A model for automatic categorization of software applications using non-parametric clustering and bytecode analysis.

Javier Ricardo Escobar Avila
Ingeniero de Sistemas

Universidad Nacional de Colombia
Facultad de Ingeniería
Departamento de Ingeniería de Sistemas e Industrial
Bogotá, D.C.
Mayo de 2015

A model for automatic categorization of software applications using non-parametric clustering and bytecode analysis.

Javier Ricardo Escobar Avila
Ingeniero de Sistemas

Thesis Work to Obtain the Degree of
Magister en Ingeniería - Sistemas y Computación

Advisor
Mario Linares Vásquez, Ph.D(c)
PhD(c) Computer Science

Coadvisor
Jairo Hernan Aponte Melo, Ph.D.
Doctor en Ingeniería de Sistemas y Computación

Universidad Nacional de Colombia
Facultad de Ingeniería
Departamento de Ingeniería de Sistemas e Industrial
Bogotá, D.C.
Mayo de 2015

Title in English

A model for automatic categorization of software applications using non-parametric clustering and bytecode analysis.

Título en español

Un modelo para la categorización automática de aplicaciones de software usando clustering no paramétrico y análisis de bytecode

Abstract: Automatic software categorization is the task of assigning software systems or libraries to categories based on their functionality. Correctly assigning these categories is essential to ensure that relevant libraries can be easily retrieved by developers from large repositories. State of the art approaches rely on the semantics reflected by identifiers and comments in the source code of the libraries in order to determine their category. However, these approaches fail when the source code of the libraries is not available. In this document, we describe a novel approach for the automatic categorization of Java libraries, which needs only the *bytecode* of a library in order to determine its category. We show that the approach, based on Dirichlet Process Clustering with automatic labeling, is able to successfully categorize libraries from the Apache Foundation Repository.

Resumen: Categorización automática de software es la tarea de asignar categorias o etiquetas a aplicaciones o librerias para representar su funcionalidad. Una asignación correcta de estas categorías es esencial para asegurar que las librerias puedan ser fácilmente consultadas y recuperadas por los desarolladores, cuando estos últimos usan grandes repositorios de software. Técnicas actuales se basan en la información semántica reflejada en los identificadores de código fuente y sus comentarios con el objetivo de determinar su categoría. Sin embargo, estas técnicas no son adecuadas cuando el código fuente de las aplicaciones o librerias no está disponible. En este documento, se describe una nueva técnica para la categorización automática de librerias escritas en Java, la cual necesita solo el *bytecode* de las librerias para asignarles una categoría. Este documento muestra que la técnica, basada en Dirichlet Process Clustering con etiquetado automático de clusters, es capaz de categorizar exitosamente librerias almacenadas en el repositorio del la Fundación Apache.

Keywords: Software categorization, bytecode, non-parametric clustering, automatic labeling

Palabras clave: Categorización de software, bytecode, clustering no paramétrico, etiquetado automático

Acceptation Note

Thesis Work

Jury
Sonia Haiduc

Jury
Luis Fernando Niño

Advisor
Mario Linares Vásquez

Coadvisor
Jairo Hernan Aponte Melo

Bogotá, D.C., Mayo 11 de 2015

Dedication

This work is dedicated to Angela, who encouraged me all the time. Likewise, to my parent, brothers and specially to my aunt Laura, who always offered their support.

Acknowledgments

I am and will always grateful to my advisors Mario Linares Vásquez and Jairo Aponte, who have done an invaluable contribution to my professional training and so it has made me a better person. Their dedication and guidance were the major factor that allow me to reach the dissertation goal.

I express my grateful to the members of ColSWE research group from Universidad Nacional de Colombia and SERENE group from Florida State University for their time and feedback. Their opinions contributed a lot to the objectives of this thesis.

To my love, Angela, thanks for her unconditional support and sacrifice which encourage me to achieve this goal. I also grateful with my family, for their patience and support.

Finally, thanks to all those who supported me to write and conclude this thesis.

Contents

List of Tables

List of Figures

<

CHAPTER 1

Introduction

1.1 Background and Justification

Software reuse is defined as the use of software artifacts and libraries already developed in the implementation of new software applications. Software reuse enables developers and organizations to:

- Improve productivity: Time to market of applications is lesser because some functionality (mostly generic functionalities as logging, database access, transactions management, etc.) is delegated to software components previously built and tested.
- Improve quality: The construction of reusable software libraries can use an strict testing plan, which guarantees its quality. Therefore, the testing plan of a new application only focuses on testing the new source code. Additionally, a highly used library is followed by a community of users and developers concerned about its quality.
- Improve application performance: A widely reuse of a library encourages developers to improve its source code and architecture in order to increase its quality. This means that developers carry out a constant maintenance and improvement activity, launching new releases of the library. Each new version of this reusable library can upgrade the performance of applications which make use of it.

Likewise, Mohagheghi and Conradi [29] report, using a review of previous works oriented to software reuse in the industry between 1994 and 2005, that practicing reuse:

- Reduce errors and failures density.
- Decrease the effort required for implementing a new application.
- Reduce significantly reworking on source code, using systematic software reuse
- Improve the assignment of staff, taking advantage of their experience, because the specialized developers are in charge of building reusable assets.
- Improve the reuse without modification (black box reuse) in large projects.

- Improve software comprehension, because reusable components are built with reusable architectures and enhanced abstractions.

Software repositories support software reuse through the centralization and indexation of reusable software assets, particularly software libraries. These repositories, such as SourceForge[1], OpenHub[2], and Maven Central[3] offer a single place to search and retrieve software libraries to be reused in existing or new application. The success of this model is evidenced by the large number of software libraries hosted in the repositories, which increases on a daily basis. For example, Maven Central contains more than 90 thousand unique artifacts[4] and OpenHub contains more than 300 thousand projects (however most of them are inactive)[5].

Given the size and the changeability of the software repositories, there is therefore the need to ensure that these libraries remain easily accessible to developers. Previous work initially focused on building indexes or search engines to look for libraries in these repositories; for example [23] and [12] used documentation of software libraries to create information retrieval systems to support the identification of software applications.

Nowadays, a common approach to support this task is to assign a set of keywords or *categories* to software libraries which describe its purpose or main functionality. Developers of this libraries can assign these categories by hand by either coming up with new labels or choosing among the ones already defined by the software repository, which is a time-consuming activity[14]. For example, if the repository does not offer a proper category for a new application, the developer should search and define by hand the most appropriate category, even though this is not accurate at all. On the other hand, if the software repository does not provide a predefined set of categories, the developer will assign arbitrary labels which can increase the complexity of the retrieval process.

Automatic categorization of software libraries reduces the manual effort required for assigning a category to a library[28]. Previous works have proposed successful techniques based on information retrieval and text classification approaches [15, 16, 14, 28, 45, 22]. The main input of existing approaches is the source code of the libraries, leveraging the text found in identifiers and comments to extract semantics about the functionality of the library. The reasoning behind these approaches is that the vocabulary used to name the source code identifiers is closely related with the application domain of the library. Thereby, information retrieval approaches can extract this knowledge in order to summarize the main purpose of the library. The main limitation of these techniques is that the source code is not always available, as in the case of repositories with proprietary libraries. To deal with this limitation, a new type of approach is needed, which relies only on artifacts that are always available.

The Java environment is an interesting place to design automatic categorization approaches. In the first place, its one of the most used programming languages[6], and the design of the programming language enables to obtain the semantics about the functionality of the code from the compiled artifacts: the source code of the applications written in Java is "compiled" in a set of instructions to be interpreted by the Java Virtual Ma-

[1] `http://sourceforge.net/`
[2] `https://www.openhub.net/`
[3] `http://search.maven.org/`
[4] `http://search.maven.org/#stats`
[5] `https://www.openhub.net/explore/projects`
[6] `http://spectrum.ieee.org/computing/software/top-10-programming-languages`

chine. This set of instructions is called *bytecode*. Since the bytecode is needed to run the application, it will always be available. While some information is lost in the compilation process (e.g., comments), the bytecode still contains most of the textual information found in the source code (i.e. the identifiers). The bytecode can therefore constitute a good replacement for source code in the process of automatic library categorization.

1.2 Problem Definition

In the same manner that developers implicitly introduce knowledge about the application domain and functionality of a software library – through proper names for source code identifiers–, they explicitly introduce the same knowledge in the *profiles* and main web pages of the software projects. Commonly, developers release a software library using i) a web page with a text description of the library –in some cases a detailed description–, or ii) a software repository with a particular software profile. A software profile is composed by a high-level text description of the library, and in most of the cases, a set of tags or *categories* which succinctly summarize the functionality or application domain of the library. Therefore, if the profiles of several software libraries contain the same category, it means that such libraries may share the same application domain, and additionally, they may have implemented similar functionalities. Previous work have used software profiles to categorize software applications [45]. Thus, at this point, there are two approaches to extract the semantics about the functionality of a library: i) using the source code identifiers, or ii) using the text and categories extracted from software profiles.

If a particular software repository contains only the bytecode of its libraries (i.e. the source code is not available), and additionally these libraries were not categorized by their developers, an automatic approach can analyze the bytecode to create *clusters*, identifying similar libraries that share some semantics about its functionality. Nevertheless, these clusters are not useful by their own because there is not an easy way to describe the common properties or semantics within each cluster, so we need an human readable way to describe each cluster. What we do know at this point is that libraries in each cluster were created using similar vocabulary related with a particular application domain, so we could use this vocabulary in some way to create a succinct and human readable description to the cluster.

In addition, we can create descriptions for a particular set of categories, which are provided by other software repositories. Using the text of the profiles of the libraries tagged with a category, we can extract the semantics about the functionalities of those libraries, and with these semantics, we can obtain the relevant vocabulary used in the construction of such software profiles. At the end, we will get in one hand, clusters of similar applications (created through the analysis of bytecode) and the relevant vocabulary used by the application within each cluster; and in the other hand, a description of each category (created through the analysis of software profiles) represented by the relevant vocabulary used in the profiles of the profiles tagged with that category. If we identify a strong relationship between the vocabulary extracted from a particular cluster and the vocabulary extracted from a particular category, we can assign the category to the cluster, which in turn means that the libraries contained in the cluster are automatically categorized.

However, there are inherent issues related with the clustering process of the bytecode libraries. The first one, the size and changeability of a software repository, together with

the unstructured nature of the bytecode, do not allow to know beforehand the number of clusters that can be identified; indeed, the unstructured nature of bytecode lead to model the clustering process with a potentially infinite number of cluster, since the combinations of source code identifiers in a software library are virtually infinite. In this sense, clustering algorithms such as *k-means* (and its variations), or information retrieval approaches such as Latent Dirichlet Allocation (LDA) which need the number of topics (widely known as k) as a parameter are not appropriate to perform the task previously described. Another issue is related with the description and the nature of each cluster; each cluster should be constructed based on the similarity of the semantics about functionality of the software libraries, and these common semantics should be represented by the clustering algorithm. Hierarchical clustering algorithms such as Hierarchical Agglomerative Clustering (HAC) with single-linkage, average-linkage or complete-linkage schemes, and density based clustering algorithm such as DBSCAN can identify automatically the number of clusters in the dataset, but do not represent the common semantics within each cluster; particularly, HAC would need a cut point in the dendogram to obtain a configuration of clusters to model several groups of libraries. Therefore, we need an algorithm that i) can model an arbitrary and unknown number of clusters, and ii) can offer a description about the semantics within each cluster. To overcome these limitations, we use Dirichlet Process Clustering (DPC), which is used in non-parametric models of data, particularly in Bayesian mixture models. DPC is able to recognize the distribution representation of the data without a target number of clusters, therefore this algorithm meets our first condition. Additionally, DPC assumes that each component (cluster) is generated by a particular probability distribution; if that probability distribution is described by the vocabulary given by the semantics contained in the cluster, we can obtain a description of the clusters, which meets our second condition. DPC is called a non-parametric clustering algorithm because it does not need the number of clusters as parameter.

1.3 Contributions

In this sense, this document presents an approach called DBYCAT (**D**irichlet-based **BY**code Software **CAT**egorization), which leverages the bytecode and software profiles to automatically categorize software libraries. The bytecode is extracted from the software libraries to be categorized, then similar libraries are grouped using Dirichlet Process Clustering, a non-parametric clustering algorithm. Finally, the clusters are automatically labeled using existing categories extracted from software profiles stored in software repositories. The trained model then can be used to categorize a new library assigning it to one of the existing clusters; since the clusters were previously labeled with categories, these categories become the new categories of the library.

Then, the principal capabilities of DBYCAT are:

1. It can automatically categorize libraries written in Java whose source code is not available, through the analysis if its bytecode.

2. It does not need the number of clusters as a parameter, because it is able to determine the appropriate number of clusters.

3. It automatically assigns categories to the clusters, comparing the vocabulary of the libraries within each cluster with the vocabulary present in software profiles tagged with a particular category.

4. Given a baseline of categorized libraries in a repository using DBYCAT, it can additionally suggests relevant categories to new software libraries, measuring the similarity of the vocabulary of the new library with the relevant vocabulary present in the existing clusters. The categories previously assigned to the closest cluster becomes the suggested categories for the new library.

1.4 Goal

This thesis aims at **defining and implementing a software categorization model via non-parametric clustering with automatic labeling, using the extraction of features from java bytecode statements.**

We defined the following sub-goals:

- Build a bag-of-words representation for domain categories of software repositories, in particular the categories in Apache Software Foundation.
- Define a non-parametric software clustering model based on the automatic extraction of features from Java bytecode statements.
- Define an automatic labeling process for software clusters using the bag-of-words representation of predefined domain categories in a software repository.
- Validate the accuracy of the model (clustering and automatic labeling) using a cross-validation study (i.e., using labeled applications from Apache Software Foundation).
- Validate the accuracy of the model (clustering and automatic labeling) using a user study (i.e., asking users to validate the correctness of the categories provided by the model).

And in general, this thesis makes the following contributions:

- An approach, DBYCAT, for automatic categorization of software libraries, which is ideal in environments where the source code is not available; and
- An empirical study with 17 participants which evaluated the relevance of the categories suggested by DBYCAT to 15 software systems randomly chosen.

1.5 Outline

This document is structured as follows:

- Chapter 2 presents an brief survey of existing techniques and tools addressing automatic software categorization, clustering to support software engineering tasks, and applications of bytecode analysis.

- Chapter 3 introduces the model for automatic software categorization using non-parametric clustering on Java bytecode.
- Chapter 4 details the validation process using an experiment involving libraries retrieved from Apache Software Foundation and a survey.
- Chapter 5 draws the conclusions and future work.

CHAPTER 2

Related work

Previous work have focused on developing automatic tools to index and retrieve retrieve software artifacts from repositories. In particular, search engines and automatic categorization approaches have been proposed, which leverages the lexical information contained in such software artifacts. Earlier approaches focused in the construction of indexes, to be used as a search engines. GURU[23] and Exemplar[11, 10, 27] are two examples of this type of systems:

Maarek et al. [23] proposes a technique to create indexed software repositories, through the use of information retrieval algorithms. A retrieval system called GURU let easily explore similar components to help reuse. The system uses the documentation provided with the software packages (e.g. man in UNIX); then an indexing space is built, where the similarity between components is computed by its *syntagmatic lexical affinity*. Similar applications are detected by clustering common terms among components.

Exemplar [11, 10, 27] is a search engine to find highly relevant software projects through the analysis of calls to APIs made by the applications. When a user enters a query, Exemplar searches for API calls which are relevant for that query, then it suggests the set of applications that use the set of API calls. Exemplar was implemented through two crawlers; the first one retrieved a great amount of projects from SourceForge and the second one traversed this information to create the search engine.

Then, approaches to summarize the purpose of the libraries were proposed, which are mostly focused on the analysis of the source code of the software applications. The reasoning behind most of these techniques is that developers introduces their knowledge about a particular application domain into the source code using the vocabulary of such domain to create source code identifiers. Thus, automatic approaches based on information retrieval and natural language processing can automatically identify this vocabulary to label the applications with meaningful categories.

Kawaguchi et al [15] three approaches to automatically categorize software systems thorough the analysis if its source code. The first approach uses a similarity measure for software systems based on the code clones that those systems shares. The second one uses a traing dataset to create classification criteria based on a decision tree. The last one uses Latent Semantic Analysis (LSA) and cosine similarity to evaluate the similarity of software systems belonging to the same category.

Kawaguchi et al. [16] introduces MUDABlue as a tool for automatic software categorization. MUDABlue uses an algorithm that (i) extracts source code identifiers (variable and method names); (ii) creates a matrix with the identifiers; (iii) removes outliers; (iv) defines the categories grouping the identifiers; (v) finds the software clusters from the clusters of identifiers; (vi) sets the title of each software cluster, defining its main category. The experimentation used software projects extracted from SourceForge.

Another approach is proposed by Kuhn et. al. [19, 21], using a technique called *semantic clustering* that defines the semantic similarity between source code identifiers. Based in Latent Semantic Analysis, which is a statistical method to induce and represent the aspects of the meanings of the words and sentences (documents) wrote in natural language, generating representations of the documents through real value vectors, this technique: (i) preprocesses the source code; (ii) applies LSA to represent source code attributes; (iii) perform the clustering using a similarity measurement over source code representations; (iv) automatically assigns the clusters tags. The experimentation process was realized with JEdit and JBoss , two open source Java projects.

LACT[14] is a technique for automatically categorize software systems, which is based on Latent Dirichlet Allocation. Given a document which represents a software library , LACT uses LDA to model a mixture of topics, and each topic as a distribution over words. Thus, each software system can be described with a particular set of topics. To generate the categories, LACT takes the topics found by LDA in the whole corpus and computes the cosine similarity between each pair of topics; at the end, the closest topics are clustered together to build a category. LACT provided comparable results as MUDABLue.

McMillan et. all. [28] and Linares-Vasquez et. al.[22] have used the calls to external Application Programming Interfaces (API) to categorize software applications. Software applications retrieved form SourceForge and Sharejar were parsed in order to extract package names and classes names of APIs called by these applications. Then, the proposed approaches used several machine learning-based algorithms such as decision trees, Naïve-Bayes classifier and Support Vector Machines, to perform the categorization. It should be noted that the applications retrieved from Sharejar did not contain source code, so in their work the bytecode of the applications was used to support the automatic categorization.

Wang et. al. [45] describes an approach to categorize software application leveraging their profiles. The profile of an application commonly contains a description of the project and a set of tags, Additionally, a software project can have a different profile in several repositories. Thus, this approach takes the profiles of software projects from several repositories to categorize them. The study uses the set of labels provided by SourceForge, building a local binary classifer (using Support Vector Machines) to estimate whether a software application belongs or not to a particular category.

The main differences between DBYCAT and the previous approaches is that DBYCAT only relies on the analysis of bytecode. Although is true that [28] and [22] used bytecode to categorize software applications, it should be noted that these previous works used just an small subset of the information (i.e. calls to APIs) that could be extracted from bytecode. In this sense, DBYCAT has an advantage over previous works because it can be used in environments where the source code is not available.

Moreover, clustering algorithm have been used to support software engineering tasks, such as the discovery of API usage patterns [1], software clustering and architecture recovery [17, 3, 33, 46], bussiness rules recovery [32], code clones detection [30, 13], program

comprehension [21, 24, 34] and quality assesment [8, 39, 7, 2, 37, 35, 20, 6, 38]. The most used clustering algorithm in this set of previous works are Hierarchical Agglomerative Clustering [3, 46, 21, 34, 8, 39, 2, 35, 20, 6] and K-means with its variations [37, 1, 44, 32]. Therefore, DBYCAT contributes to the software engineering community through the exploration of a different clustering algorithm, Dirichlet Process Clustering, which does not need previous knowledge about the number of clusters

CHAPTER 3

Automatic software categorization using non-parametric clustering on Java bytecode.

The main principle of DBYCAT is that it builds a model for the categorization of software libraries based on a set of training data and uses this model in order to categorize new libraries. DBYCAT first clusters similar libraries together using information extracted from their bytecode. Then, the high-level text descriptions of the libraries in the training dataset (i.e., "profiles") are used to determine the most appropriate category (or label) for each cluster. The proposed approach follows several main steps: (i) data extraction, (ii) text preprocessing, (iii) model construction (which describes the clustering process and the automatic labeling process), and (iv) the categorization of new projects. In the *data extraction* step (Section 3.1), DBYCAT builds a training dataset composed by bytecode of software applications and text extracted from software profiles; together with the profiles, the set of categories to be used by DBYCAT are defined as well. Then, the *preprocessing* step (Secion 3.2) remove noise and meaningless information from the training dataset. The *model construction* step (Section 3.3) performs the clustering using the bytecode, and automatically labels the clusters with categories extracted from software profiles; therefore, the software libraries within each cluster are categorized using the labels assigned to the clusters. At this point, DBYCAT have built a categorization model, composed by the labeled clusters; this categorization model can be used to *categorize new software libraries* which are not originally in the training dataset (Section 3.4).

These mains steps are shown in Figure 3.1 and described in the following sub-sections.

3.1 Data extraction

The data extraction step builds the dataset to train the model. This dataset is composed by: i) the source code *identifiers* which are extracted from the bytecode of the software libraries, and ii) the profiles and categories of these software libraries, extracted from its web pages or extracted from a software repository.

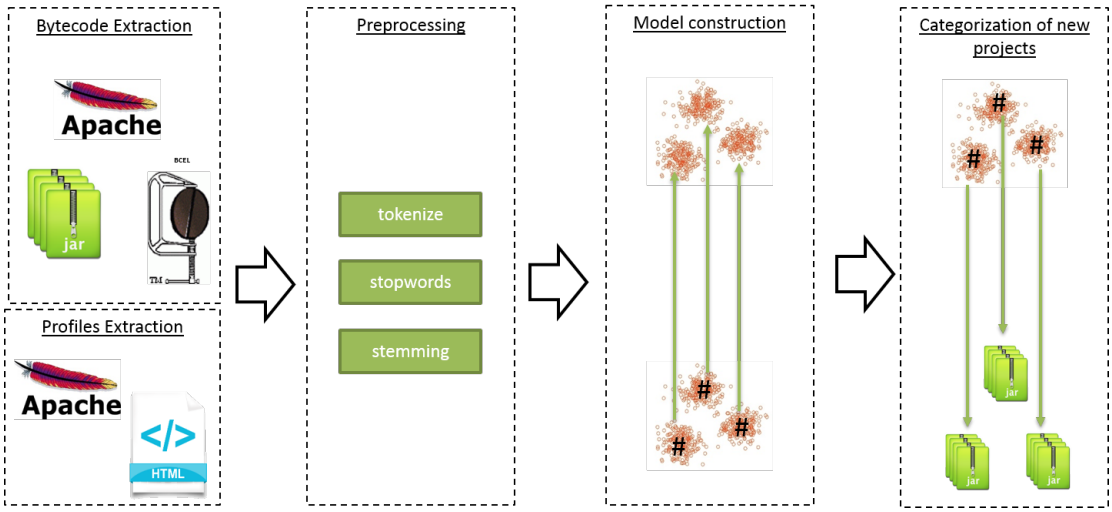


FIGURE 3.1. Automatic software categorization model (DBYCAT) based on bytecode code analysis and non-parametric clustering

3.1.1 Source code identifiers

The identifiers of the source code entities can be found in the Java bytecode, which is commonly released as .class files [1]. The set of *.class* files containing all the instructions of an entire Java application are commonly deployed using files with *.jar*, *.war* or *.ear* extensions (for simplicity, these files are going to be referred just as *.jar* files). Therefore, the first part of the training dataset is constructed with the set of *.jar* files of the software libraries that will be used to train the model. Each library in this set is analyzed using the Bytecode Engineering Library (Apache Commons BCEL)[2][42], which offers an object oriented representation of the source code entities found in the bytecode.

Using this object-oriented representation provided by BCEL, DBYCAT extracts the identifiers of the following object-oriented programming entities: **classes**, **fields** or **attributes**, **methods**, **arguments in methods** and **local variables**. These names can be obtained with its Fully Qualified Name (FQN) which is composed by the sequence of package names in which the entity is stored, ended with the identifier. Except for the identifier, a set of classes in the same package, and their methods share the same sequence of package names in the root of its FQN, so these roots do not provide distinguishing information; therefore, the roots of the FQN are ignored and only the last portion of the FQN –i.e. the name of the class or method– is preserved. At the end, all the source code statements successfully extracted from the bytecode of a single software library are compiled in a single text file, which can be considered as a document. For processing purposes, this text file is named using the name of the software library. The first part of the training dataset then is composed by the "bytecode" documents of the software libraries used to train the model, where each document contains the source code identifiers of a single software library.

[1] A *.class* file is the result of compiling a *.java* file. Then, the Java Virtual Machine uses the sequences of bytecode in the *.class* file to execute a program

[2] Apache Commons BCEL (The Byte Code Engineering Library) is "intended to give the users a convenient way to analyze, create and manipulate (binary) Java class files (those ending with .class)".

3.1.2 Software profiles and categories

Moreover, DBYCAT uses the profiles of software projects as the main input of the automatic labeling process. A software project profile is the description of the main functionality or application domain of a software library, which normally is provided by its developers. The profile of a software library can be found in two ways: (i) Using the web page of the software library, and (ii) If the software library is hosted in a software repository (e.g SourceForge and Apache Software Foundation), then the repository can provide a short description of the project. Additionally, the software repository can offer a set of categories or tags which describe the purpose or main functionality of the library. For DBYCAT, both sources of information are important: examining the web page of a software project is useful to obtain relevant terms and vocabulary related to the functionality of the library, and the categories provided by the repository are used as *labels* for the clusters that DBYCAT use to characterize the training dataset and to automatically categorize new applications.

The profiles described in the web pages of the software libraries in the training dataset were extracted by hand, forming a text file or document with all the text reflecting the functionality of the project. Since a software repository can provide a set of categories for each library in the training dataset, DBYCAT extracts the set of different categories that developers assigned to the libraries of the training dataset; an example of a software profile is shown in Figure 3.2. Finally, for each category, DBYCAT builds a document containing the text of all of the profiles of the software libraries previously labeled with this category. For example, since MySQL, HSQLDB and PostgreSQL are software systems related with databases, a single document called *database* should contain the text of the profiles extracted from the web pages of such systems.

Thus, the training dataset of DBYCAT is composed by the set of document containing the bytecode of the libraries – one document per library –, and the set of documents containing the profiles of the libraries per category –one document per category.

3.2 Preprocessing

DBYCAT uses information retrieval techniques to preprocess the documents in the training dataset. This process starts with tokenization, taking into special account the CamelCase style present in bytecode. After a lower case transformation, DBYCAT filters English stop words, numbers, java reserved words and java common words (e.g. object and string). Finally, the remaining terms are stemmed using the Porter algorithm[43]. A custom preprocessor built at top of Apache Lucene[3] performs the preprocessing steps.

3.3 Model construction

DBYCAT builds an automatic software categorization algorithm based on the clustering of similar bytecode documents. Then, the model automatically assigns tags to the clusters using the bag-of-word representation of existing categories. A detailed explanation of this process is presented as follows:

[3] http://lucene.apache.org/core/

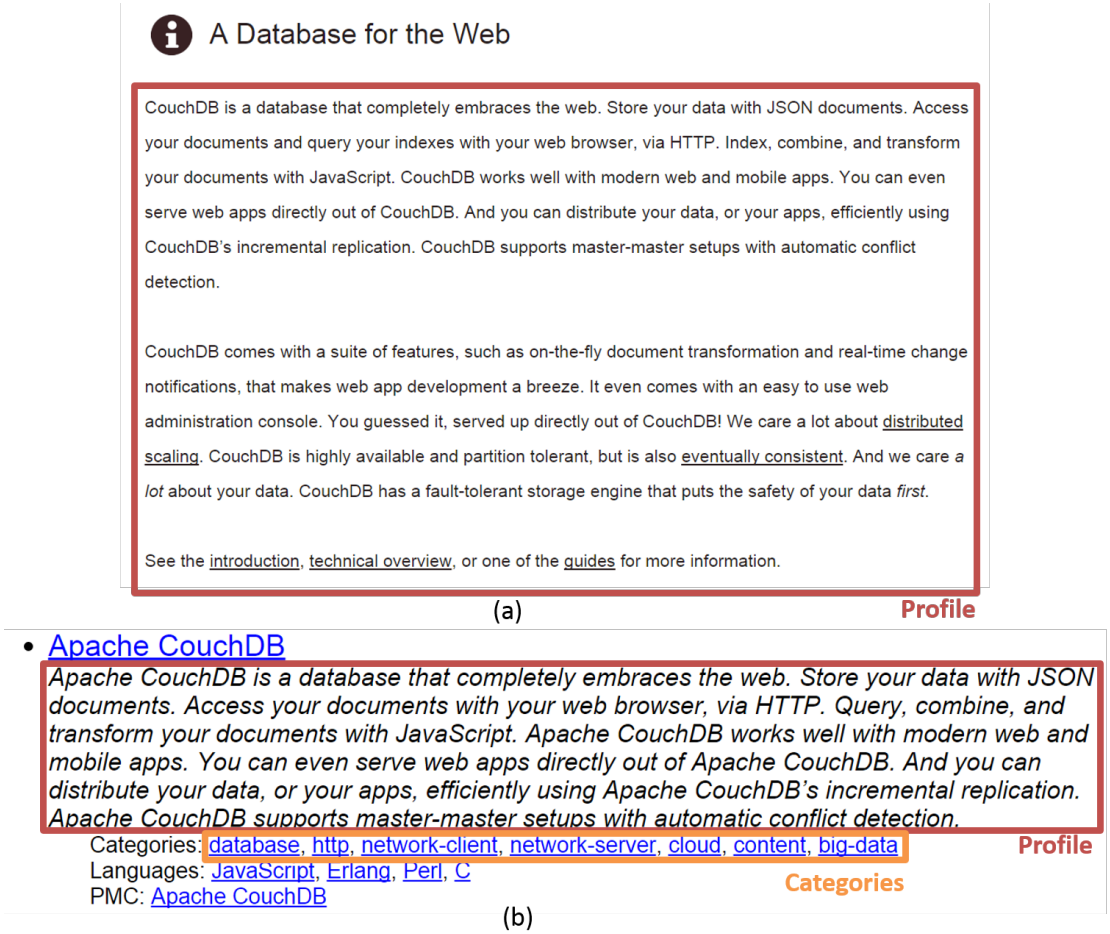


FIGURE 3.2. Example of a software profile. (a) Software profile of Apache CouchDB provided by its main web page, (b) Software profile of Apache CouchDB offered by the Apache Software Foundation. Note that the repository offers a set of categories to summarize the purpose of the library.

3.3.1 Clustering bytecode documents

Source code is intended for humans, not for computers. In this sense, the vocabulary used to name the source code identifiers of a software library should relate with its functionality and domain. Thus, the use of this vocabulary related with a specific topic along the source code can be leveraged to recover its semantics[19]. Therefore, through the identification of the semantics of a set of software libraries, DBYCAT can group them taking into account the similarity among the relevant terms that describe the purpose of each library.

3.3.1.1 TF-IDF and the Vector Space Model

The extraction of the vocabulary related with the application domain of a software library is carried out through the detection of the relevant terms of each bytecode document. DBYCAT starts using a *bag of words* representation of each software project; with this view, each bytecode document becomes a set of tuples which in turn are composed by a term and its weight regarding the document[25]. At the end of this step, the weights represent the relevance of the terms in a document.

DBYCAT computes the relevance of the terms using TF-IDF weighting. This technique assigns to a term t contained in a *bytecode document* d a weight using the following scheme[25]:

$$tf\_idf_{t,d} = tf_{t,d} \times idf_{t\cdot} \tag{3.1}$$

Where $tf_{t,d}$ or *term frequency* is the number of occurrences of the term t in document d. For the second part of equation 3.1, the *inverse document frequency* of the term t in the collection of documents (*corpus*) is given by the following equation:

$$idf_{t\cdot} = log\frac{N}{df_t} \tag{3.2}$$

Where N is the number of documents in the corpus and df_t is the number of occurrences of the term t in the corpus.

According to [25], $tf\_idf_{t,d}$ of a term is:

- Highest when t occurs many times within a small number of documents.
- Lower when t occurs fewer times in a document, or occurs in many documents.
- Lowest when t occurs in virtually all documents.

At this point, DBYCAT has transformed each document into a set of terms along with a measure of their relevance, but additionally, has detected all the terms contained in the corpus; this set of terms is called a *dictionary from identifiers* or *i-dictionary*. Taking into account this *i-dictionary*, each set of terms is seen as a *vector* formed by the weights given by (3.1). For *i-dictionary* terms that do not occur in a document, the weight is zero. At the end, DBYCAT builds a *vector space model* which is a representation of documents in a common vector space.

3.3.1.2 Non-relevant terms removal and Documents resizing

The next step performed by DBYCAT is to group documents considering their similarity. However, this similarity is influenced by the non-relevant terms which could not be removed by the preprocessing step, and by the size of the documents. For example, when calculating the similarity between documents using some similarity measure, it may happen that two vectors dramatically different appear as similar because both vectors contains several terms with very low $tf - idf$. In this case, the similarity criteria only focuses in the common but irrelevant terms with low weight ignoring the terms with high $tf - idf$. To avoid this behavior, DBYCAT removes the weights (i.e. the weight is changed to zero) of the terms of a vector whose values are lesser than the value corresponding to the first quartile of the weights of the vector.

Likewise, the size of the vectors can affects the similarity measure. For example, DBYCAT can state that a vector with many non-zero dimensions[4] is similar to many vectors with many dimensions equals to zero. It may happen because a similarity measure focuses only in the terms that both vectors share. Given a vector $\overrightarrow{V}(d) = [t_1, t_2, ..., t_n]$ where $\overrightarrow{V}$ is the vector representing the document d, this model proposes a Euclidean normalization to resize the magnitude of the vector $\overrightarrow{V}$ which is addressed with the following equation:

$$norm(\overrightarrow{V}(d)) = \overrightarrow{v}(d) = [\frac{w_1}{\sqrt{\sum_{i=1}^{n}(w_i)^2}}, \frac{w_2}{\sqrt{\sum_{i=1}^{n}(w_i)^2}}, \ldots, \frac{w_n}{\sqrt{\sum_{i=1}^{n}(w_i)^2}}] \tag{3.3}$$

3.3.1.3 Dirichlet Process Clustering

A Dirichlet Process is used in Bayesian non-parametric models of data, particularly in infinite mixture models. It is a distribution over distributions, i.e. if one pick a draw from a Dirichlet process, it is going to be a distribution itself which can be interpreted as a distribution over some probability space Θ. The Dirichlet Process is an infinite dimensional generalization of the Dirichlet distributions with a finite set of K components which is shown as follows:

$$\begin{aligned} \pi \mid \alpha &\sim Dirichlet\left(\frac{\alpha}{K}, \cdots, \frac{\alpha}{K}\right) \\ z_i \mid \pi &\sim Multinomial\left(\pi\right) \\ \theta_k^* \mid H &\sim H \\ x_i \mid z_i \{\theta_k^*\} &\sim F\left(\theta_{z_i}^*\right) \end{aligned} \tag{3.4}$$

Where $\pi = [\pi_1, \cdots, \pi_k]$ is the mixing proportion, $\alpha = [a_1, \cdots, a_k]$ is the concentration parameter of the Dirichlet distribution, H is the distribution over the parameters θ_k^* and $F\left(\theta\right)$ is the component distribution parametrized by θ[40].

Thus, for a random variable G distributed according to a Dirichlet Process, its marginal distribution have to be distributed according to a Dirichlet Distribution[9]. Specifically, let H be a distribution over Θ and α be a positive real number. Then, the random variable G is

[4]Since the Vector Space Model represent a document using a vector, each dimension of this vector stores the weight of a term contained in the document

distributed according to a Dirichlet Process with a base distribution H and concentration parameter α, written $G \sim DP(\alpha, H)$, if

$$(G(A_1), \cdots, G(A_r)) \sim Dir(\alpha H(A_1), \cdots, \alpha H(A_r)) \tag{3.5}$$

for every finite partition $A_1, \cdots, A_r$ of Θ[40].

An advantage of Dirichlet Process as a infinite mixture model is that it assumes that the data comes from an infinite number of probability distributions, i.e. it does not restrict the analysis to a particular number of clusters; however, most of the applications requires a finite mixture model or a finite set of clusters. To address this issue, the Dirichlet Process can be modeled placing most of the probability mass at the beginning of the infinite distribution, making possible to assign probabilities to clusters without restricting the number of possible clusters[5]. An approach to place the probability mass at the beginning of the infinite distribution comes from a representation of a Dirichlet Process known as *Stick Breaking Process*.

3.3.1.3.1 Stick Breaking Process The construction of the mixing proportions $\pi = [\pi_1, \cdots, \pi_k]$ for the distribution for a random variable $G \sim DP(\alpha, H)$ can be represented metaphorically as follows: Starting with a stick of length 1, it is broken at β_1, assigning π_1 to be the length of the broken portion representing the proportion of probability assigned to that component. Then the process is repeated with the remaining portion of the stick in order to assign values to π_2, π_3 and so on. Since the stick at the beginning of the process had a length of 1, the values of π_k's decrease exponentially quickly, then only an small number of clusters with a representative value of π_i will be used to model the data[40]. The *Stick Breaking* distribution over π is written $\pi \sim GEM(\alpha)$, where the letters stand for Griffits, Engen and McCloskey[31]. Thus, the Dirichlet Process mixture model can be written as:

$$\begin{aligned} \beta &\sim GEM(\alpha) \\ z_i &\sim Multinomial(\beta) \\ \theta_{z_1} &\sim H \\ x_i &\sim F(\theta_{z_i}) \end{aligned} \tag{3.6}$$

3.3.1.3.2 Chinese Restaurant Process The *Chinese Restaurant Process* is another metaphor to represent a Dirichlet Process which can explain the distributions over partitions (clusters) of data. In this metaphor, there is a Chinese restaurant with a potentially infinite number of tables. The first customer enters the restaurant and sits at the first table. Then, the second customer enters and decides either to sit with the first customer, or by herself at a new table. In general, the $n+1st$ customer either joins an already occupied table k with probability $\frac{n_k}{n+\alpha}$ where n_k is the number of customers already sitting there (all the occupants of a table k share the same dish), or sits at a new table with probability $\frac{\alpha}{n+\alpha}$[40, 5].

Using DPC, DBYCAT can identify clusters of similar applications without prior knowledge of the number of clusters but obtaining a vector of representative terms from each cluster. This vector is not meaningful by itself to describer the cluster, but can be used to establish a relation between the relevant terms in the clusters, and the relevant terms in

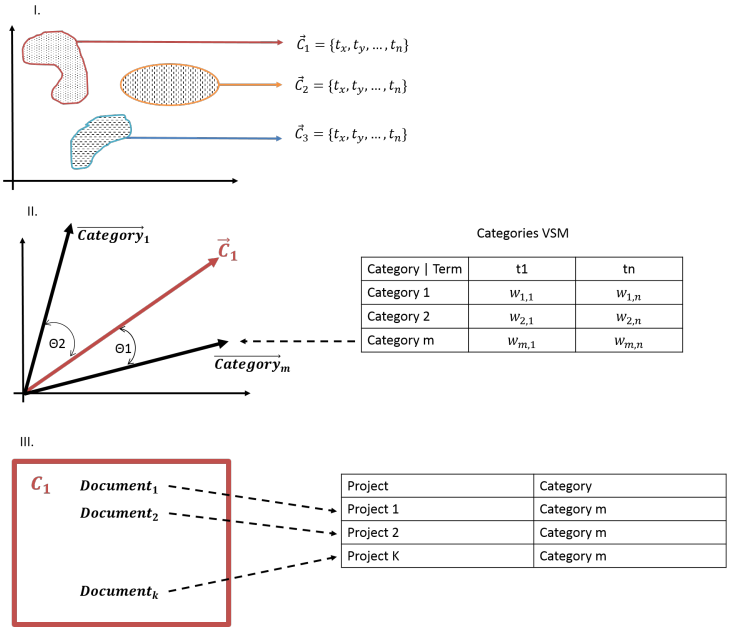


FIGURE 3.3. Algorithm to automatically assign tags to clusters. I. DBYCAT extracts the relevant terms of each cluster forming a vector. II. A cosine similarity is computed among each centroid and the set of vectors of the categories VSM. III. DBYCAT assigns the closest category to the software projects belonging to the cluster.

the profiles of the projects belonging to a particular category. The next section describes how this relationship is established in order to categorize each cluster – categorizing the libraries within the cluster as well –, providing a meaningful and succinct description of the nature of the cluster.

3.3.2 Automatic labeling using the bag-of-words representation of categories

In this step, DBYCAT automatically assigns categories to each cluster using the profiles of the software projects from which the documents were extracted. To achieve this goal, the dataset described in section 3.1.2 is preprocessed using the same methods described in sections 3.2,3.3.1.1 and 3.3.1.2 except the removal of non-relevant terms using the first quartile. At the end, DBYCAT obtains a *vector space model* representing the categories and its weighted terms as vectors; in addition, DBYCAT collects all the terms contained in the profiles *corpus* as a dictionary called *c-dictionary*.

DBYCAT describes each cluster with a vector of relevant terms (hereinafter referred to as *centroid*); this centroid represents all the documents (software projects) contained in a cluster. Thus, the model assigns the categories to the clusters by calculating the *cosine similarity* between each *centroid* and the vectors contained in the *vector space model* of categories. The closest category vectors are the assigned ones. Finally DBYCAT assigns the top 5 closest categories to each software project contained in the cluster under analysis, i.e., DBYCAT recommends a set of 5 categories. A summary of the algorithm is depicted in Figure 3.3.

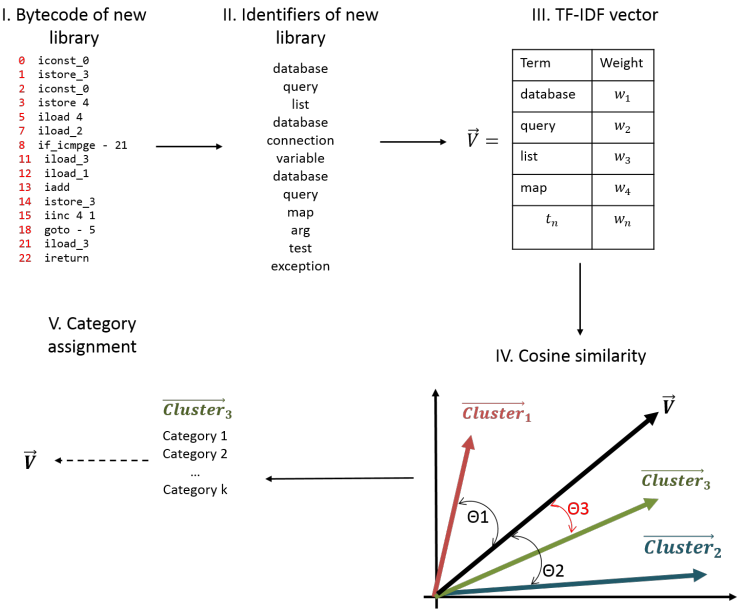


FIGURE 3.4. Algorithm to automatically assign tags to new software projects.

3.4 Categorization of new projects

At this point, DBYCAT has trained a classifier of software projects based on vectors composed by the weights of the relevant terms of each cluster —*centroids*. To categorize a new software library, DBYCAT extracts the source code identifiers from its bytecode (as described in section 3.1.1) and builds a document with these identifiers. Then, DBYCAT preprocesses the document as described in section 3.2 and extracts the vector of weights using tf_idf as described in section 3.3.1.1. Finally, DBYCAT computes the *cosine similarity* of this vectors with each *centroid*; the categories previously assigned to the closest *centroid* are the categories of the new software project. A representation of this process is depicted in Figure 3.4

3.5 Summary

DBYCAT uses the bytecode and profiles of software projects to categorize software libraries. To assign a set of categories for a new software library, DBYCAT: (i) analyzes the bytecode given by the training dataset in order to cluster similar libraries together. Each cluster contains a vector of relevant terms that describes the nature of the libraries within the cluster; (ii) analyzes the software profiles and categories given by the training dataset; DBYCAT processes the software profiles of the projects that belongs to each category in order to find a vector of relevant terms to describe each category appropriately (iii) assigns a set of categories to each cluster, computing the cosine similarity between the representative vector of each cluster and the vector of relevant terms of each category found in the step (iii), then, DBYCAT assigns the closest categories to the cluster; (iv) builds a vector of terms for the new software library through the analysis of its bytecode, (v) computes the cosine similarity between this vector of terms and the representative vector of each cluster, and (vi) identifies the closest cluster and suggests the categories previously assigned to it.

CHAPTER 4

Experiments and results

To validate the capabilities of DBYCAT, two studies were conducted. The first study focused on estimating the accuracy and generalization capabilities of DBYCAT, i.e., we wanted to know whether the categories suggested by DBYCAT are relevant to the software libraries. The second study aimed at knowing if the information provided by bytecode is enough to perform the categorization.

4.1 Experimental design

To estimate the correctness and usefulness of DBYCAT, we analyzed 158 software libraries written in Java, which are published and maintained by the Apache Software Foundation (ASF)[1]. This repository contains the .jar files (i.e. the bytecode) and the software project profiles. Likewise, developers of these software projects categorized them using a set of 18 predefined categories. It should be noted that the developers of these projects may assign one or more categories to each software project.

In addition, a set of 15 software libraries not developed nor maintained by ASF were randomly collected to be analyzed and categorized by DBYCAT, in order to measure its capability of categorizing new software libraries, it means, software projects not related to ASF. Thus, the dataset composed by the libraries extracted from ASF is suitable to work as a training set and the dataset composed by the libraries not developed by ASF becomes in our testing set.

On the other hand, previous works in software categorization have focused on analyzing the source code to extract semantic information about the software artifacts, because the source code is the most used artifact in which developers reflect their knowledge about a particular application domain. Since DBYCAT is based on the extraction of lexical information only from the bytecode of the applications, that is to say, the bytecode does not offer all the information available in the source code (e.g., the comments or JavaDoc documentation), this study is interested in comparing the results (i.e., the appropriateness of the recommendations) of DBYCAT trained with bytecode documents, and the results of DBYCAT but trained with identifiers and comments extracted from source code. The

[1] http://www.apache.org/

intuition behind this comparison is that the bytecode will become a good replacement for the source code – for the software categorization problem – only if the accuracy of the recommendations provided by DBYCAT trained with information extracted from bytecode is comparable (i.e. it is not statistically different) with the accuracy of the recommendations made by DBYCAT but trained with information extracted from source code.

In summary, given a categorization model DBYCAT-1 trained through the analysis of the bytecode of the ASF software projects, we propose a first study *Study-1* , with the following objectives:

1. To compare the categories recommended by DBYCAT-1 to some ASF projects with the categories previously defined by its developers (*accuracy*).
2. To establish wether it is possible to recommend categories to new software projects (*generalization*).

Study-1 intends to address these goals taking into account that a software project can be categorized using only one label (i.e. single label), or using a set of labels (i.e. multi-label).

Finally, given an additional categorization model DBYCAT-2 trained win the information extracted from the source code of the ASF software projects, we propose a second study *Study-2*, to establish whether there is a statistical difference between the series of metrics obtained from the model trained with bytecode and the model trained with source code.

4.1.1 Research questions

In the context of this thesis, the following research questions are formulated:

Research Question 1 (RQ1): *What is the accuracy of DBYCAT when recommending a category (single-label software categorization)?* This research question aims at establishing if DBYCAT can suggest a single correct category for an ASF library, given that the model is trained through the analysis of the bytecode of others ASF libraries.

Research Question 2 (RQ2): *What is the accuracy of DBYCAT when recommending several categories (multi-label software categorization)?.* For this question, this study intends to verify if the set of categories suggested by DBYCAT contains one or more correct categories for and ASF library.

Research Question 3 (RQ3): *Can a library not maintained by the ASF be categorized using DBYCAT, which was trained through the analysis of ASF projects?* This question aims at estimating the generalization capability of DBYCAT.

Research Question 4 (RQ4): *Are the results of DBYCAT-1 (which was trained with bytecode documents) comparable to the results of DBYCAT-2 (which was trained with documents extracted form source code)?* This question aims at investigating if the difference between the results of both models are statistically significant.

To answer **RQ1** and **RQ2**, Study-1 used a 10-fold cross validation, in which the set of 158 ASF projects were divided in 10 random subsets. DBYCAT analyzed the subsets 10 times, each time using a different subset as a testing dataset and all the remaining subsets as a training dataset.

The results of **RQ3** were obtained using an survey involving 17 participants, among students, developers and researchers. The survey contained 15 questions, each one asking for the appropriateness of a set of categories suggested by DBYCAT to each of the 15 software projects not related with ASF.

The results of **RQ4** are obtained comparing the values of *precision@1*, *precision@5*, *recall@1*, *recall@5*, *MAP* y *RR* provided by DBYCAT-1 and DBYCAT-2.

4.1.2 Analysis Method

4.1.2.1 Research Question 1

For **RQ1**, Study-1 proposes a custom criteria to determine whether libraries are correctly categorized assuming the categorization is single-label; our proposed approach provides several labels as part of the DCP-based categorization. However, for RQ1 we are only interested in a single label categorization.

Definition 4.1.1. The model *categorizes a library correctly* if at least one of the suggested categories is equal to any of the categories assigned by the ASF developers. Thus, given $S_i = \{c_1, c_2, \ldots, c_5\}$ the set of suggested categories to the library i by the model and $P_i = \{p_1, p_2, \ldots, p_k\}$ the set of categories assigned to the library i by ASF developers, an example is correctly classified if $S \cap P \neq \emptyset$.

Definition 4.1.2. The *score* of a categorized library is defined as follows:

$$score_i(S_i, P_i) = \begin{cases} 1, & \text{if } S_i \cap P_i \neq \emptyset \\ 0, & \text{if } S_i \cap P_i = \emptyset \end{cases} \tag{4.1}$$

Definition 4.1.3. The *accuracy* for a categorization of the documents in the set L, with $\mid L \mid = n$:

$$accuracy = \frac{\sum_{i=1}^{n} score_i(S_i, P_i)}{n} \tag{4.2}$$

It should be noted that DBYCAT suggests the same number of categories for each library, i.e. $|S_i| = |S_j| \forall i \neq j$.

Whit this definition of *accuracy*, the metrics of training precision and testing precision for the results of the automatic categorization are proposed:

Definition 4.1.4. The *training accuracy* is the *accuracy* of the classifier at recommending categories to the libraries contained in the training dataset.

Definition 4.1.5. The *testing accuracy* the *accuracy* of the classifier at recommending categories to the the libraries in the testing dataset.

The *training accuracy* provides some insights about how the model is being trained. However, the accuracy of the recommendations given by DBYCAT will be measured using the *testing accuracy*, which reflects the capability of the model to categorize libraries not contained in the training dataset.

4.1.2.2 Research Question 2

To answer **RQ2**, Study-1 uses the standard approach to evaluate information retrieval systems based on *precision* and *recall*. This standard approach bases the evaluation on the notion of a *relevant* category, which is defined as a category predefined by ASF developer to the software libraries under analysis. The set of *relevant* categories suggested by the mentioned developers to a software project is called a *gold set*. Thus, the *precision* is defined as the fraction of the suggested categories that are relevant[25]:

$$precision = \frac{\#(relevant\ categories\ retrieved)}{\#(retrieved\ categories)} \tag{4.3}$$

And *recall* is the fraction of relevant categories that are retrieved[25]:

$$recall = \frac{\#(relevant\ categories\ retrieved)}{\#(relevant\ categories)} \tag{4.4}$$

DBYCAT assigns a set of categories to the new library *ranked* them by its relevance. Therefore, we were interested in measuring the accuracy of the recommendations considering the order in which those categories are given. Because the model describes each cluster with the top five relevant categories, each software project is categorized with the same set of five categories ordered by its relevance; therefore, for the multi-label case this study uses

$$precision@k = \frac{\#(relevant\ categories\ retrieved)}{k} \tag{4.5}$$

and

$$recall@k = \frac{\#(relevant\ categories\ retrieved)}{\#(relevant\ categories)} \tag{4.6}$$

Where k is the number of recommended categories ($k = 5$) and $\#(relevantcategoriesretrieved) \leq k$

It should be noted that the value of *precision@k* and *recall@k* are influenced by the number of retrieved results in a different manner. For example, if a software project has only one category assigned by its developers, and the model suggest a set of 5 categories containing the relevant one, the value of *precision@5* for this query would be $1/5 = 0.2$, whereas the value of *recall@5* would be 1. Another scenario, where a software project has 7 categories assigned by its developers, and the model suggests a set of 5 relevant categories, the value of *precision@5* would be $5/5 = 1$ but the value of *recall@5* would be $5/7 = 0.714$ so the model would have a high precision but it would be unable to retrieve all the relevant results.

The metrics aforementioned are appropriate to estimate the accuracy of the set of k categories, but these do not take into account the order in which these categories are given. For example, a set of suggested categories containing only a relevant category in the first position will be more appreciated than a set with the relevant category in the fifth position. In this sense, the *mean average precision* (MAP) provides a measure of quality among recall levels [25]. Given a set of queries Q and R_k the set of ranked and retrieved results, then

$$MAP(Q) = \frac{1}{|Q|} \sum_{j=1}^{|Q|} \frac{1}{m_j} \sum_{k=1}^{m_j} precision(R_{jk}) \tag{4.7}$$

We were interested in identifying which are the values of *precision@k* and *recall@k* if the value of k varies from 1 to 5, and the value of MAP for $k = 5$. The reasons to propose this analysis with the results limited to 5 are: (i) The distribution of projects per category in ASF dataset varies from 1 to 8, but only three projects were given 6,7 and 8 categories respectively. Additionally, the majority of the ASF projects were categorized using 1 or 2 categories; and (ii) if the majority of the projects only have 1 or 2 categories, MAP can measure the usefulness of the results taking into account the position of the relevant categories in a set of suggested categories. The characteristics of the ASF dataset will be shown in Section 4.2.1

For each step in the 10-cross validation process, the measurements of *precision@k*, *recall@k* and MAP are computed for each categorized software project in both training and testing datasets.

4.1.2.3 Research Question 3

This research question aims at investigating the capability of DBYCAT to categorize projects not related to the Apache Software Foundation, using the categories provided by the developers of ASF projects. To measure this capability, a dataset composed by 15 software libraries non maintained by ASF was analyzed and categorized using DBYCAT, i.e., DBYCAT suggested a set of 5 categories extracted from ASF dataset to each non-ASF software project. Then, a survey with 17 developers was conducted, where the participants were asked to evaluate the correctness of each suggested category and their opinions were collected using a 5-item Likert scale. Finally, the reasoning behind their evaluation was collected through an open question.

In this case there is no *gold set*, it means, a set of predefined categories for each non-ASF project is not available. Therefore, Study-1 uses the opinions given by the participants to identify the set of relevant categories for each non-ASF project. In this sense, Study-1 is interested in assessing the accuracy of DBYCAT when it categorizes the projects with a single tag (single-label) and when it categories the libraries with a set of relevant tags (multi-label).

For the single-label case, we were interested in estimating whether *any* of the recommended libraries is relevant for each non-ASF software project. Since a 5-item Likert scale was used to collect the opinion of the participants of the survey about the relevance of a suggested category for a particular library, we need a simplified criteria to evaluate these opinions. The construction of this simplified criteria begins with the computation of the

mode of the answers of each question, then if the *mode* of the answers of a particular question was composed by several Likert items, this study penalized this uncertainty taking the item with the least relevance (e.g. if the *mode* of a particular question was *Strongly agree* and *Strongly disagree*, this study only took into account the item *Strongly disagree* as the *mode*). Because the values of these *modes* can be any of the 5 items of the Likert scale, we want to simplify the analysis transforming the *modes* into two new items: *Agree* and *Disagree*. Therefore, the following transformation is proposed: $Strongly\ agree, Agree \rightarrow Agree; Strongly\ disagree, Disagree, Neither\ agree\ or\ disagree \rightarrow Disagree$. Given this transformation, an example correctly classified is defined as follows:

Definition 4.1.6. The model *categorizes an example correctly* if most of the participants of the survey (mode) *agree* with the appropriateness of at least one of the suggested categories.

Taking into account that DBYCAT suggests an ordered set of categories to the projects in the non-ASF dataset, this study is interested in identifying the position in which the first relevant category is offered. The *Mean Reciprocal Rank (MMR)*[4] shows whether the model can provide a relevant category at the top of the suggested set of categories. To define the MRR, it is needed to define first the Reciprocal Rank score (RR); given a set of queries $Q = q_1, q_2, \cdots, q_n$, the RR for a query j is defined as follows:

$$RR_j = \frac{1}{k} \tag{4.8}$$

Where k is the position of the first relevant results in the query j. Thus, the MRR is defined as follows:

$$MRR = \frac{\sum_1^j RR_j}{|Q|} \tag{4.9}$$

In other words, MRR is the average of the RR values across multiple categorizations.

On the other hand, to measure the multi-label performance, this study uses once again the standard criteria for information retrieval problems. However, the dataset of non-ASF projects does not contain predefined categories, therefore *recall@k* can not be figured. In this sense, let p be the number of relevant categories retrieved by DBYCAT and k the number of retrieved (i.e. provided by the model) categories (k can contain relevant and irrelevant categories), the *precision@k* is defined as follows:

$$precision@k = \frac{p}{k} \tag{4.10}$$

We were interested in measuring the accuracy of DBYCAT when it recommends $k = 1, 2, 3, 4, 5$ categories to each software project, in order to know for which k the model can recommend the greatest amount of relevant categories in the top k results.

Finally, since Likert scale offers a non-binary notion of relevance of each suggested category, this study intends to measure the usefulness of the ranked results, that is to say, whether DBYCAT can recommend the most relevant categories at the top of the list of suggested categories. The Discounted Cumulative Gain (DCG)[25] uses *graded* relevance as a measure of *gain*. However, to summarize the ranking, it means, to compare the usefulness or *gain* of the suggested order of categories with the *gain* of the expected order of the categories, this study uses the Normalized Cumulative Gain ($NDCG$). The reasoning

behind this metric is that the ideal ranking would return the most relevant categories at the beginning of the list, it means, reflecting the highest level of relevance.

The Discounted Cumulative Gain (DCG) for a recommendation j is defined as follows:

$$DCG_j = rel_1 + \sum_{i=2}^{p} \frac{rel_i}{log_2(i)} \tag{4.11}$$

Where p is the number of retrieved categories. The Ideal DCG ($ICDG$) for a recommendation j can be defined as the CDG for the ideal ranking in which the recommendation j must provide the categories. Since the purpose of $NCDG$ is to compare the real ranking of a recommendation with the ideal ranking of the recommendation, it is defined as follows:

$$NDCG_j = \frac{DCG_j}{IDCG_j} \tag{4.12}$$

To estimate the ideal ranking of a recommendation, Study-1 uses the answers provided by the participants of the survey using the 5-item Likert scale as a non-binary measurement of relevance. In this case, the 5-item Likert scale is represented as numbers as follows:

- $Strongly\ disagree \rightarrow 0$
- $Disagree \rightarrow 1$
- $Neither\ agree\ or\ disagree \rightarrow 2$
- $Agree \rightarrow 3$
- $Strongly\ agree \rightarrow 4$

For example, the participants of the study expressed their opinions about the relevance of the suggested categories for the library *eclipselink-2.5.1* as follows (it should be noted that these values correspond to the *modes* of the answers of the participants):

- $d_1 : library = Agree$
- $d_2 : web - framework = Disagree$
- $d_3 : network - server = Disagree$
- $d_4 : database = Strongly\ agree$
- $d_5 : xml = Agree$

With the proposed transformation, the numerical values for the ranking $\{d_1, d_2, d_3, d_4, d_5\}$ would be $\{3, 1, 1, 4, 3\}$. The expected ranking is obtained with the decreasing order of this values, i.e. $\{4, 3, 3, 1, 1\}$, therefore an expected order of the results (ranking) could be $\{d_4, d_1, d_5, d_2, d_3\}$. For this recommendation, the value of DCG, i.e the value for the Discounting Cumulative Gain for the ranking $\{d_1, d_2, d_3, d_4, d_5\}$ is 7.923, whereas the value of $IDCG$, i.e. the value for the expected order $\{d_4, d_1, d_5, d_2, d_3\}$ is 9.823. At the end, the value of $NDCG$ for this example is $7.923/9.823 = 0.8065$, indicating that the real ranking has the 80% of the usefulness of the expected ranking.

	Class names	Method names	Fields	Local variables	String constants	Calls to methods	Comments
Bytecode 1 (Original dataset)	X	X	X	X			
Bytecode 2	X	X	X	X	X	X	
Source code	X	X	X	X	X	X	X

TABLE 4.1. Composition of the different datasets used to answer **RQ4**

4.1.2.4 Research Question 4

To answer **RQ4**, we want to establish if the accuracy of DBYCAT trained with the bytecode documents is comparable with the accuracy of the same model but trained with the documents extracted from source code. Therefore, several experiments with DBYCAT involving different training datasets are conducted, and the metrics provided by each experiment are stored in order to be compared with each other. Since each experiment provides a series of different results, we want to estimate the *statistical difference* between these series of observations;. In this sense, the following *statistical hypothesis* are proposed:

Null Hypothesis 1 (NH1): *The probability distribution of each series of observations corresponds to a normal distribution.*

Alternative Hypothesis 1 (AH1): *The probability distribution of each series of observations corresponds to an arbitrary distribution.*

Null Hypothesis 2 (NH2): *The probability distribution of every series of observation is the same .*

Alternative Hypothesis 1 (AH2): *The probability distribution of every series of observation vary in at least one element.*

According to the description of the data extraction process performed by DBYCAT (Section 3.1), it originally extracts from bytecode a particular set of source code identifiers (classes, methods, fields, local variables). So this experiment additionally wants to extend the results to the same model but trained with two different dataset: the first one, a different set of bytecode documents which includes string literals (those string constants that developer uses to show some messages to the user) and calls to methods, and the set of source code documents. Table 4.1 summarizes the differences among the datasets *Bytecode 1* (the original dataset), *Bytecode 2* and *Source code*.

Some validation metrics for the multi-label categorization scenario (*precision@1*, *recall@1*, *precision@5*, *recall@5* and *MAP*), and the *reciprocal rank* were obtained by running the experiment (i.e. the 10-fold cross validation) several times using a different dataset (i.e. bytecode1, bytecode 2 and source code) each time. Then, to evaluate **NH1**, i.e. to verify if each series of metrics are distributed according with a normal distribution, this study uses the *Shapiro-Wilk*[36] and the *Kolmogorov-Smirnov*[26] tests. Because the results of these tests suggest that the measures (i.e. the values of *precision@k*, *recall@k*, *MAP* and *reciprocal rank*) are not normally distributed (as described in Section 4.3.4), this study uses

the Kruskall-Wallis test [18] to verify if the series of measures are statistically different. The statistical tests are applied using a confidence level of 95%, i.e. $\alpha = 0.05$.

4.2 Data Extraction Process

The data needed to answer the research questions are (i) the bytecode of the 158 ASF projects, (2) the source code of the 158 ASF projects, (iii) the software profiles of those 158 projects, (iv) the categories provided by ASF developers, (v) the bytecode of the 15 software projects not related with ASF, and (vi) the answers of the survey conducted with 17 software developers.

4.2.1 Apache Software Foundation dataset (ASF dataset)

The ASF dataset is composed by 158 software projects, each one having the following artifacts:

JAR files The thesis proponent visited all the 158 main web pages belonging to the software projects in order to download the deployable artifacts that each project offers. The ASF website contains more than 158 projects, but some of them were discarded because they are not written in Java (e.g. Apache http server) or they did not provide .class files. In some cases, the web page provided the source code of a project, but the building process was not trivial, thus the project was discarded.

Categories Each software project is tagged with at least one predefined category. We collected the categories assigned to each project in order to built a labeled dataset, which was used to train DBYCAT. The distribution of projects per category is shown in Figure 4.1 and the distribution of assigned categories per project is shown in Figure 4.2.

Since ASF developers can assign more than one category to each project, the sets of projects per category shown in Figure 4.1 are not disjoint. Additionally, Figure 4.1 shows an evident bias in the distribution of projects per category, particularly, 44.9% of projects in ASF dataset were categorized by ASF developers as *library*, and around 44% (53) of projects that have only one category (120 projects according with Figure 4.2) were tagged as *library*.

Profiles The author compiled the profile of each project, reading and extracting from the web page the chunks of text related with the main functionality and features of the software projects. The extraction process was done by hand.

Source code The source code of the 158 software projects was collected manually.

It should be noted that the source code of the projects is only used to answer **RQ4**, as described in Section 4.1.2.4; therefore the training phase of the DBYCAT uses the *JAR* files, *categories* and *profiles* of the ASF dataset. Finally, an example of the information provided by the ASF web page about its software libraries is shown in Figure 4.3.

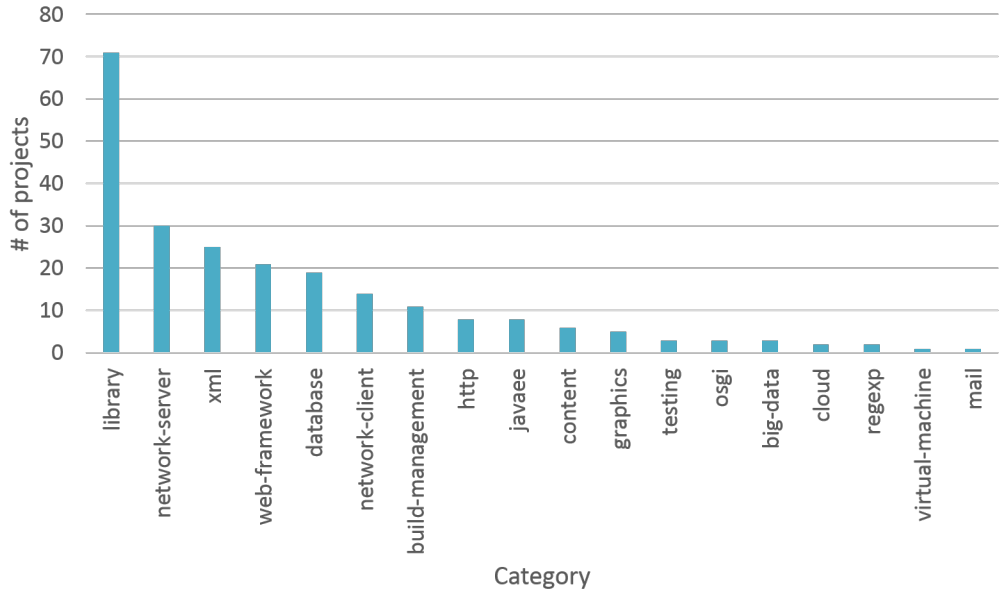


FIGURE 4.1. Distribution of projects per category in the ASF dataset

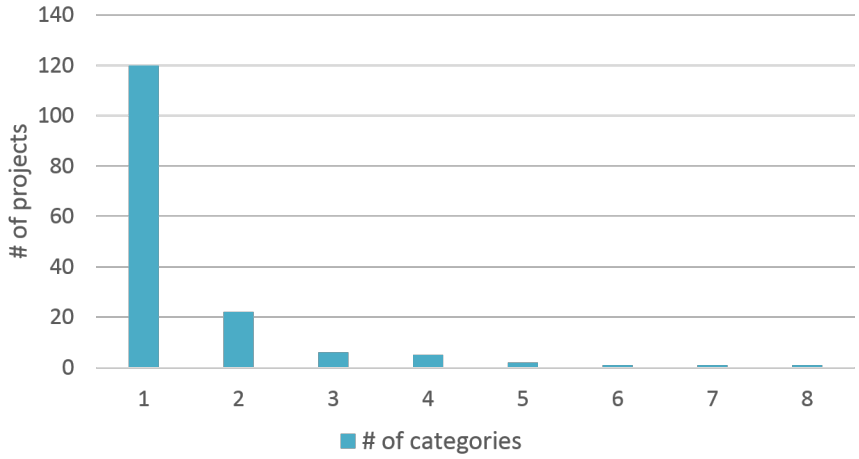


FIGURE 4.2. Distribution of categories per project in the ASF dataset

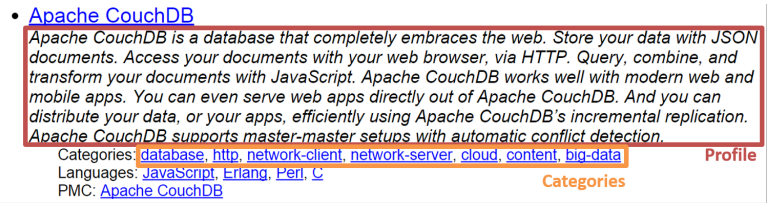


FIGURE 4.3. Example of the information provided by the ASF web page about a software library

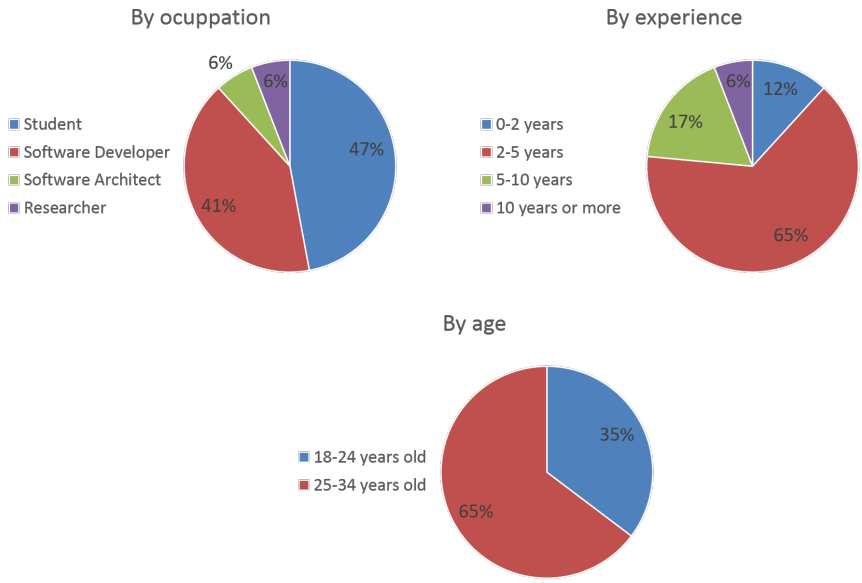


FIGURE 4.4. Distribution of the participants of the survey according to a) their occupation, b) their experience, and c) their age

4.2.2 Survey

The survey was composed by 15 questions about the correctness and usefulness of the categories suggested by DBYCAT for 15 software projects randomly chosen. Table 4.2 presents a brief description of these projects.

We categorized each project of this testing set using DBYCAT, extracting the bytecode and computing the weighted vectors using $tf - idf$. Then, the cosine similarity between each vector and the *centroids* obtained in the best iteration of the 10-fold cross validation process performed for **RQ1** and **RQ2** was computed – the best iteration was chosen taking the iteration with the highest value of *accuracy* (see Section 4.2). Then, the categories assigned to the closest cluster becomes the new categories of the new testing project. For each project, the survey asked to read the profile of the project in order to know a little more about it, then it asked about the correctness of each one of the suggested categories.

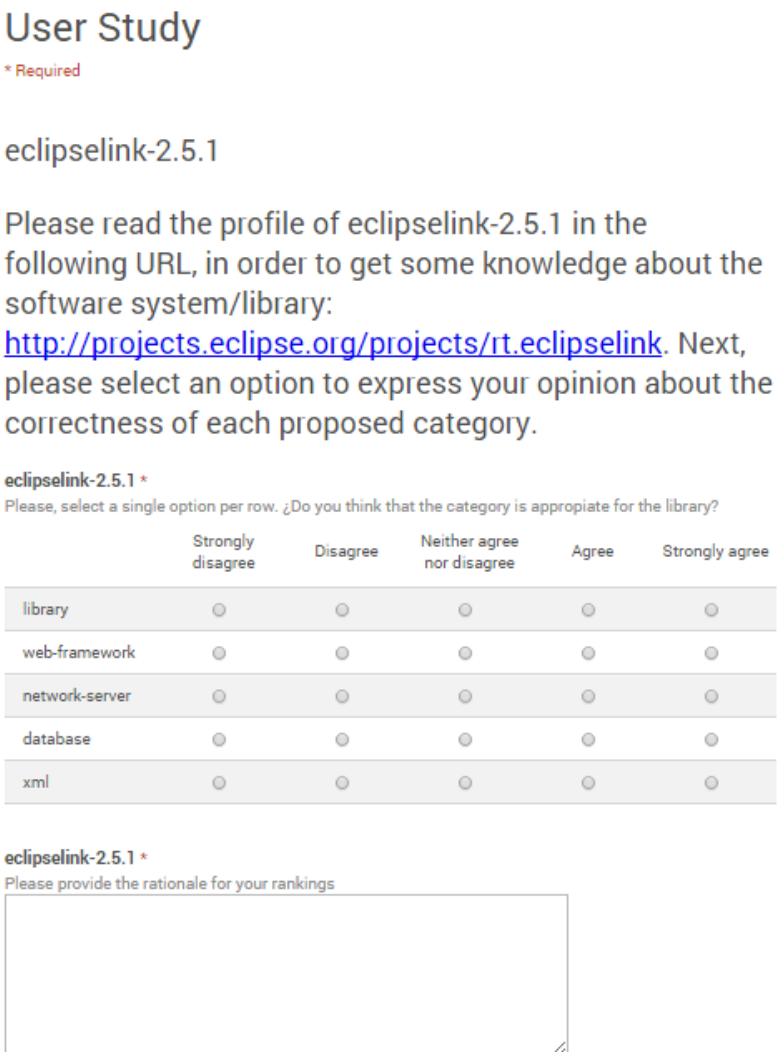
The survey, composed by 15 question, was responded by 17 software developers and students with experience in software development, software testing, software architecture, software project management or research in software engineering. Figure 4.4 shows an overview of the participants of the survey.

Each question in the survey aimed at knowing the opinion of each participant about the correctness and usefulness of 5 categories suggested by DBYCAT, to the projects described in Table 4.2; this opinion was measured using a 5-item Likert scale. Additionally, each question had an open item in which each participant could express the rationale for her selection or opinion. An example of a question used in the experiment is shown in Figure 4.5

The results of the metrics for each research question (i.e. *accuracy*, *precision*@5, *recall*@5, MAP, MMR and $NDCG$) are analyzed through boxplots.

Project	Description	URL
Eclipselink	ORM library and database connector	http://projects.eclipse.org/projects/rt.eclipselink
Sweet Home 3D	Free interior design application	http://www.sweethome3d.com/features.jsp
Hibernate	ORM library and database connector	http://hibernate.org/orm/
UIMA	Analyze large volumes of unstructured information in order to discover knowledge that is relevant to an end user.	http://uima.apache.org/
Artifactory	Provides out-of-the-box all that is needed to set up and run a robust secured repository within your organization	http://www.jfrog.com/home/v_artifactory_opensource_features
ZK	Open source Java framework for building enterprise web and mobile apps.	http://www.zkoss.org/product/zk
Web-Harvest	Open Source Web Data Extraction tool written in Java	http://web-harvest.sourceforge.net/
Jsoup	Java library for working with real-world HTML	http://jsoup.org/
HSQLDB	HSQLDB (HyperSQL DataBase) is a SQL relational database software written in Java	http://hsqldb.org/
JUnit	A programmer-oriented testing framework for Java	http://junit.org/
Dom4j	Open source library for working with XML, XPath and XSLT on the Java platform	http://dom4j.sourceforge.net/
Slf4j	The Simple Logging Facade for Java (SLF4J) serves as a simple facade or abstraction for various logging frameworks	http://www.slf4j.org/
JHotDraw	JHotDraw is a Java GUI framework for technical and structured Graphics.	http://www.jhotdraw.org/
Liquidbase	Revision control system for databases	http://www.liquibase.org/
XMLUnit	XML can be used for deciding if two documents are equal to each other	http://xmlunit.sourceforge.net/

TABLE 4.2. Java projects used in the survey



User Study

* Required

eclipselink-2.5.1

Please read the profile of eclipselink-2.5.1 in the following URL, in order to get some knowledge about the software system/library: http://projects.eclipse.org/projects/rt.eclipselink. Next, please select an option to express your opinion about the correctness of each proposed category.

eclipselink-2.5.1 *

Please, select a single option per row. ¿Do you think that the category is appropiate for the library?

	Strongly disagree	Disagree	Neither agree nor disagree	Agree	Strongly agree
library					
web-framework					
network-server					
database					
xml					

eclipselink-2.5.1 *

Please provide the rationale for your rankings

FIGURE 4.5. Example of a question in the survey used as instrument to validate the correctness of the suggested categories.

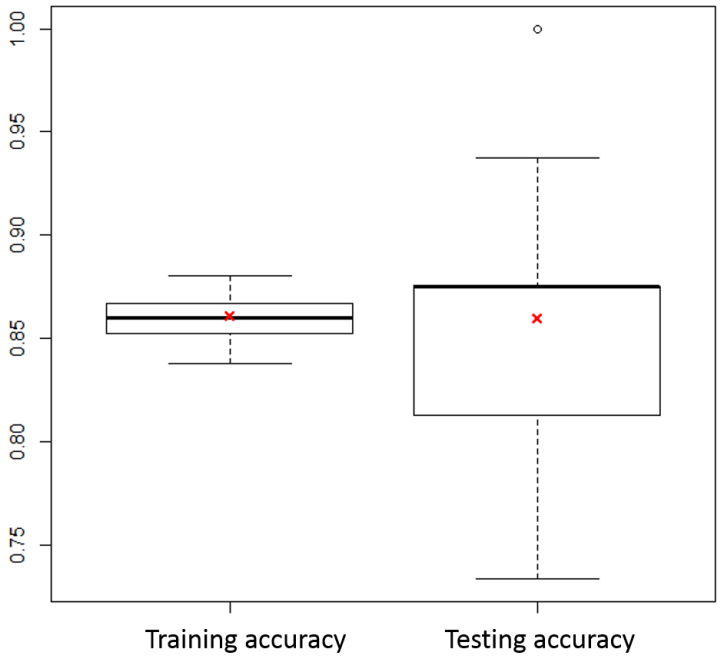


FIGURE 4.6. Training accuracy and testing accuracy values obtained for the 10 steps in the cross-validation process

4.3 Results

4.3.1 Single label (RQ1)

As discussed in section 4.1.2.1, the accuracy of DBYCAT according to **RQ1** is evaluated measuring the precision of a classifier that uses the results of the clustering model; therefore, a custom precision measure is proposed considering the nature of the dataset under analysis (see Equation 4.1). The accuracy values obtained for each iteration of the cross-validation process for both training (training precision) and testing (testing precision) datasets are shown in Figure 4.6.

The average accuracy on training dataset was 0.860, indicating that DBYCAT can extract groups of similar software projects and propose suitable vectors of relevant terms from each group which can greatly recreate the assignment of categories previously made by ASF developers. Likewise, the value of accuracy ranged from 0.83803 to 0.88028, suggesting that DBYCAT offer consistent results using different training sets.

On the other hand, the average accuracy on testing dataset took a similar place, with a value of 0.859. This result denotes an important capability of DBYCAT to categorize new software projects with a single label using just the relevant terms extracted from each cluster. Nevertheless, the value of accuracy ranged from 0.73333 to 1.00000, suggesting that the classifier is sensitive to the nature of the training dataset.

With this last result (testing accuracy) the **Research Question 1** can be answered. We conclude that **the accuracy of DBYCAT at recommending relevant categories to software libraries of ASF not contained in the training dataset was on average 0.859, i.e., in 8 out of 10 cases DBYCAT is appropriate to categorize applications.**

4.3.2 Multi-label results (RQ2)

The set of categories suggested by DBYCAT can contain more than one appropriate category for a software project. As aforementioned on section 4.1.2.2, the effectiveness of this kind of categorization is measured using *precision@k* and *recall@k*. The values for *precision@k* and *recall@k* for $k = 1, 2, 3, 4, 5$ obtained after the 10-fold validation process using the training datasets are shown in Figure 4.7; likewise, the same values but using the testing datasets are show in Figure 4.8, and the averages of this values are shown in Figure 4.9; in these figures, the reader can perceive how the values of *precision@k* and *recall@k* changes whereas the value of k increases. Additionally, some examples of categorized libraries belonging to the ASF dataset are shown in Table 4.3; these examples were taken from the testing datasets of the 10-fold cross validation process.

For both sets (training and testing) the values of *precision@k* by themselves shows that DBYCAT never reaches an high level of precision, due to it greatest value (near to 0.4 for $k = 1$) dramatically decrease as k (i.e. the number of categories suggested by the model) increases. This reduction in the *precision@k* values is stabilized in $k = 5$, proving that given the set of 5 suggested categories, at least 1 is relevant and can be assigned to the project. This conclusion is consequent with the results obtained in **RQ1**.

However, analyzing together the values of *recall@k* and *precision@k*, the results show that for $k = 4$, DBYCAT already have recommended most of the appropriate categories for each software project, as shown by the values of *recall@4* which are close to 0.8. Thus, despite the low values of *precision@4* which are close to 0.2, DBYCAT can suggests all the relevant categories. These results are closely related with the nature of the ASF dataset; as shown in Section 4.2.1, the distribution of the number of categories per project showed that most of the projects were tagged with only one category, thus the values of *precision@4* and *recall@4* are biased by the software projects with only one relevant category. Thereby, the performance of DBYCAT can not be properly estimated using the libraries that were categorized with more than one tag. To solve this uncertainty, this study uses MAP because it provides a measure of quality across recall levels, it means, this metric is insensitive to the bias provided by the dataset. The values for MAP for each step of the 10-fold cross validation process for both training and testing dataset are shown in Figure 4.10.

Considering the nature of the ASF dataset, a $MAP = 1$ would indicate that DBYCAT is recommending all the relevant categories at the top of the suggested set for every software project being categorized; hence the values obtained for the training ($MAP = 0.532$) and testing ($MAP = 0.502$) dataset show that, in general, DBYCAT always recommend a suitable number of categories, but those are not shown at the top of the suggested set.

Given these results, **Research Question 2** can be answered. We conclude that **the accuracy of DBYCAT at recommending several categories for a ASF software project not contained in the training dataset is acceptable. The values of *precision@1* indicates that in four out of ten cases the first recommended category is a correct one**; however, a random suggestion using the categories of ASF would lead to

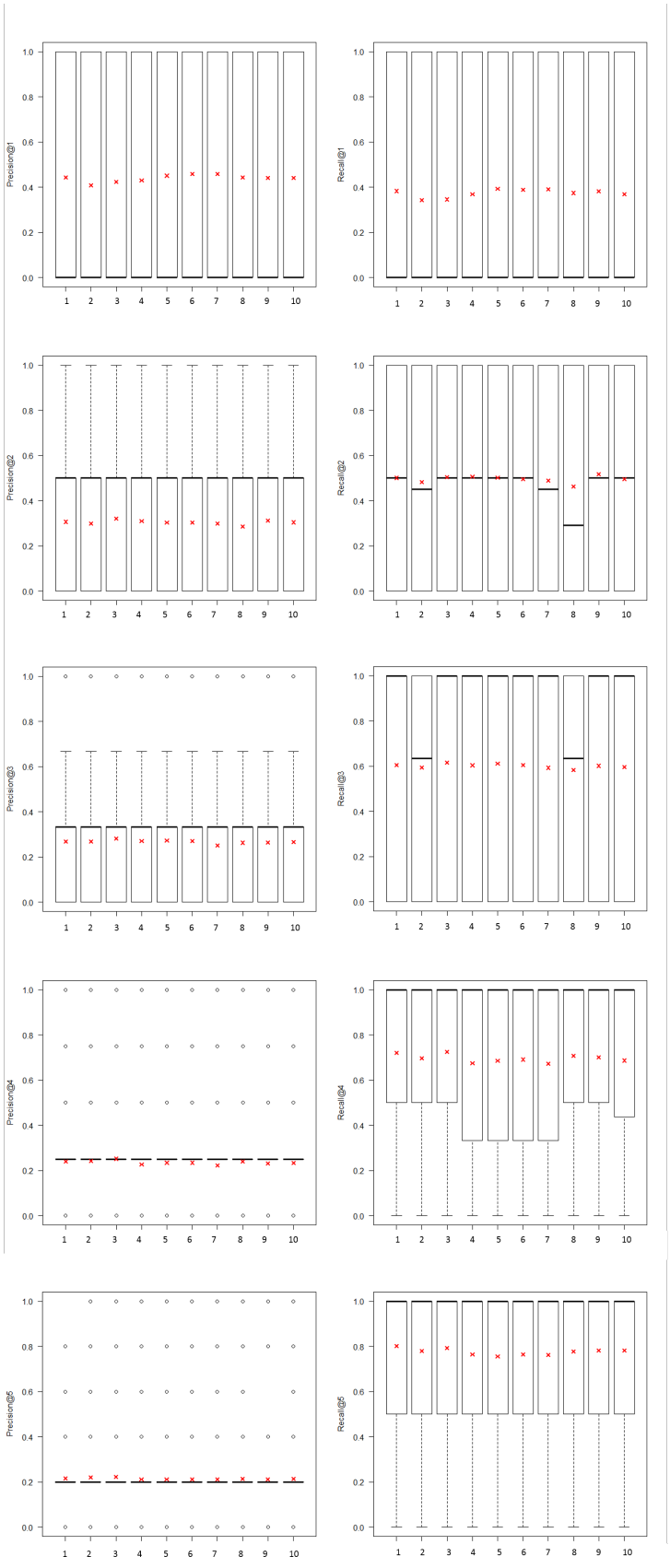


FIGURE 4.7. Precision@k and recall@k values for k=1,2,3,4,5 in each 10-fold validation step using the training dataset. The x-axis shows the number of each step in the 10-fold cross validation process

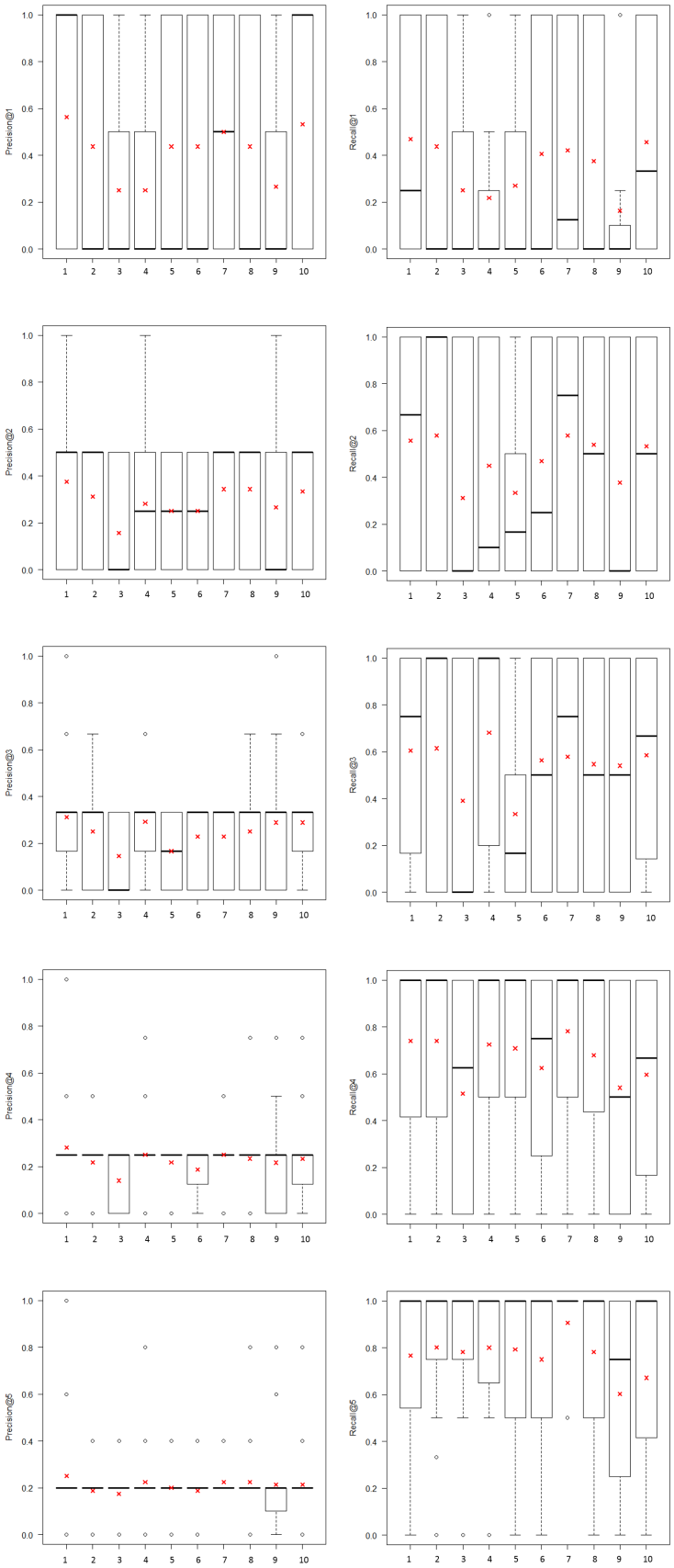


FIGURE 4.8. Precision@k and recall@k values for k=1,2,3,4,5 in each 10-fold validation step using the testing dataset.The x-axis shows the number of each step in the 10-fold cross validation process

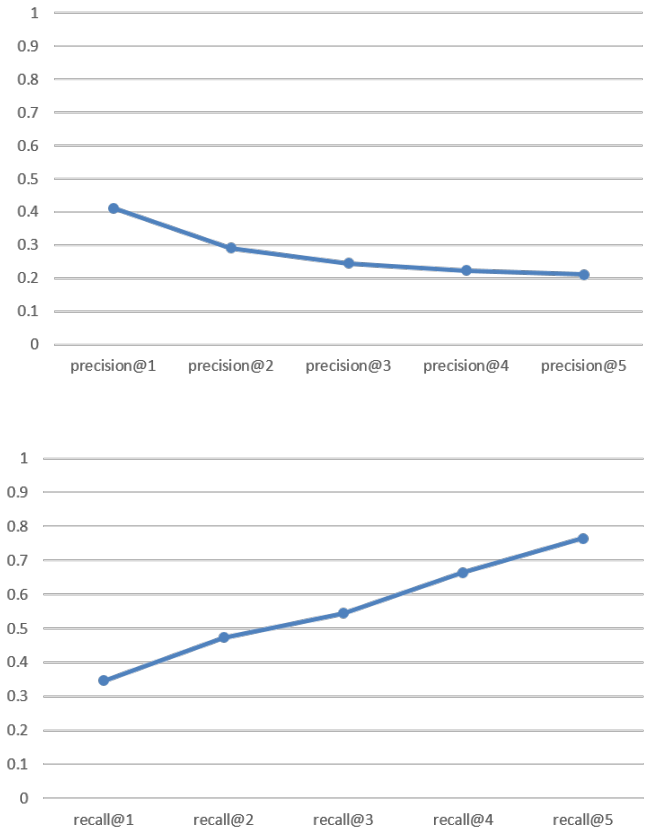


FIGURE 4.9. Average values of Precision@k and recall@k for k=1,2,3,4,5

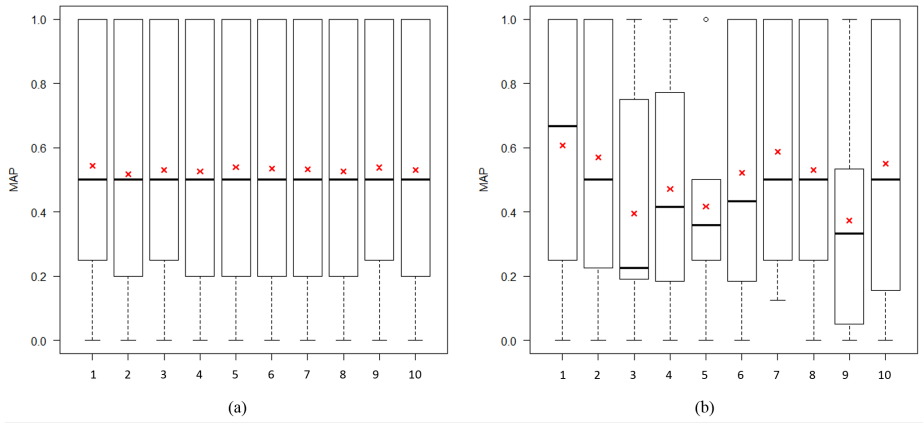


FIGURE 4.10. Mean average precision for each step in the 10-fold cross validation process for (a) the training datasets, and (b) the testing datasets

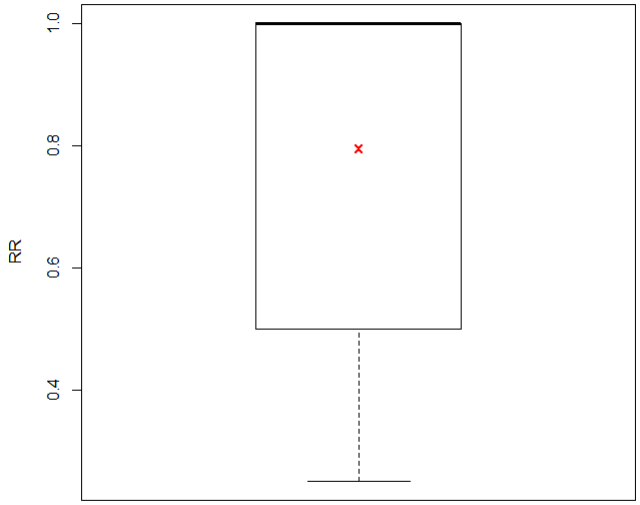


FIGURE 4.11. Reciprocal Rank and Mean Reciprocal Rank values for the single-label categorization of non-ASF projects

a value of $1/18 = 0.055$ so the accuracy of DBYCAT at recommending only one category is not underestimated. **Additionally, the values of precision@5** (0.210) **and recall@5** (0.765) **show that the set of 5 categories recommended by DBYCAT contains at least 1 relevant category, result that reinforces the results obtained for RQ1. Analyzing together the values of *recall@5* and *precision@5*, the results look remarkable, showing that in fact, despite the seemingly low value for the precision at $k = 5$, DBYCAT in fact recommends 76% of the correct categories for each library in the top five recommendations. Finally, the value of *MAP*** (0.502) **show that DBYCAT does not show the relevant categories at the top of the suggested set.**

4.3.3 Generalization (RQ3)

For the categorization with a single label, the answers of the survey were transformed into *Agree* and *Disagree* values, as shown in Section 4.1.2.3. According to the answers of the participants, the set of categories suggested by DBYCAT for every project in the dataset conformed by 15 non-ASF projects contains at least one appropriate category for that project. In this sense, the trained model is capable of recommending a single category for a new software project through the analysis of the bytecode of existing software libraries. Regarding the position of the relevant category in the set of recommendations, the values of the *Reciprocal Rank* (*RR*) varies from 0.25 to 1, with a *Mean Reciprocal Rank* of 0.794. This result shows that a model trained to recommend categories provided by ASF can recommend relevant tags within the first 2 positions of the recommended set. Figure 4.11 shows the values of *RR* and the value of *MRR* obtained for RQ3.

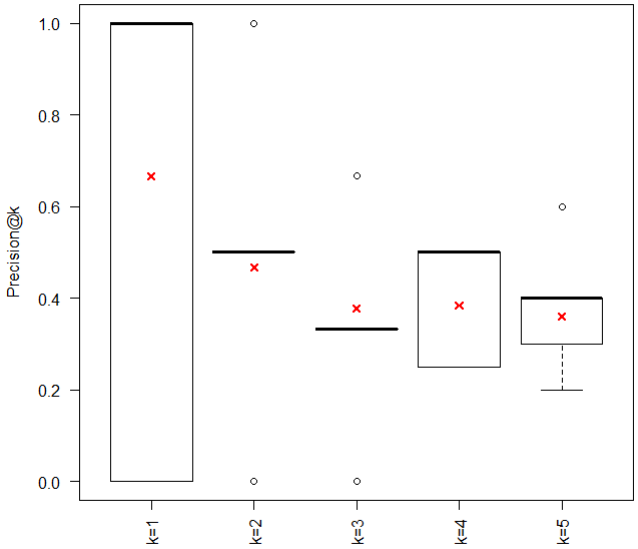


FIGURE 4.12. Values of *precision@k* for the multi-label clasification of non-ASF software projects.

The results of *precision@k* with $k = 1, 2, 3, 4, 5$ for the multi-label categorization are shown in Figure 4.12. The average of the values for *precision@5* suggests that, in general, DBYCAT can recommend 2 relevant ASF categories to each non-ASF software project. This conclusion proves an important capability of the model to generalize its recommendations, that is to say, although the model is trained with an small number of labeled samples and can suggest an limited number of categories, the recommendations managed to describe the purpose or main functionality of some non-ASF software projects.

Regarding the order of the categories, the values of *NDCG* suggest that the usefulness of the recommendations is close to the expected one. The average value of *NDCG* was 0.84 varying from 0.64 to 1, this means that in general, the real ranking has an 84% of the usefulness of the expected ranking. The values obtained of *NDCG* for all the recommendations are presented in Figure 4.13.

4.3.3.1 Analysis of the open questions

The purpose of the open section in each question of the survey was to understand the reasoning behind the opinions of the participants. It was found that some participants could identify the main purpose using the web profiles of the applications, and then they could find a succinct category to describe that purpose; for example, this is one of the answers to the open question related with the main purpose of *dom4j-1.6.1* (the suggested categories were intentionally highlighted in bold): *"Based on the short description of the software on the link, all of the categories* ***library***, ***web-framework*** *and* ***xml*** *are very relevant in identifying the purpose or details of dom4j. I agree that* ***network-server*** *is mildly relevant as a target for the software. I don't think that* ***database*** *is a particularly helpful categorization given the purpose of the application."*. Moreover, in some cases most of the suggested categories did not match the purpose of the software project, it means, only one

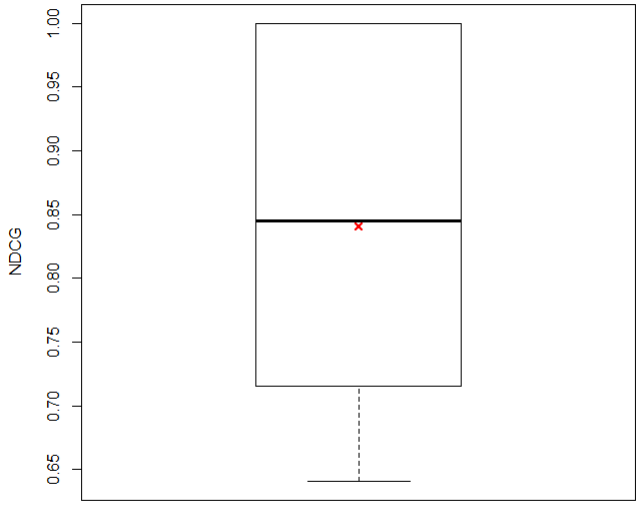


FIGURE 4.13. Values of *Normalized Discountive Cumultive Gain* for the ranked results of the multi-label classification of non-ASF software projects.

of them works as a relevant category for the library; an example of this case, where only one recommended category was appropriate for the project *SweetHome3D-4.3* is presented as follows: *"The SwetHome3D application, while extendable does not appear to be a **library** for developmental purposes as much as a customizable application. I see no evidence of this being a **web-based** technology, as it appears to be downloaded and executed on a local machine, although it appears to have downloadable content. I agree that **graphics** play a large part in the application as it primarily seems to be visualization based, but I disagree that **xml** and **network-server** are valid categories, as neither seem to really relate to the purpose of the application itself or any overt implementation details specific to the application."*; in this case *SweetHome3D-4.3* is a *"free interior design application"*, i.e. an application to draw and design indoor environment. Therefore, the only relevant category in set of categories provided by ASF was *graphics*, which was correctly recommended by DBYCAT. Finally, there was a case where any of the suggested categories represented the purpose of the library, so in this case the participant decided to choose a generic category as an appropriate tag; the following comment, which was given for the library *junit-4.11*, expresses this scenario: *"I don't think that any of the available categories are exceptional at identifying the purpose of this piece of software. I agree that it is a **library** that assists with unit test creation, but have insufficient exposure or information to know whether that is in the form of a **web-framework** or not. I disagree that **xml** and **database** are relevant because they don't really apply to junit at all from what I see."*; this case DBYCAT recommended a set of irrelevant categories to the library *junit-4.11*; the set of categories provided by ASF contains a relevant category *testing* which was not recommended by DBYCAT.

To answer *Research Question 3*, The results indicated that **the survey participants agreed that at least one of the five suggested categories was relevant for each of the 15 non-ASF projects used, resulting in an *accuracy* of** 1.0**, by using our accuracy estimation in Equation 4.2. The value of *MMR* for this part of the**

study was 0.794, indicating that according to the survey participants, DBYCAT recommended the first relevant category within the top 2 positions.

4.3.4 Comparison (RQ4)

The results for the statistical test to evaluate **NH1** (Kolmogorov-Smirnov and Shapiro-Wilk tests) are shown in Table 4.4, and the values for the Kruskall-Wallis test are shown in Table 4.5.

According to Table 4.4, the values of *p-value* are lesser than 0.05 (the confidence level), therefore **NH1** is rejected and it can be concluded that the probability distribution of the observations (measures) is an arbitrary distribution (**AH1**). For this reason, the Kruskall-Wallis test is used to estimate whether the results are statistically different because unlike ANOVA, Kruskall-Wallis does not need Gaussians distributions. Finally, the *p-values* in the Kruskall-Wallis test are greater than 0.05, thus **NH2** can not be rejected and it can be concluded that the measures are not significantly different, that is to say, **the bytecode is a good option when source code is not available, which answers our Research Question 4**.

4.4 Threats to validity

The validity of the results and its generalization are affected by some internal and external threats. An internal threat to validity is the nature of the code entities extracted from bytecode. DBYCAT uses a subset of the code entities that can be extracted from the bytecode; particularly, DBYCAT does not take into account the calls to methods and the string constants present in the bytecode. However, we showed that the accuracy of DBYCAT is comparable with the accuracy of similar models but trained with all code entities extracted from bytecode and source code.

Another internal threat to validity is the nature of the categories uses to train DBYCAT. The Apache Software Foundation dataset does not contain all possible application domains and the small number of categories (18) suggests that many of them are not specialized. Then, this set of categories cannot represent every application domain in the Java ecosystem; additionally, the training dataset contains many software libraries previously categorized with a popular category (*library*), which impact the capability of DBYCAT to assign specialized labels to the clusters. To minimize this threat, we used a user study to estimate the generalization capability of DBYCAT using the categories provided by ASF.

A third internal threat to validity is the nature of Dirichlet Process Clustering. This clustering algorithm does not need the number of clusters as parameters; instead, it needs the *concentration parameter* (α) as a parameter. Therefore, choosing the right value of α to obtain an optimal number of clusters becomes a non-trivial optimization problem. Additionally, since DPC models each cluster with a particular probability distribution assigning to each one a mixing proportion π, the results of this algorithm depend of the values of π which can be different if the algorithm is executed several times.

A fourth internal threat to validity is the use of libraries written in only one programing language (Java). Therefore, the results might not be generalized to other software libraries written programming languages such as C++, Python, Ruby, Groovy, etc. Nevertheless, this study was focused in the exploration of the bytecode as the source of relevant vocab-

ulary to categorize software applications, but it is not restricted to the use of bytecode of Java libraries. Future work can leverage any artifact with source code identifiers (e.g. source code files for C++ and Ruby, bytecode for Groovy) to categorize applications using DPC and automatic labeling using software profiles.

One external threat to validity is the repository chosen to train DBYCAT. The Apache Software Foundation dataset does not contain a comprehensive amount of libraries with different application domains, thus we can not guarantee that DBYCAT can automatically categorize appropriately every application in the Java ecosystem.

Finally, another external threat to validity is related to the categories that developers assigned to the ASF software libraries. Since the set of available categories in the ASF repository is not fully comprehensive, developers could assign to software libraries the closest categories even if these do not represent properly the application domain or main functionality of the library.

4.5 Lessons learned

Reviewing the categories provided by ASF developers, which were shown in Section 4.2.2, it can be said that those categories appropriately describe libraries and applications mainly used by programers and developers, it means, people with requirements closely related with software development or computer science. For this reason, a model trained with this dataset would not be able to appropriately categorize libraries and applications containing other applications domains. For example, any of the ASF categories can provide a succinct explanation of a library or application whose main functionality is related with audio, media, etc.

Moreover, the Dirichlet Process Clustering (DPC) offers a non-parametric approach for flat clustering; since its purpose is to group the elements in disjunctive sets, or seen from another perspective, *DPC* assigns each element to a single cluster, this approach is suitable for software categorization with a single label but not to categorize with multiples labels. DBYCAT addressed this issue describing each cluster with several categories, but the results for **RQ2** show that the set of categories describing each cluster in general only contains a single relevant label for the libraries contained in it. Future work will try to improve this aspect.

Library	Recommended categories	Real categories
cayenne-3.0.2	library, web-framework, network-server, xml, database	database, library, network-client, network-server, web-framework, xml
apache-forrest-0.9	library, web-framework, network-server, xml, database	build-management, database, graphics, http, network-client, network-server, web-framework, xml
cocoon-2.2.0	library, web-framework, network-server, xml, database	database, graphics, http, network-client, network-server, web-framework, xml
click-2.3.0	library, web-framework, network-server, database, xml	library, network-client, network-server, web-framework, xml
apache-ofbiz-12.04.02	library, web-framework, network-server, database, xml	database, http, network-server, web-framework, xml
apache-cxf-3.0.0	library, web-framework, network-server, database, xml	library, network-client, network-server, xml
jackrabbit-2.6.5	web-framework, library, xml, content, network-server	database, library, network-server, xml
geronimo-framework-3.0.1	library, web-framework, network-server, xml, database	http, javaee, network-server, web-framework
axis2-1.6.2	library, web-framework, network-server, database, xml	http, network-client, network-server, xml
httpcomponents-core-4.3.2	library, web-framework, database, network-server, xml	http, library, network-client, network-server
apache-etch-1.3.0	library, web-framework, network-server, database, xml	library, network-client, network-server
apache-openjpa-2.2.2	library, web-framework, network-server, xml, database	database, javaee, library
shiro-all-1.2.3	web-framework, library, network-server, xml, database	library, web-framework
apache-ode-war-1.3.6	library, web-framework, network-server, database, xml	network-server, xml
apache-servicemix-5.0.0	library, web-framework, network-server, database, xml	network-server, xml
solr-4.7.0	library, web-framework, database, network-server, xml	network-server, web-framework

TABLE 4.3. Examples of ASF libraries categorized using DBYCAT. These examples were taken from the testing datasets in the 10-fold cross validation process

TABLE 4.4. Values for the Kolmogorov-Smirnov and Shapiro-Wilk tests

		Kolmogorov-Smirnov		Shapiro-Wilk	
		Statistic	p-value	Statistic	p-value
Bytecode 1	Precision@1	0.386	1.758×10^{-67}	0.624	1.889×10^{-18}
	Precision@5	0.419	8.749×10^{-80}	0.600	5.302×10^{-19}
	Recall@1	0.368	5.880×10^{-61}	0.664	1.624×10^{-17}
	Recall@5	0.397	1.860×10^{-71}	0.652	8.609×10^{-18}
	Reciprocal Rank	0.234	9.823×10^{-24}	0.732	1.160×10^{-15}
Bytecode 2	Precision@1	0.373	7.700×10^{-63}	0.630	2.560×10^{-18}
	Precision@5	0.415	3.505×10^{-78}	0.608	8.057×10^{-19}
	Recall@1	0.354	3.186×10^{-56}	0.672	2.662×10^{-17}
	Recall@5	0.402	2.633×10^{-73}	0.646	6.108×10^{-18}
	Reciprocal Rank	0.233	1.230×10^{-23}	0.743	2.558×10^{-15}
Source code	Precision@1	0.363	1.827×10^{-59}	0.633	3.009×10^{-18}
	Precision@5	0.406	9.154×10^{-75}	0.620	1.462×10^{-18}
	Recall@1	0.342	2.565×10^{-52}	0.682	4.822×10^{-17}
	Recall@5	0.395	1.386×10^{-70}	0.657	1.085×10^{-17}
	Reciprocal Rank	0.243	8.819×10^{-26}	0.717	4.302×10^{-16}

TABLE 4.5. Values for the Kruskall-Wallis test

	Precision@1	Precision@5	Recall@1	Recall@5	Reciprocal Rank
Statistic	0.634	0.023	0.575	0.065	0.211
p-value	0.728	0.989	0.750	0.968	0.900

CHAPTER 5

Conclusions and Future Work

5.1 Conclusions

This document described DBYCAT, a model to categorize software libraries through the analysis of bytecode. DBYCAT uses Dirichlet Process Clustering to group similar *bytecode documents* together, and then the clusters were automatically labeled with existing categories extracted from a software repository. DBYCAT explored Dirichlet Process Clustering to group similar bytecode documents because it finds the appropriate number of clusters automatically, thus DPC can be considered as a non-parametric clustering algorithm.

DBYCAT takes advantage of the similarity between the vocabulary used by developers in the source code of the software libraries and the vocabulary present in software profiles, to assign a relevant category to a software library. Therefore, DBYCAT is an example of how unstructured information (i.e. identifiers extracted from bytecode and text extracted from software profiles) can be used to create information retrieval systems. Additionally, DBYCAT can be viewed as a classifier which is trained with the bytecode and profiles of a particular set of software libraries, in order to categorize new ones. The categorization of a new software library is not computationally expensive; to categorize a new library, DBYCAT just computes the cosine similarity between the vector of terms which represents a library and the vector of terms which represent the existing clusters.

An experiment was conducted in order to evaluate the accuracy of DBYCAT. The first step was the training of the model using libraries extracted from Apache Software Foundation. The next step was the categorization of ASF libraries through a 10-fold cross validation process recommending a set of 5 categories. The final step was the categorization of projects not maintained by ASF (i.e. generalization), using the model trained with the bytecode and categories of the ASF libraries and recommending a set of 5 categories. The accuracy of the categorization was estimated when it recommends a single category (single-label) and several categories (multi-label).

The results of the experiment for the single-label case indicated that DBYCAT can accurately recommend a relevant category for new software library. For the multi-label case, the results indicates that in four out of ten cases the first recommended category is a correct one. We consider this result as being very encouraging, given that randomly suggesting a category would lead to a value of 0.055. The value for precision@5 indicates

that given the set of five suggested categories, at least one of them is correct. This reinforces the results obtained for the single-label case. Analyzing together the values of recall@k and precision@k, the results look remarkable, showing that in fact, despite the seemingly low value for the precision at k = 5, DBYCAT in fact recommends 76% of the correct categories for each library in the top five recommendations.

For the generalization case, the results of the survey showed that DBYCAT recommended a relevant category to every non-ASF project. Additionally, DBYCAT recommended the first relevant category within the top 2 positions.

On the other hand, since DBYCAT uses bytecode to train the classifier, and taking into account that some of information contained in the source code of the software libraries is lost during its compilation (e.g. the comments), this study was interested in comparing the results of the proposed model trained with bytecode with the results of the same model but trained with source code. The results of this comparison suggest that the bytecode is a good replacement of source code for the proposed model, encouraging us to explore the capability of the bytecode to create information retrieval approaches to support additional software engineering tasks.

The potential of DBYCAT resides in that it is able to categorize the libraries in a software repository using the existing tags of another repository, establishing a relationship between the vocabulary used in the libraries with the vocabulary used in the software profiles containing the categories. Thus, for example a software repository whose source code is not available and its libraries are not categorized can be characterized using the categories provided by other software repository such as SourceForge or OpenHub. Another example of the possible uses of DBYCAT is Maven Central, a repository of Java libraries (more than 90.000 unique artifacts[1]) without categories. Since this repository does not contain the source code of many of the libraries in it, it can be characterized using DBYCAT and the categories provided by other repository. This categorization is performed using a clustering algorithm that does not need prior knowledge about the number of clusters, offering at the end a suitable representation of the semantics of each cluster.

Additionally, when a new software developer wants to store a new software library in this type of software repositories, DBYCAT can support the assignment of relevant categories through the recommendation of a particular set. Taking into account the training dataset used in this study, it can be said that a developer of the Apache Software Foundation will be able to use DBYCAT to obtain suggestions about the appropriate categories to her new application. Using a suggested category, the ASF developer will ensure that its new library shares the same application domain with existing libraries. Nevertheless, at the end the developer will be the person who will choose the appropriate categories.

5.2 Future work

Future work will be focused on three aspects:

1. *Improving DBYCAT:* The proposed model should be trained using repositories which contains a comprehensive representation of application domains of Java libraries. An example of this kind of repositories is Maven Central[2]; the libraries could be

[1] `http://search.maven.org/#stats`
[2] `http://search.maven.org/`

automatically categorized without the intervention of developers using the set of categories of a similar repository such as SourceForge or Ohlo. In general, future work will be focused on exploring the capability of DBYCAT to categorize software libraries using previous knowledge extracted from different software repositories.

2. *Exploiting the bytecode:* Current information retrieval (IR) techniques and machine learning (MR) approaches use the source code of applications to support software engineering tasks. Since the bytecode contains most of the information contained in the source code, the bytecode becomes a good alternative to solve problems where the source code is not available. Hence, future work will be devoted to leverage the bytecode to create information retrieval and machine learning based approaches to solve that kind of problems.

3. *Exploring Dirichlet Process Clustering:* This clustering algorithm is considered a non-parametric algorithm because it does not require the number of cluster as a parameter (conversely to clustering algorithms such as K-Means and its variations), then it is suitable for modeling problems where the number of groups is unknown. However, this algorithm produces *partitions* over data, i.e. each datum is assigned to a single cluster. Because software categorization approaches need to deal with categorization using multiple labels, a partitional clustering algorithm where each cluster is described using a single label seems not to be adequate. Therefore, future work will focus on exploring algorithms that allow the assignment of a datum to several clusters, in order to describe a cluster with a single relevant label; thus, a library belonging to several clusters could be categorized with multiple labels. Particularly, future work will focus on the use of Hierarchical Dirichlet Process Clustering[47, 41] to improve the accuracy of the model at categorizing a software library using multiple labels.

Bibliography

[1] Raymond P L Buse and Westley Weimer, *Synthesizing API usage examples*, Proceedings of the 2012 International Conference on Software Engineering (Piscataway, NJ, USA), ICSE 2012, IEEE Press, 2012, pp. 782–792.

[2] Ryan Carlson, Hyunsook Do, and Anne Denton, *A clustering approach to improving test case prioritization: An industrial case study*, 2011 27th IEEE International Conference on Software Maintenance (ICSM), IEEE, September 2011, pp. 382–391 (English).

[3] Anna Corazza, Sergio Di Martino, Valerio Maggio, and Giuseppe Scanniello, *Investigating the Use of Lexical Information for Software System Clustering*, 2011 15th European Conference on Software Maintenance and Reengineering, IEEE, March 2011, pp. 35–44.

[4] Nick Craswell, *Mean Reciprocal Rank*, Encyclopedia of Database Systems, Springer, 2009, p. 1703.

[5] Nigel Crook, Ramon Granell, and Stephen Pulman, *Unsupervised Classification of Dialogue Acts Using a Dirichlet Process Mixture Model*, Proceedings of the SIGDIAL 2009 Conference: The 10th Annual Meeting of the Special Interest Group on Discourse and Dialogue (Stroudsburg, PA, USA), SIGDIAL '09, Association for Computational Linguistics, 2009, pp. 341–348.

[6] Yingnong Dang, Rongxin Wu, Hongyu Zhang, Dongmei Zhang, and Peter Nobel, *ReBucket: A method for clustering duplicate crash reports based on call stack similarity*, 2012 34th International Conference on Software Engineering (ICSE), IEEE, June 2012, pp. 1084–1093.

[7] Tejinder Dhaliwal, Foutse Khomh, and Ying Zou, *Classifying field crash reports for fixing bugs: A case study of Mozilla Firefox*, 2011 27th IEEE International Conference on Software Maintenance (ICSM), IEEE, September 2011, pp. 333–342.

[8] William Dickinson, David Leon, A. Fodgurski, and Andy Podgurski, *Finding Failures by Cluster Analysis of Execution Profiles*, Proceedings of the 23rd International Conference on Software Engineering (Washington, DC, USA), ICSE '01, IEEE Computer Society, 2001, pp. 339–348.

[9] Thomas S Ferguson, *A Bayesian analysis of some nonparametric problems*, The annals of statistics (1973), 209–230.

[10] M Grechanik, Fu Chen, Xie Qing, C McMillan, D Poshyvanyk, and C Cumby, *A search engine for finding highly relevant applications*, Software Engineering, 2010 ACM/IEEE 32nd International Conference on, vol. 1, 2010, pp. 475–484.

[11] M Grechanik, K M Conroy, and K A Probst, *Finding Relevant Applications for Prototyping*, Mining Software Repositories, 2007. ICSE Workshops MSR '07. Fourth International Workshop on, 2007, p. 12.

[12] Nie Jian-Yun, F Paradis, and J Vaucher, *Using information retrieval for software reuse*, Computing and Information, 1993. Proceedings ICCI '93., Fifth International Conference on, 1993, pp. 448–452.

[13] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu, *DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones*, 29th International Conference on Software Engineering (ICSE'07), IEEE, May 2007, pp. 96–105.

[14] Tian Kai, M Revelle, and D Poshyvanyk, *Using Latent Dirichlet Allocation for automatic categorization of software*, Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on, 2009, pp. 163–166.

[15] S. Kawaguchi, P.K. Garg, M. Matsushita, K. Inoue, and Z. Source, *Automatic categorization algorithm for evolvable software archive*, Sixth International Workshop on Principles of Software Evolution, 2003. Proceedings., IEEE, 2003, pp. 195–200.

[16] Shinji Kawaguchi, Pankaj K Garg, Makoto Matsushita, and Katsuro Inoue, *MUDABlue: An automatic categorization system for Open Source repositories*, Journal of Systems and Software **79** (2006), no. 7, 939–953.

[17] Kenichi Kobayashi, Manabu Kamimura, Koki Kato, Keisuke Yano, and Akihiko Matsuo, *Feature-gathering dependency-based software clustering using Dedication and Modularity*, 2012 28th IEEE International Conference on Software Maintenance (ICSM), IEEE, September 2012, pp. 462–471.

[18] William H Kruskal and W Allen Wallis, *Use of ranks in one-criterion variance analysis*, Journal of the American statistical Association **47** (1952), no. 260, 583–621.

[19] A Kuhn, S Ducasse, and T Girba, *Enriching reverse engineering with semantic clustering*, Reverse Engineering, 12th Working Conference on, 2005, p. 10 pp.

[20] Adrian Kuhn, *On extracting unit tests from interactive live programming sessions*, 2013 35th International Conference on Software Engineering (ICSE), IEEE, May 2013, pp. 1241–1244.

[21] Adrian Kuhn, Stéphane Ducasse, and Tudor Gîrba, *Semantic clustering: Identifying topics in source code*, Information and Software Technology **49** (2007), no. 3, 230–243.

[22] M Linares-Vásquez, C McMillan, D Poshyvanyk, and M Grechanik, *On Using Machine Learning to Automatically Classify Software Applications into Domain Categories*, Empirical Software Engineering (EMSE) (2013).

[23] Y S Maarek, D M Berry, and G E Kaiser, *An information retrieval approach for automatically constructing software libraries*, Software Engineering, IEEE Transactions on **17** (1991), no. 8, 800–813.

[24] J.I. Maletic and A. Marcus, *Supporting program comprehension using semantic and structural information*, Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001, IEEE Comput. Soc, 2001, pp. 103–112.

[25] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze, *Introduction to Information Retrieval*, Cambridge University Press, New York, NY, USA, 2008.

[26] Frank J Massey, *The Kolmogorov-Smirnov Test for Goodness of Fit*, Journal of the American Statistical Association **46** (1951), no. 253, 68–78.

[27] C McMillan, M Grechanik, D Poshyvanyk, C Fu, and Q Xie, *Exemplar: A Source Code Search Engine For Finding Highly Relevant Applications*, Software Engineering, IEEE Transactions on **PP** (2011), no. 99, 1.

[28] C McMillan, M Linares-Vasquez, D Poshyvanyk, and M Grechanik, *Categorizing software applications for maintenance*, Software Maintenance (ICSM), 2011 27th IEEE International Conference on, 2011, pp. 343–352.

[29] Parastoo Mohagheghi and Reidar Conradi, *Quality, productivity and economic benefits of software reuse: a review of industrial studies*, Empirical Software Engineering **12** (2007), no. 5, 471–516.

[30] Tung Thanh Nguyen, Hoan Anh Nguyen, Jafar M. Al-Kofahi, Nam H. Pham, and Tien N. Nguyen, *Scalable and incremental clone detection for evolving software*, 2009 IEEE International Conference on Software Maintenance, IEEE, September 2009, pp. 491–494.

[31] Jim Pitman and Others, *Combinatorial stochastic processes*, Tech. report, Springer, 2002.

[32] Wenyi Qian, Xin Peng, Zhenchang Xing, Stan Jarzabek, and Wenyun Zhao, *Mining Logical Clones in Software: Revealing High-Level Business and Programming Rules*, 2013 IEEE International Conference on Software Maintenance, IEEE, September 2013, pp. 40–49.

[33] Simone Romano, Giuseppe Scanniello, Michele Risi, and Carmine Gravino, *Clustering and lexical information support for the recovery of design pattern in source code*, 2011 27th IEEE International Conference on Software Maintenance (ICSM), IEEE, September 2011, pp. 500–503.

[34] Thorsten Schafer, Ivica Aracic, Matthias Merz, Mira Mezini, and Klaus Ostermann, *Clustering for Generating Framework Top-Level Views*, 14th Working Conference on Reverse Engineering (WCRE 2007), IEEE, October 2007, pp. 239–248.

[35] Robert Schwanke, Lu Xiao, and Yuanfang Cai, *Measuring architecture quality by structure plus history analysis*, 2013 35th International Conference on Software Engineering (ICSE), IEEE, May 2013, pp. 891–900.

[36] Samuel Sanford Shapiro and Martin B Wilk, *An analysis of variance test for normality (complete samples)*, Biometrika (1965), 591–611.

[37] Lwin Khin Shar, Hee Beng Kuan Tan, and Lionel C. Briand, *Mining SQL injection and cross site scripting vulnerabilities using hybrid program analysis*, 2013 35th International Conference on Software Engineering (ICSE), IEEE, May 2013, pp. 642–651.

[38] Charles Song, Adam Porter, and Jeffrey S. Foster, *iTree: Efficiently Discovering High-Coverage Configurations Using Interaction Trees*, IEEE Transactions on Software Engineering **40** (2014), no. 3, 251–265.

[39] Mark D. Syer, Bram Adams, and Ahmed E. Hassan, *Identifying performance deviations in thread pools*, 2011 27th IEEE International Conference on Software Maintenance (ICSM), IEEE, September 2011, pp. 83–92.

[40] Yee Whye Teh, *Dirichlet process*, Encyclopedia of machine learning, Springer, 2010, pp. 280–287.

[41] Yee Whye Teh, Michael I Jordan, Matthew J Beal, and David M Blei, *Hierarchical dirichlet processes*, Journal of the american statistical association **101** (2006), no. 476.

[42] The Apache Software Foundation, *The Byte Code Engineering Library (Apache Commons BCEL)*.

[43] Cornelis J Van Rijsbergen, Stephen Edward Robertson, and Martin F Porter, *New models in probabilistic information retrieval*, Computer Laboratory, University of Cambridge, 1980.

[44] Shaowei Wang, David Lo, and Lingxiao Jiang, *Inferring semantically related software terms and their taxonomy by leveraging collaborative tagging*, 2012 28th IEEE International Conference on Software Maintenance (ICSM), IEEE, September 2012, pp. 604–607.

[45] Tao Wang, Huaimin Wang, Gang Yin, C X Ling, Xiang Li, and Peng Zou, *Mining Software Profile across Multiple Repositories for Hierarchical Categorization*, 2013, pp. 240–249.

[46] Wei Zhao, Lu Zhang, Hong Mei, and Jiasu Sun, *Requirements guided dynamic software clustering*, 21st IEEE International Conference on Software Maintenance (ICSM'05), IEEE, 2005, pp. 605–608.

[47] Tianbing Xu, Zhongfei Zhang, Philip S Yu, and Bo Long, *Evolutionary clustering by hierarchical dirichlet process with hidden markov state*, Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on, IEEE, 2008, pp. 658–667.