. To obtain the primary categories, **Gensim** [18] is used to compute TF-IDF [19] values for the identifiers which are then used to obtain *relevance measures* for each one of the terms. The relevance measure is calculated as:

$$\frac{tfidf(t_i)}{\sum_{k=1}^{m} tfidf(t_k)}$$

where $tfidf(t_i)$ is the TF-IDF value for identifier i and m is the number of identifiers obtained from the project. Finally, identifiers are sorted in descending order by relevance and their roots (recall that identifiers are stemmed after being extracted) become the primary tags.

Secondary Tags: The secondary tags of a project are the primary tags of its direct dependencies[1] with their relevance measures scaled using the *dependency measure* that was previously computed. The secondary tags are also sorted in descending order by their relevance values. The number of tags that are obtained for each project can be customized either by the number of required tags or by establishing a relevance threshold, e.g., only tags with a relevance measure $> 15\%$.

D. Displaying Tags Definition

Depending on the background and programming experience of repository users, it is possible that tags generated by *Sally* could be perceived as irrelevant. To deal with this, given a tag, *Sally* is able to obtain definitions from various information sources. Currently, *SO*, *Wikipedia*, *Wiktionary* and *TechTarget* are the supported sources for concept definition.

Having an automatic way to obtain these definitions aids repository users and developers to get a better understanding of the tags. However, the main reason for developing the definition module is that although the selected identifiers are related to the application domains of projects, they are not necessarily enough to describe them; this is because using identifiers as tags can present low-level information about projects, e.g., frameworks, communication protocols, related programming languages, technologies, etc. In order for a user to find the concepts that describe what a project's application domain is, it is necessary to go beyond the details embedded in the identifiers by relating them to more general concepts. Future work will be devoted to automatic definition of concepts. Our current implementation, links tags to definitions; we expect definitions to be the bridge between identifiers and the complex concepts that serve to describe the application domains of software projects.

E. Tag Cloud Output

In order to visually present the obtained information in a useful way, Sally's user interface presents the extracted tags and their relevance measures as a **tag cloud**, where the size of each tag is directly related to the relevance of the tag for the project that is being visualized. The intention behind this decision is to provide visual aid to users in finding the most relevant tags for a particular project, i.e., although all extracted tags are related to a project in one way or another, there are some of them that are more relevant than the others.

[1] A deeper exploration of the dependency graph (i.e. using transitive dependencies) was tried, but for most of the projects, results were not satisfactory.

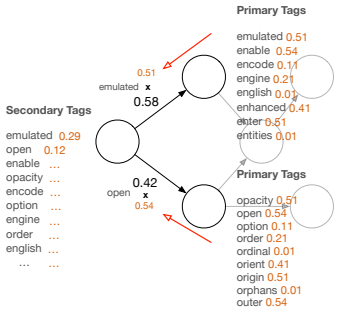


Fig. 2: Propagation of primary tags

F. Sally: The application

Sally is composed by two main components, one is responsible for core functionality and the other is in charge of making the information obtained by the core component available to users. All modules in charge of information extraction (i.e. identifier extraction / filtering, dependency resolution, category generation, definition mining) were developed as a Java Maven project that makes use of Python and Bash scripts. The web application was developed using the Meteor Javascript platform and all information is stored in a MongoDB database. *Sally* is currently available at http://sally.meteor.com.

IV. Experiment 1: Sally vs. SourceForge and MVNRepository

We compared *Sally* to the categorization/tagging approaches of two widely used online resources for browsing Java projects published as JAR files: **SourceForge**, an open source repository, and **MVNRepository**, a search engine for Maven projects. 68 Maven projects from the `net.sourceforge` groupId were selected randomly using MVNRepository and all transitive dependencies were resolved to obtain a set of 167 jar files for the evaluation.

SourceForge is an open source project hosting site with over 3.7 million registered users and is one of the most widely used alternatives by developers to publish their projects. After creating a project, a developer is presented with options to manually add tags to describe its features and also to choose from a predefined set of topics that serve as categories.

MVNRepository is a search tool for Maven projects. By providing only a groupId or artifactId, users can get the GAV coordinates (*i.e.*, *GroupId*, *ArtifactId* and *Version*) of matching projects. The site also displays information such as related books and the content of the description tag in the POM file (if available). Projects in MVNRepository are labeled with both tags and categories; tags are extracted from text in the POM file and the categories are assigned manually. To distinguish categories from tags in the following section, the same convention from MVNRepository is used: categories are presented with initial capital letters and tags are shown in lower case.

A. Experiment setup

Our goal was to compare the tags generated by *Sally* to those from SourceForge and MVNRepository in terms of their availability (whether the approaches have tags for all of the projects in the corpus) and how descriptive (how good are the terms at describing the application domains and purpose of the projects they are assigned to) they are. Therefore, the following research questions were addressed in this experiment:

- *RQ_1: How do the tags generated by Sally and the ones assigned by developers in SourceForge compare in terms of their descriptiveness?*
- *RQ_2: How do the tags generated by Sally and the ones assigned by MVNRepository compare in terms of their descriptiveness?*
- *RQ_3: How do Sally, SourceForge and MVNRepository compare in terms of the availability of tags?*

In order to answer RQs, categories and tags were manually obtained from the SourceForge and MVNRepository and *Sally* was used to generate five primary and five secondary tags for each project. One of the authors manually compared the categories/tags assigned by each approach.

B. Results

1) RQ_1 (Sally vs. SourceForge): 106 of the analyzed 167 projects were present in SourceForge. Categories in SourceForge are selected manually by developers, which allows them to have some control over the indexing of their projects on the site. However, this allows also for ambiguous categorizations and assignation of categories that do not correspond to the real purpose or domain application of projects. Also, since the set of categories is predefined, there are categories in the site that are too broad to give useful information, (e.g., software development, framework, libraries, etc.). It is important to keep in mind that SourceForge is not commonly used to host libraries that depend on each other as the Maven Central repository does, it is rather oriented towards hosting standalone applications or self-contained libraries. This is evidenced by two particular facts. Firstly, even though all the projects declared as dependencies in the pom file that was used to form the corpus belonged to the `net.sourceforge` groupId, a large amount (37%) of transitive dependencies were not available in SourceForge. This subject is mentioned in depth in the discussion of RQ_3. Secondly, since Maven software projects are generally divided into modules that have particular functions inside the project, by using the SourceForge categorization scheme (e.g., ignoring these modules) all modules get classified under the same category although this is not necessarily correct and could lead to overly broad categories for projects.

We identified 41 projects out of the 106 projects from the repository present in SourceForge (39%) as being categorized under an overly broad category. For 38 out of these 41 projects, *Sally* was able to produce more specific tags than those given by SourceForge. In most of the cases where developers assigned appropriate and specific categories, *Sally* was able to generate tags closely related to them. However, for standalone applications that are categorized under a specialized category in a manual fashion, we can not say that *Sally* produces better tags because there are high-level concepts that can not be abstracted only by looking at information obtained from bytecode.

TABLE I: Number of projects without categories per approach.

Sally		MVNRepository		SourceForge
Primary	Secondary	Categories	Tags	Categories
2	71	94	67	23
1.20%	42.51%	56.29%	40.12%	21.70%

2) RQ_2 (Sally vs. MVNRepository): MVNRepository assigns tags to projects by extracting information from their pom file[2]. Unlike SourceForge, MVNRepository is focused solely on Maven projects, this allows us to do a better comparison with *Sally*. Manual examination showed that in most cases, *Sally* is able to generate tags that are at least as descriptive as those generated by MVNRepository.

For 27 out of the 167 analyzed projects (16%), at least one of the tags generated by *Sally* exactly matched one or more tags from MVNRepository. Additionally, there are 67 projects ($\approx 40\%$) for which MVNRepository does not have any tag assigned but *Sally* does. Moreover, there is only one project for which MVNRepository has a tag and *Sally* does not: *javax.inject-1.jar* and the assigned tag is *javax*, which is filtered by *Sally* because we do not consider file names to be valid tags (e.g., project *activation.jar* can not have a tag named `activation`).

When available, tags assigned by MVNRepository were mostly considered appropriate for the projects they were assigned to. However, since these tags depend on the `description` tag from the pom file while the ones generated by *Sally* do not, *in the majority of cases Sally was able to produce more descriptive tags for Maven projects than MVNRepository*.

3) RQ_3: Availability of categories/tags: In order to measure category availability, we counted the number of projects that were indexed on the sites but had no categories or tags assigned. Table I depicts these stats. The number of projects that were not tagged by *Sally* is minimal as compared to the number of projects without categories or tags assigned by the other approaches. This is a direct consequence of the fact that *Sally* does not have special requirements for developers such as manually categorizing or describing their projects.

Since both SourceForge and MVNRepository need a certain set of conditions to be met in order to be able to categorize a project (i.e., manual categorization for SourceForge and a descriptive pom file for MVNRepository), we can conclude that *the availability of categories of Sally is superior than those of SourceForge and MVNRepository*.

C. Summary of Experiment 1

We compared *Sally* to two popular online tools with different categorization schemes and found that both SourceForge and MVNRepository have weaknesses that our proposed approach does not. The success of the categorization made by SourceForge strongly depends on developers carefully choosing categories for their projects. On the other hand, the success of the categorization scheme applied by MVNRepository depends on developers adding a description of their projects on the pom files; neither of the requirements can be guaranteed to be fulfilled at all times. Results obtained from this experiment show that *Sally* can produce competitive results without the need for any special requirements from developers.

[2]We do not include categories in the comparison because we do not have information about how they are assigned to projects

TABLE II: Computed tags for project stringtemplate-3.2

Sally		MVNRepository	SourceForge
test - 0.41	token - 0.26	Template Engines	Missing
templates - 0.36	ast - 0.26		
expr - 0.08	grammar - 0.17		
region - 0.08	gen - 0.17		
group - 0.06	antlr - 0.14		

As an example, Table II shows a summary of the tags generated by *Sally* for project *stringtemplate-3.2* [20], which is described as follows: *StringTemplate is a java template engine for generating source code, web pages, emails, or any other formatted text output. StringTemplate is particularly good at code generators, multiple site skins, and internationalization / localization. StringTemplate also powers ANTLR.* The example shows how both primary and secondary categories found by *Sally* contain terms associated to regular expressions and grammars, which directly relate to the application domain of the project under analysis. MVNRepository has a relevant category as well, however it does not present any automatically generated tags. The project is not available in SourceForge.

> Comparing tags generated by *Sally* to those from MVNRepository and SourceForge, it is apparent that *Sally* can produce competitive and in many cases superior results in terms of descriptiveness of generated tags.

V. Experiment 2: Evaluating Sally vs. MUDABlue with Professional Developers

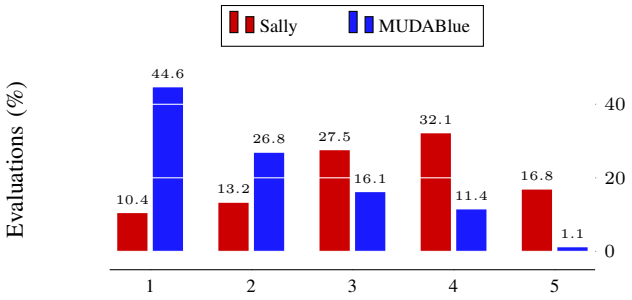
MUDABlue [3] is an unsupervised categorization approach for software projects based on Latent Semantic Analysis (LSA) [21]. We consider it as a good baseline for comparison given the fact that both *Sally* and MUDABlue are unsupervised, thus, both approaches do not need a set of predefined categories/tags. The MUDABlue approach works by extracting and filtering identifiers obtained from source code and building an term-document matrix from them. LSA is applied and cosine similarity metrics are used to cluster identifiers and software projects together.

A. Empirical Study

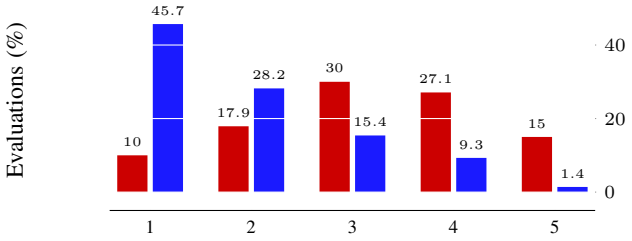
To compare *Sally* and MUDABlue, we conducted a survey in which 14 professional developers were asked to rate the *Expressiveness* and *Completeness* of the categories/tags presented by both approaches for 50 randomly selected projects from the same corpus as experiment 1. We refer to the expressiveness of a set of categories as the measure of how good is the set at describing the application domain of the software project it is assigned to; and completeness refers to whether the set of categories is able to fully describe the projects' application domain.

1) Research Questions:

- *RQ_4: How do the tags generated by Sally compare to the tags generated by MUDABlue in terms of completeness?*



(a) Expressiveness



(b) Completeness

Fig. 3: Amount of evaluations per rating for both approaches

- *RQ_5: How do the tags generated by Sally compare to the tags generated by MUDABlue in terms of expressiveness?*

Both RQs directly aim at comparing the categories/tags generated by the approaches. To answer them, we asked developers to rate — using a 5-point Likert scale where 1 represents the lowest possible score for completeness/expressiveness and 5 the highest— ten terms generated by MUDABlue and ten terms generated by *Sally* for each of the 50 projects. The tags obtained from *Sally* correspond to five primary tags and five secondary tags. When no secondary tags were available, ten primary tags were extracted. All tags were presented to developers without the relevance measure to avoid negative effects on the validity of the study.

B. Results

Figures 3a and 3b depict the ratings given by developers to the categories/tags presented by both approaches. It can be seen that *Sally* obtained the lowest score possible (1) for both Expressiveness and Completeness on close to 10% of the evaluations, while MUDABlue obtained it on approximately 45% of them. *Sally* obtained a score greater or equal to 3 on 76% for Expressiveness and 72% for Completeness versus 29% and 26% for MUDABlue. Finally, *Sally* obtained top scores on 16.8% and 15% while MUDABlue was below 2%.

> In general, developers perceived the tags by *Sally* to be superior to the category names generated by MUDABlue regarding both expressiveness and completeness.

VI. Conclusion

We proposed *Sally*, and automated tagging approach for generating useful tags for software projects by analyzing information obtained from bytecode, which makes it also applicable to closed-source repositories. Additionally, *Sally* makes use of information obtained not only from projects by themselves but from their dependencies, which allows to produce tags that describe the projects as well as the context in which they are used. *Sally* is able to work without the need for predefined tags nor any special requirements from developers, making it an attractive approach for automatically tagging software projects in large repositories. In addition to the approach, we developed a web-based application that presents the generated tags in an intuitive way that provides information about how each tag is relevant for a given project.

Acknowledgements. This work is supported in part by the NSF CCF-1525902 and NSF CCF-1253837 grants. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

References

[1] J. Guo and Luqi, "A survey of software reuse repositories," in *ECBS'00*.

[2] S. Kawaguchi, P. Garg, M. Matsushita, and K. Inoue, "Automatic categorization algorithm for evolvable software archive," in *IWPSE'03*.

[3] ——, "Mudablue: an automatic categorization system for open source repositories," in *APSEC'11*, Nov 2004, pp. 184–193.

[4] S. Ugurel, R. Krovetz, and C. L. Giles, "What's the code?: Automatic classification of source code archives," in *KDD'02*, 2002, pp. 632–638.

[5] M. Linares-Vásquez, C. McMillan, D. Poshyvanyk, and M. Grechanik, "On using machine learning to automatically classify software applications into domain categories," *EMSE 19/3*, pp. 582–618, 2014.

[6] C. McMillan, M. Linares-Vasquez, D. Poshyvanyk, and M. Grechanik, "Categorizing software applications for maintenance," in *ICSM'11*.

[7] J. Escobar-Avila, M. Linares-Vásquez, and S. Haiduc, "Unsupervised software categorization using bytecode," in *ICPC'15*, 2015.

[8] G. Di Lucca, M. Di Penta, and S. Gradara, "An approach to classify software maintenance requests," in *ICSM'02*, 2002, pp. 93–102.

[9] E. Alpaydin, *Introduction to machine learning*. MIT press, 2004.

[10] K. Tian, M. Revelle, and D. Poshyvanyk, "Using latent dirichlet allocation for automatic categorization of software," in *MSR'09*.

[11] T. Wang, H. Wang, G. Yin, C. Ling, X. Li, and P. Zou, "Mining software profile across multiple repositories for hierarchical categorization," in *ICSM'13*, Sept 2013, pp. 240–249.

[12] F. Thung, D. Lo, and L. Jiang, "Detecting similar applications with collaborative tagging," in *ICSM'12*, Sept 2012, pp. 600–603.

[13] J. Al-Kofahi, A. Tamrawi, T. T. Nguyen, H. A. Nguyen, and H. A. Nguyen, "Fuzzy set approach for automatic tagging in evolving software," in *ICSM'10*, Sept 2010, pp. 1–10.

[14] S. Wang and L. J. Lo, D., "Inferring semantically related software terms and their taxonomy by leveraging collaborative tagging," in *ICSM'12*.

[15] [Online]. Available: http://asm.ow2.org/

[16] [Online]. Available: https://lucene.apache.org/

[17] [Online]. Available: http://depfind.sourceforge.net/

[18] R. Řehůřek and P. Sojka, "Software Framework for Topic Modelling with Large Corpora," in *LREC'10*, May 2010, pp. 45–50.

[19] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[20] [Online]. Available: http://www.stringtemplate.org/index.html

[21] T. K. Landauer, P. W. Foltz, and D. Laham, "An introduction to latent semantic analysis," *Discourse processes 25/2-3*.