On Using Machine Learning to Automatically Classify Software Applications into Domain Categories

Mario Linares-Vásquez, Collin McMillan, Denys Poshyvanyk, Mark Grechanik

Abstract Software repositories hold applications that are often categorized to improve the effectiveness of various maintenance tasks. Properly categorized applications allow stakeholders to identify requirements related to their applications and predict maintenance problems in software projects. Manual categorization is expensive, tedious, and laborious – this is why automatic categorization approaches are gaining widespread importance. Unfortunately, for different legal and organizational reasons, the applications' source code is often not available, thus making it difficult to automatically categorize these applications. In this paper, we propose a novel approach in which we use Application Programming Interface (API) calls from third-party libraries for automatic categorization of software applications that use these API calls. Our approach is general since it enables different categorization algorithms to be applied to repositories that contain both source code and bytecode of applications, since API calls can be extracted from both the source code and byte-code. We compare our approach to a state-of-the-art approach that uses machine learning algorithms for software categorization, and conduct experiments on two large Java repositories: an open-source repository containing 3,286 projects and a closed-source repository with 745 applications, where the source code was not available. Our contribution is twofold: we propose a new approach that makes it possible to categorize software projects without any source code using a small number of API calls as attributes, and furthermore we carried out a comprehensive empirical evaluation of automatic categorization approaches.

Keywords *Closed-source · Open-source · Software categorization · Machine learning.*

1 Introduction

Different software repositories have mushroomed in the past decade with many of them containing massive amounts of source code and different software artifacts. To facilitate browsing and searching of these repositories, software systems are placed into categories (e.g., text editors, financial, or databases). Since many stakeholders are engaged in maintaining software, these stakeholders benefit from properly categorized software repositories for two reasons. First, grouping applications with similar features allows stakeholders to decide what features they should implement in their own applications that belong to same groups or categories (Kawaguchi et al. 2006; Dumitru et al. 2011). Second, stakeholders can determine what problems or bugs are common to many applications in the same category, and in turn predict what problems or bugs other applications from the same category are likely to encounter (Weiss et al. 2007; Zimmermann et al. 2009); this type of prediction could be used as a quality assurance technique to recognize typical bad smells or mistakes in the code that should be avoided during programming.

Automatic categorization of software applications in repositories is increasingly gaining acceptance since it reduces the manual effort significantly (Di Lucca et al. 2002; Ugurel et al. 2002;

Kawaguchi et al. 2003; Bruno et al. 2005; Kawaguchi et al. 2006; Sandhu et al. 2007; Tian et al. 2009; Dumitru et al. 2011). Currently, software applications are categorized by applying text classification approaches; terms (i.e., words in identifiers and comments) are extracted from the source code of applications and these terms serve as the *attributes* used as input to a machine learning algorithm that places applications into categories. Even though automatic categorization approaches do not achieve perfect precision, they still enable stakeholders to quickly benefit from categorized applications when solving software maintenance tasks.

1.1 Categorization in Maintenance

Different software maintenance tasks rely on knowing groups of similar software, even if the programmers need similar artifacts other than source code. For example, commercial software development firms use similar software to help build new products that are based on existing products – e.g., by including features which are commonly requested, or to reuse software designs that have worked successfully in the past (Kang et al. 1990; Frakes et al. 1998; Dumitru et al. 2011). Categorization is also useful for preparing for similar software bugs (Weiss et al. 2007; Zimmermann et al. 2009), and in knowing which teams in a global development environment have the expertise to build a particular kind of software (Bugde et al. 2008). In these cases, the manual categorization as done on repositories such as SourceForge is impractical because the software repository already exists. Likewise, even approximate solutions are useful because the penalty for incorrect categorization is often low. For example, programmers are adept at separating relevant results from irrelevant results when searching for software (Sim et al. 2011).

Unfortunately, existing approaches are untenable in a commercial environment. The reason is that these approaches rely on source code, but the source code is not available. For example, consulting companies such as Accenture, IBM, and HP Global Services, do not own the source code that they produce – their clients do[1]. Moreover, companies have a lot of legacy applications that are deployed on production environments and the source code is not available because it is lost.

Another case when source code may not be usable is when consultants build mission-critical software for different industries. Programmers in financial and biopharmaceutical companies often work in "clean-rooms," where the source code is written and kept on company's premises in physically secured environments (Grechanik et al. 2010; Jones 2010). External consultants from outsourcing companies such as Accenture, IBM, and HP Global Services come to the clean-room of the client company to write client's applications. Actions of these consultants are tightly monitored; electronic connections to outside of the company's network, phone calls, USB keys, and cameras are strictly forbidden. Once the applications are built, their executables are often released to consulting companies for testing. Clean-room development effectively negates the opportunity for consulting companies to accumulate knowledge about applications they build, and more importantly to use this knowledge for different software maintenance tasks.

1.2 Our Solution

Our idea is to use external *Application Programming Interface (API) calls* from third-party libraries and packages that are invoked in software applications (e.g., the Java Development Kit (JDK)) as a set of attributes for categorization. Using API calls as attributes is based on the fact that programmers typically build software using API calls from well-defined and widely used libraries (Poshyvanyk and Grechanik 2009; Grechanik et al. 2010; McMillan et al. 2011; McMillan et al. 2011). APIs are already grouped in packages and libraries based on their functionalities by the programmers who built those APIs. That fact that the APIs are grouped makes APIs ideal for use in machine learning approaches to categorize applications. For example, a music player application is more likely than a text editor to use a sound output library, and finding APIs from this library in the music player application enables us to put it in a proper category. Moreover, APIs are common to many software programs and invocations of the API calls can be extracted from the executable form of applications because the API calls exist in external packages and libraries. In addition, using API calls results in fewer attributes when compared to approaches that use all words from the source code to categorize applications, thus potentially improving the performance of categorization approaches. *A key question is how selecting APIs for categorizing applications compares with approaches that rely on selecting all words in the source code of applications?*

[1]Accenture policy 69 states that source code constitutes confidential information because it is information or material, not generally available to the public, that is generated, collected or used by the Company and that relates to its business, research and development activities, clients, or employees.

In this paper, we investigate this question by empirically studying large sets of Java applications from different repositories and applying different machine learning algorithms with different settings to obtain the answer by analyzing the results using statistical methods; we used the categories defined by the repositories, where these categories describe functionalities (domain categories). To address this, we defined the following research questions:

- **RQ_1**: Which machine learning algorithm is most effective for software categorization?
- **RQ_2**: Which level of granularity of API-based is more effective for software categorization?
- **RQ_3**: Are the API methods, classes or packages as effective attributes as source code for software categorization?
- **RQ_4**: Which kernel type is most effective for software categorization using Support Vector Machines?

To our knowledge, this is the first time that different machine learning approaches were thoroughly evaluated for software categorization on large sets of software applications. All of our case study data is available online[2]. Our contributions are summarized the following:

- A new approach to software categorization based on the APIs used by the applications. We extracted the API information in three forms: as the API packages that contain calls used by applications, as the API classes, and as the terms in the API methods. We found that using API packages results, on average, in F-Measure that is 82.34% better than using API classes; and API methods results in F-Measure that is 6.83% better than using API packages. Our approach is the first one that is able to categorize applications both in open-source and *closed-source* software repositories.
- We have built our approach and tested it on two software repositories: 745 Java closed-source applications from Sharejar [3], and 3,286 Java open-source applications from SourceForge[4]. We categorized software applications using five different machine-learning algorithms and four types of attributes, and show that *Support Vector Machines (SVM)* is the best-performing algorithm for categorization over these repositories. Despite SVM was also reported as the best performing algorithm in previous work using single labels (Ugurel et al. 2002), we used a transformation process to allow the SVM to deal with multiple categories. Additionally, we compared four types of kernel functions used with SVM and the results show that the linear kernel has the best performance.
- To demonstrate how competitive our approach is, we compared it with the closest baseline approach by Ugurel et al. (Ugurel et al. 2002) that has previously been tested on 330 applications from SourceForge and 1,353 projects from IBiblio[5]. As mentioned before, programmers typically build software using API calls from well-defined and widely used libraries, and these API are grouped based on their functionalities. Thus, this grouping can be used to categorize software applications. Additionally, API calls can be extracted from bytecode of closed-source applications. Our results show that our approach is a good alternative to this competitive approach in that it reaches comparable rates of true and false positives while using significantly fewer attributes as the names of API packages or methods whose calls are made in applications. Although source code contains in many cases sensitive information in the client API names, our approach does not compromise any of this since we are relying on publicly available API calls (packages, classes and methods).

2 Background

In general, categorization is the task of assigning a finite set of *categories* to *software artifacts* such as software applications, bug reports, commits logs. Typical applications of categorization in software maintenance tasks are:

- Domain analysis and software reuse: domain analysis is related to identifying and structuring information for reuse (Prieto-Diaz 1990). Software categories represent the

[2] http://www.cs.wm.edu/semeru/catml_ese/ (verified on 05/07/2012)

[3] http://sharejar.com/ (verified on 05/07/2012)

[4] http://sourceforge.net/ (verified on 05/07/2012)

[5] http://www.ibiblio.org/ (verified on 05/07/2012)

Table 1. Approaches for software categorization

Approach	Categories	Relationship between categories	Fuzzy categorization
Binary	One of two	Mutually exclusive	No
Multi-class	One of *n*	Mutually exclusive	No
Multi-label (Non-hierarchical)	*m* of *n*	Non hierarchical order	No
Hierarchical	*m* of *n*	Hierarchical order	No
Ranking	*m* of *n*	Non hierarchical order	Yes

dominant groups of features in software projects of the same domain (Kelly et al. 2011), therefore categories are used as labels for code retrieval and reuse in software repositories, and to analyze concepts of applications with similar features.

- Bug prediction: bugs or types of bugs describe design and implementation problems in software applications, and predicting these bugs is usually based on analysis of change requests, bug reports, and bad smells recognition (Antoniol et al. 2008; Menzies and Marcus 2008). Thus, bug types assigned to software applications could be considered as categories that can be assigned to new applications (during implementation), as a non-human-based-execution testing technique. Moreover, because bugs are common to many applications in the same domain category, the bugs discovered in each application could be used as attributes that permit distinguish applications by domain categories. A possible application of software categorization is using bugs reported on applications belonging to the same category to describe the kind of problems that represents the applications in the same category.
- Quality prediction: source-code metrics or the results of evaluating software applications with a quality model can be used to determine what quality attributes are common to many applications with similar features. Thus, a possible application of software categorization is to identify the quality attributes that describe software applications by category domain.

The automatic categorization process can be outlined as follows. First, a set of *attributes* is selected that characterize the software applications. These attributes may contain all words in an application (not including language keywords) or only the names of API packages/classes/methods whose calls are made in the application, as we have done in our approach. Second, a machine-learning algorithm uses the attributes, applications, and categories to build a model and then generate *predictions*, which are the algorithm's mapping of applications to a category. That is, the job of an automatic categorization tool is to compute a function that maps applications to categories.

The intuition behind the reason of why automatic categorization works is that certain attributes occur more often in applications belonging to one category than another category. For example, applications in the category `Email` contain terms such as "replyto" and "mailbox," whereas applications in the category `Databases` have terms such as "sql." Machine learning algorithms rely on the specificity of these attributes to certain categories. The accuracy of these algorithms worsens if the attributes are distributed arbitrarily across applications that belong to different categories. The intuition behind our idea is that the APIs used by applications are less likely to be distributed arbitrarily than the terms, because programmers often choose the terms (i.e., the names of identifiers) arbitrarily, whereas the set of APIs is predefined.

2.1 Overview of Software Categorization Approaches

In this paper, we implement a *multi-label* approach for software categorization, whereas previous approaches have been implemented with *single-label* algorithms. Single-label categorization can be *binary* if the set of available categories is composed of two mutually exclusive categories or *multi-class* if the set of available categories is composed of *m* mutually exclusive categories.

Multi-class problems are usually resolved by dividing the original problem into binary classification sub-problems. Two typical strategies are *one-against-all* and *one-against-one* (Hsu and Lin 2002; Lorena and De Carvalho 2004). In the one-against-all approach, the multi-class problem is solved building *n* binary classifiers (*n* is the number of categories), each one aimed to classify a category versus the remaining categories. For each category, a classifier is trained with all the applications in that category as positive examples and the rest of the applications as negatives. The classifier that produces the highest output usually predicts the category of a new application. In the one-against-one approach, *n(n-1)/2* binary classifiers are trained, each one aimed to categorize between two different categories. To predict the class of a new application, a majority voting strategy must be implemented.

Multi-label categorization has three approaches. The first one is a *non-hierarchical* approach, which consists in categorizing each application as belonging to *m* of *n* categories and there is not a hierarchical order between them. The second one is a *hierarchical* approach in which exists a hierarchical order between categories; for example the category "Role-Playing" in SourceForge belongs to "Card Games" category and its root-category is "Games/Entertainment". And the third approach is called *ranking* or *fuzzy categorization*, in which each application is classified as belonging to *m* of *n* categories by using a membership vector. Table 1 summarizes software categorization approaches; in Table 1, the second column is related to the number of categories that can be assigned to an application, the third column is related to the kind of relationship that exists between the categories, and the last column describes if the approach supports fuzzy categorization.

Multi-label problems are usually resolved by transforming them into one or more single-label ones. According to de Carvalho et al. (de Carvalho and Freitas 2009), the methods for multi-label problems can be divided into two main approaches: algorithm dependent and algorithm independent. The first approach is also called *adaptation* and consists of using machine learning algorithms specifically designed for multi-label problems such as MMP (Crammer and Singer 2003), ML-KNN (Zhang and Zhou 2005), or BP-MLL (Zhang and Zhou 2006). The second approach consists of transforming the multi-label dataset to obtain single-label datasets and then perform the categorization process using an ensemble of binary classifiers (that is the reason to call this approach "algorithm independent"). These transformations are applied to the categories or the applications. For example, if an application called *FooZip* belongs to categories *backup*, *system* and *compression*, then the new transformed category would be *backup-system-compression*; in this case, the transformation consists in creating a new category from multi-label categories.

2.2 Attribute Selection

The accuracy of the classification depends on the attributes that are chosen to represent applications, and it is important to select attributes that distinguish the applications in each category (Guyon and Elisseeff 2003). For example, APIs that come from a music library are likely to occur in application that processes music than in other types of applications. In this paper, we use *Expected Entropy Loss (EEL)* to determine which attributes to use for categorization. Related approaches for categorization have used EEL, and we describe it further in Section 3 ((Di Lucca et al. 2002; Ugurel et al. 2002; Bruno et al. 2005; Kawaguchi et al. 2006; Antoniol et al. 2008; Menzies and Marcus 2008; Hindle et al. 2009)).

3 Our Approach

Since a plethora of automatic categorization approaches use the same categorization process, this paper builds upon the work by Ugurel et al. (Ugurel et al. 2002) that serves as the implementation baseline of this process. Ugurel et al. used only one type of attribute and one machine-learning algorithm to categorize a small number of applications (330 from SourceForge and 1,353 from IBiblio). Our approach builds on this work by evaluating multiple types of attributes and algorithms. Another important difference from previous studies is that our approach can be applied to closed-source repositories, whereas all previous approaches to software categorization were applicable to open-source repositories only. To define the specifics of our approach that set it apart from other competitive approaches, we must answer the following three design questions (DQ).

DQ_1: What is an application and what is a category?

An application is a collection of software artifacts that include source code files and/or executables, and this collection is defined as the latest release of a project from a software repository. A category is a grouping of applications based on the functionality the applications provide (e.g., `Games` or `Email`). For example, SourceForge contains thousands of projects that are organized into many categories, and we use these projects in our experiments in this paper.

DQ_2: Which attributes do we use for categorization?

Different approaches to software categorization use words from comments and identifiers as attributes that are extracted from the source code of those applications. We consider only single words as attributes and not combinations of single words such as bigrams, since previous empirical results showed that single words outperform combinations of words for software categorization (Ugurel et al. 2002).

Words from comments and identifiers cannot be used as attributes if only executable applications are available (as in closed repositories), since it is not possible to extract descriptive names of identifiers and comments without having access to the source code. This oftentimes practical scenario motivates us to select three more types of attributes from applications: API packages, API classes, and terms in the names of API methods, whose API calls are invoked in applications. In this paper, we use these three types of API-based attributes for closed- and open-source applications, and words from comments and identifiers in open-source applications, with different classification algorithms.

The three API-based attributes are both based on the API calls that applications use but refer to different levels of granularity. API packages are one such level of granularity. For example, an application that processes music files may use the package `javax.sound.midi`. We refer to API packages as simply *packages*. API classes are grouped in these packages and represent more fine-grained details about the utilized functionality. For example, a music player may use the class `MidiDevice` from `javax.sound.midi`. We refer to API classes as *classes*. API methods are the methods implemented in these classes and provide more-fine grained details about the functionality than classes. For example, a music player may invoke `getDeviceInfo` and `getMicroSecondPosition` methods of `javax.sound.midi.MidiDevice` class. We refer to API methods as *methods.* In this study we extracted the terms in the *methods* splitting the method names and then we used the terms as the attributes for software categorization. One important advantage to using packages, classes and methods as attributes is that they can be extracted from Java byte code – it is not mandatory to have the source code. The API packages, classes, and methods we detect in applications are from the Java SDK[6]. In spite of the fact that different applications call different third-party APIs, using machine learning methods to classify software applications with several third-party APIs can lead to model-overfitting (Alpaydin 2010) because some APIs could be very specific for a few applications, or model-underfitting (Alpaydin 2010) because some APIs are used by a lot of applications for a purpose not related to their domain (for example using a library for logging). Features provided by JDK are used by every application written in Java and those features are used by similar applications (Grechanik et al. 2010). Therefore, using the JDK provides us with an API that implements several features that are related to application domains, and we expect our results to generalize to other software repositories of Java applications.

Since we use *Expected Entropy Loss* (EEL) by Ugurel et al. (Ugurel et al. 2002) as attributes selection approach, we briefly describe this approach here for reproducibility of our results. EEL is almost a decade old algorithm that is shown to be highly effective for selecting the most relevant attributes of systems in software repositories (Ugurel et al. 2002). In EEL, words are selected from source code, and we adapt EEL as our attribute in this paper.

EEL works by ranking software system's attributes based on how well each attribute describes each category. The likelihood that an attribute is in a given category is referred to as that attribute's *entropy* for that category. For example, the package `javax.sound.midi` is likely to be specific to applications in the category `Music`, whereas `javax.swing` may be both in applications in `Music` and `Email`. The entropy of `javax.sound.midi` would be high for the category `Music`, relative to `javax.swing`. In this paper, we adapt EEL's definition and formulas for software categorization using API call information.

---

[6] http://www.oracle.com/technetwork/java/javase/downloads/ (verified on 05/07/2012)

We provide the following formulas for the reproducibility of our approach. Entropy *e(X)* is a measure of the uncertainty associated with an event and is expressed in terms of a discrete set of probabilities Pr(*X*) over an event $x_i \in X$, where *X* is the event space:

$$e(X) = -\sum_{i=1}^{n} \Pr(x_i) \log\left(\Pr(x_i)\right) \tag{1}$$

Let *C* be the event indicating whether an application is a member of the specified category (e.g., if the application is related to the category). Let *a* denote the event that the software system contains the specified attribute,

$$\Pr(C) = \frac{numberOfRelatedApplications}{numberOfApplications} \tag{2}$$

$$\Pr(\overline{C}) = 1 - \Pr(C) \tag{3}$$

$$\Pr(a) = \frac{numberOfApplicationsWithAttributeA}{numberOfApplications} \tag{4}$$

Pr(*C*) is the probability, for each category, that an application will be in that category, and Pr(*a*) is the probability, for each attribute, that an application will contain that attribute.

$$\Pr(\overline{a}) = 1 - \Pr(a) \tag{5}$$

$$\Pr(C \mid a) = \frac{numberOfRelatedApplicationsWithAttributeA}{numberOfApplicationsWithAttributeA} \tag{6}$$

$$\Pr(\overline{C} \mid a) = 1 - \Pr(C \mid a) \tag{7}$$

$$\Pr(C \mid \overline{a}) = \frac{numberOfRelatedApplicationsWithoutAttributeA}{numberOfApplicationsWithoutAttributeA} \tag{8}$$

$$\Pr(\overline{C} \mid \overline{a}) = 1 - \Pr(C \mid \overline{a}) \tag{9}$$

The *prior entropy* represents the overall distribution of applications into a category and is calculated as follows:

$$e(C) = -\Pr(C)\log\left(\Pr(C)\right) - \Pr(\overline{C})\log\left(\Pr(\overline{C})\right) \tag{10}$$

The *posterior entropy* represents the probability of a given attribute for a given category:

$$e_a(C) = -\Pr(C \mid a)\log\left(\Pr(C \mid a)\right) - \Pr(\overline{C} \mid a)\log\left(\Pr(\overline{C} \mid a)\right) \tag{11}$$

Likewise, the posterior entropy of the class when the attribute is absent is

$$e_{\overline{a}}(C) = -\Pr(C \mid \overline{a})\log\left(\Pr(C \mid \overline{a})\right) - \Pr(\overline{C} \mid \overline{a})\log\Pr(\overline{C} \mid \overline{a}) \tag{12}$$

Thus the *expected posterior entropy* is

$$EPE(C,a) = e_a(C)\Pr(a) + e_{\overline{a}}(C)\Pr(\overline{a}) \tag{13}$$

And the *expected entropy loss* is

$$EEL(C,a) = e(C) - EPE(C,a) \tag{14}$$

Each attribute has an EEL value for each category. Attributes with higher value of EEL for a category are more discriminatory and provide more information for the categorization. The attributes with the highest EEL for each category are used to train categorization algorithms

DQ3: What machine learning algorithm do we use?

There are two types of machine learning technique: supervised and unsupervised. In supervised machine learning, a *training set* of pre-categorized applications is used to build a mapping between the attributes of the applications and the categories. Then this mapping is used to predict the categories to which uncategorized applications belong. On the other hand, unsupervised learning (also called *clustering*) generates categories (e.g., clusters) based on the latent structure (patterns, regularities, similarities, etc.) in the items being categorized. In this paper, we use supervised algorithms because we have a pre-defined set of categories. We analyzed Support

Vector Machines, Naïve Bayes, Decision Tree, RIPPER, and IBK in our experiments, and describe these algorithms further in Section 3. Support Vector Machines, Decision Trees and Naïve Bayes have been used previously for software categorization in (Ugurel et al. 2002), (Kawaguchi et al. 2003), and (Sandhu et al. 2007), respectively. RIPPER and IBK have not been used for software categorization; however they have been widely used in text categorization (Sebastiani 2002). We chose to test these five algorithms because they represent different ways that are commonly used to resolve a categorization problem using machine learning techniques. (Sebastiani 2002).

Decision Trees (DT) uses a "divide and conquer" strategy to split the problem space into subsets. A DT is modeled like a tree in which the root and the nodes are questions, and the arcs between nodes are possible answers to the questions. The leaves of the tree are the categories. DTs are able to deal with categorical inputs and multi-class problems. Thus, in a categorization problem, the inputs for DT are the attributes of one application and the output is a category. Kawaguchi et al. (Kawaguchi et al. 2003) used DTs for software categorization.

Naïve Bayes (NB) classifiers assume that all the attributes are independent and that each contributes equally to the categorization. A category is assigned to a project by combining the contribution of each feature. This combination is achieved by estimating the posterior probabilities of each category by using Bayes' Theorem. Prior probabilities are estimated with training data. This kind of classifier is able to deal with categorical inputs and multi-class problems. Thus, in a categorization problem, the inputs for NB are the attributes and the output is the probability distribution of the project on the categories. NB was used for software categorization by Sandhu et al. (Sandhu et al. 2007).

Support Vector Machines (SVM) split the problem space into two possible sets by finding a hyper-plane that maximizes the distance with the closest item of each subset. The function that splits the hyper-plane is known as the *kernel* function. If the data is linearly separable a linear kernel function is used with the SVM, in other case non-linear functions such as polynomials, radial-basis (RBF), and sigmoid should be used (Alpaydin 2010- Chapter 13). SVMs are binary classifiers, but can be used for multi-class classification using *one-against-one* or *one-against-all* strategies (Section 2.2). In this paper, we arrange multiple SVM classifiers in a *one-against-one* strategy and use a label transformation to allow multi-label classification. In the one-against-one approach, each classifier is trained to recognize two classes (Hsu and Lin 2002). We generate the predictions by a vote of the predictions from the classifiers (Ugurel et al. 2002).

IBK is an instance-based classifier that uses the *k nearest neighbors* to assign a category to a project (Aha et al. 1991). *IBK* is a lazy classifier because it does not induce a categorization model from training data. The categorization process is achieved by comparing the new instance with all the instances in the datasets. Thus, the category for the new instance is selected from the categories of the *k* most similar instances. In a categorization problem, the inputs for IBK are the features and the output is a category. Hindle et al. (Hindle et al. 2009) used *IBK* for classification of large commits in software repositories.

RIPPER is a classifier based on propositional rules (Cohen 1995). Ripper builds a rule set by adding rules to an empty rule set until all positive instances in the dataset are covered. Rules are formed adding conditions to the antecedent of a rule until no negative examples are covered. Hindle et al. (Hindle et al. 2009) used *RIPPER* to classify large commits for mining software repositories.

4 Case Study Design

In Section 5, we consider four types of attributes (terms, methods, classes, and packages) and use EEL to select a subset of those attributes for categorization. We outlined five different machine-learning algorithms for generating predictions. In this section, we discuss the design of a case study to evaluate different configurations of our approach.

4.1 Settings of the Case Study

The settings of the case study include the applications we want to categorize and the implementation (configuration) details behind the machine learning algorithms. While implementing our approach, we used the same implementations of machine learning algorithms as in (Ugurel et al. 2002) to allow direct comparison of these algorithms with our techniques.

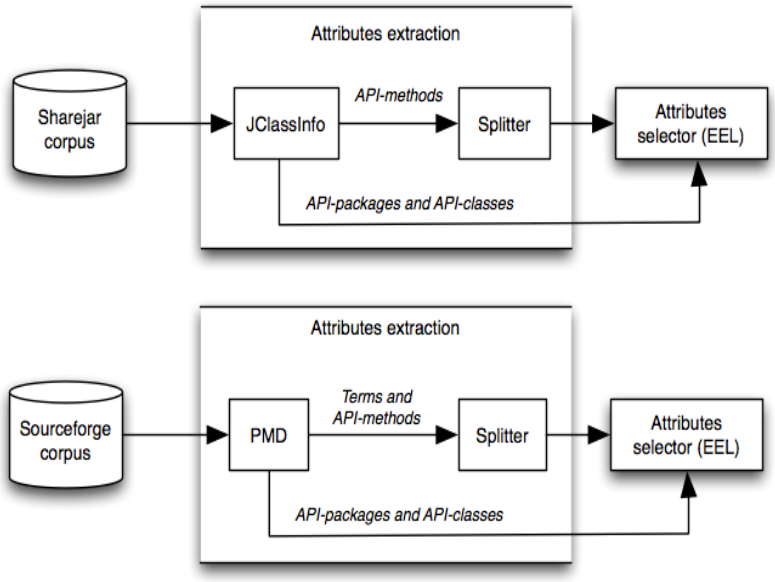

Figure 1. Attribute Selection Model

4.1.1 Software Repositories

We downloaded 8,310 Java applications from SourceForge (open source), and these applications are spread across the 22 categories in Table 2. We also downloaded 745 Java applications from Sharejar (closed source), and these applications were created for mobile phones and include only the compiled Java byte-code. The Sharejar categories are listed in Table 3.

In general, projects may belong to one or more categories in the same repository. All categories from both repositories do not include any sub-categories. Also, in SourceForge, we selected only categories with at least 100 applications in order to limit the number of categories; we did not consider applications that are not in these top categories in our case study. In Sharejar, we considered all categories, and there were no uncategorized applications. In total, we consider 3,286 of the applications from SourceForge and all 745 applications from Sharejar. The SourceForge applications are distributed into the categories as follows: 2,521 applications have one category, 645 with two categories, 103 with three categories, 16 with four categories, and 1 with five categories; distribution of Sharejar applications by number of categories is 477 with one category, 131 with two categories, and 137 with three categories.

4.1.2 Selection of Attributes

We implemented our approach using JClassInfo[7] to extract the method, package and class attribute sets from the byte-code of the Sharejar applications. For the SourceForge repository, we implemented our approach using PMD[8] to extract the API packages, classes, and terms directly from the applications' source code.

Once we had the terms and methods from each project, we split them using well-known naming conventions, such as camel case and underscores (Dit et al. 2011). For example `getSocket` is split into terms `get` and `socket`, and `send_reply` is split into terms `send` and `reply`. Each term, method, package or class is considered as an attribute, then, it was necessary to select the attributes to be used for categorization. We used EEL to rank attributes for each category from the training set (see Section III). In keeping with the case study design from (Ugurel et al. 2002), we

[7] http://jclassinfo.sourceforge.net/ (verified on 05/07/2012)

[8] http://pmd.sourceforge.net/ (verified on 05/07/2012)

Table 2. SourceForge projects and categories

Category	Count	Category	Count
1. Bio-Informatics	323	12. Indexing	329
2. Chat	504	13. Internet	1061
3. Communication	699	14. Interpreters	303
4. Compilers	309	15. Mathematics	373
5. Database	988	16. Networking	360
6. Education	775	17. Office	522
7. Email	366	18. Scientific	326
8. Frameworks	1115	19. Security	349
9. Front-Ends	584	20. Testing	904
10. Games	607	21. Visualization	456
11. Graphics	313	22. Web	534

Table 3. Sharejar projects and categories

Category	Count	Category	Count
1. Chat & SMS	320	8. Music	50
2. Dictionaries	30	9. Science	20
3. Education	90	10. Utilities	190
4. Free Time	120	11. Emulators	30
5. Internet	180	12. Programming	10
6. Localization	20	13. Sports	40
7. Messengers	50		

then selected the attributes that best distinguish each category with the following procedure. First, we excluded attributes that occurred in only one project. Second, we excluded attributes that occurred in less than 7.5% of the projects in a category. The intuition behind this threshold is that terms are unlikely to represent the meaning of a category if only a few applications in that category contain the term. We chose 7.5% because Ugurel et al. found good results using that threshold. Finally, we choose the top 100 remaining attributes for each category according to the attributes' EEL for the category. Table 4 lists the number of attributes that we used in the categorization; some attributes are common between the categories, therefore, there is a reduction of the set of the top-attributes to the set of attributes we used in the categorization.

4.1.3 Machine Learning Algorithms

We used the WEKA[9] implementation of five machine learning algorithms for our approach. For SVM, we used the WEKA Libsvm wrapper[10]. In all the cases, we used WEKA's default parameters, which were the same as were used in (Ugurel et al. 2002), except for IBK algorithm in

[9] http://weka.sourceforge.net/ (verified on 05/07/2012)

[10] http://www.csie.ntu.edu.tw/~cjlin/libsvm/ (on 05/07/2012)

Table 4. Attributes selected for software categorization. The top-attributes column lists the sum of attributes with the top values of expected entropy loss (EEL) in each category

Dataset	Attribute type	Total of Top-attributes	Attributes used in categorization
SourceForge	Packages	2,030	176
SourceForge	Classes	2,082	183
SourceForge	Methods	2,200	473
SourceForge	Terms	2,200	1,720
Sharejar	Packages	505	65
Sharejar	Classes	1,300	799
Sharejar	Methods	1,300	641

which we use several values for *k* (3, 5, 7 and 9).

4.2 Research Questions

Our goal was to determine how we can best categorize software based on the APIs used in each application. Therefore, we addressed the following research questions (RQ) in our case study:

RQ_1: Which machine learning algorithm is most effective for software categorization?

RQ_2: Which level of granularity of API-based attribute (API methods, API classes, or API packages), is more effective for software categorization?

RQ_3: Are the API methods, classes or packages as effective attributes as words (e.g., identifiers, comments) from source code for software categorization?

RQ_4[11]: Which kernel type is most effective for software categorization using Support Vector Machines?

The rationale behind RQ_1 was to compare different machine learning approaches on large application sets and obtain quantitative measurements of how well they categorize applications, including the SVM with a linear kernel used in (Ugurel et al. 2002). We also wanted to know how different types of attributes affect the accuracy of our approach. Specifically, we extracted three types of data based on the APIs used in applications, and in RQ_2 we wanted to study which of these types has better accuracy. Similarly, the purpose of RQ_3 was to compare our approach, where attributes are API methods, classes and packages, to competitive approaches that use words extracted from source code as attributes. For RQ_3, we compared the aggregate data for all five different machine learning algorithms. For RQ_2 and RQ_3 we used the results of the five algorithms (including the results of IBK using k=3,5,7,9) to generalize the results with each attribute. We assumed that some attributes provide better performance than others and it is consistent for all the algorithms[12]. Finally, we wanted to compare different kernel types as the purpose of RQ_4. These kernels are the linear, quadratic, radial-basis (RBF), and sigmoid. Table 5 shows the configurations of our approach that we use to answer each research question.

To respond to our research questions, we compared the algorithms' accuracy by using a 5-fold cross validation and the metrics described in the next section. In 5-fold validation, the dataset is randomly broken into five sections. One section is used to test the machine-learning algorithm and trained against the other four fifths. There are five iterations, and each section is used as the testing

[11] This research question was added once we analyzed the results of the study and found that SVM was the top performing ML algorithms across a range of parameters and settings

[12] By "consistency" we mean that all algorithms have the best performance for the same attribute set.

Table 5. Experiments settings. The last eight columns are the machine-learning algorithms. The rows are types of attribute from SourceForge or Sharejar. The cells indicate the research questions (RQ) that each configuration helps to answer.

Repository	Attribute	SVM	DT	NB	JRIP	IBK3	IBK5	IBK7	IBK9
Sharejar	Classes	1, 2, 4	1, 2, 4	1, 2, 4	1, 2, 4	1, 2, 4	1, 2, 4	1, 2, 4	1, 2, 4
Sharejar	Packages	1, 2, 4	1, 2, 4	1, 2, 4	1, 2, 4	1, 2, 4	1, 2, 4	1, 2, 4	1, 2, 4
Sharejar	Methods	1, 2, 4	1, 2, 4	1, 2, 4	1, 2, 4	1, 2, 4	1, 2, 4	1, 2, 4	1, 2, 4
SourceForge	Classes	1, 2, 4	1, 2, 4	1, 2, 4	1, 2, 4	1, 2, 4	1, 2, 4	1, 2, 4	1, 2, 4
SourceForge	Packages	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4
SourceForge	Methods	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4	1 - 4
SourceForge	Terms	1, 3, 4	1, 3, 4	1, 3, 4	1, 3, 4	1, 3, 4	1, 3, 4	1, 3, 4	1, 3, 4

set once. We chose 5-fold validation instead of 10-fold validation because recent studies have shown no statistical difference in the results from reduced iterations in validation (Feng et al. 2008).

4.3 Metrics and Statistical Analyses

4.3.1 Accuracy Metrics

The output of the machine learning algorithms is a set of predictions about the mapping of applications to categories. We evaluated these predictions using four metrics: precision (PRC), true positive rate or recall (REC), the F-score (F), and false positive rate (FPR). These metrics have been widely used as accuracy measures for machine learning (Újházi et al. 2010), including the case study by Ugurel et al. (Ugurel et al. 2002). We included the F-Measure in our study because it has been widely used in the machine learning and information retrieval fields as a metric that combines PRC and REC. The formulas for these metrics are as follows:

$$PRC = \frac{TP}{TP + FP}; REC = \frac{TP}{TP + FN}; F = 2\frac{PRC * REC}{PRC + REC}; FPR = \frac{FP}{FP + TN} \quad (15)$$

where TP is the number of true positives (applications correctly categorized), FP is the number of false positives (applications incorrectly categorized), TN is the number of true negatives (applications correctly identified as not belonging to the category), and FN is the number of false negatives (applications identified as not belonging to the category, that should have been). These metrics measure the following proportions.

- PRC measures the proportion of true positives over the number of instances categorized as positive.
- REC measures the proportion of true positives over the total number of positive instances.
- F-Measure, or simply F, is the harmonic mean of PRC and REC. It provides a way to combine precision and recall in a unique metric.
- FPR measures the proportion of false positives over the total number of negative instances.

We conducted a 5-fold cross validation study using different configurations of type of attribute, repository, and machine learning algorithms. For every category in a repository, we calculated the PRC, REC, and FPR of the predictions for that category for a given configuration (Table 5).

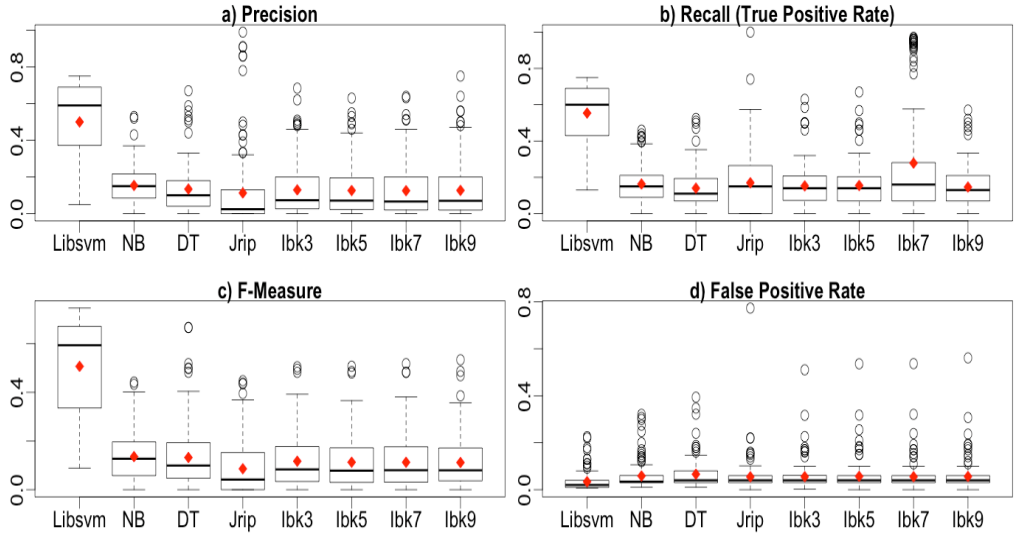

Figure 2. Performance metrics for each algorithm over all types of attributes in the categories of both repositories. The thick solid line is the median. The diamond is the mean and the box is the Interquartile Range (IQR). The thin line extends from Q1-1.5IQR to Q3+1.5IQR

4.3.2 Testing Statistical Significance

Our goal in our research questions was to compare the TPR, FPR and PRC of different algorithms using different types of attributes. Figures 2 and 3 show that TPR, FPR and PRC are not drawn from normal distributions, therefore, we cannot use parametric tests such as ANOVA to compare multiple classifiers, or t-test to compare two classifiers. Recent work in evaluating machine learning algorithms has suggested the *Wilcoxon Signed-Ranks test* and the *Friedman's test* with *Nemenyi's post-hoc procedure* to establish statistical significance without assuming that the samples are drawn from normal distributions (Demsar 2006). The Wilcoxon Rank test is a non-parametric test for comparing the performance of two classifiers over N datasets. The Friedman's test is a non-parametric test for comparing the performance of *k* classifiers over *N* datasets; if the null hypothesis is rejected using the Friedman's test for multiple classifiers, then the Nemenyi's test is used as a post-hoc procedure to compare pairs of classifiers. In our case we used the Bonferroni correction with the Friedman's test for controlling the family-wise error in multiple hypothesis testing (Demsar 2006). Moreover, we used the $\hat{p}$ non-parametric[13] effect size estimator suggested by Grissom and Kim (Grissom and Kim 2012), to measure the size effect of the difference in pairwise comparisons for the samples in the research questions; it is an estimate of the probability that a value randomly drawn from one sample will be greater than a value randomly drawn from other sample. The estimator $\hat{p}$ is defined as

$$\hat{p}_{a,b} = \frac{U}{n_a n_b} \tag{16}$$

where U is the Mann-Whitney Statistic, and $n_a n_b$ is the product of the two sample sizes. Therefore, for RQ_1, RQ_2, and RQ_4 we used the Friedman's test, and for RQ_3 we used the Wilcoxon Rank test; for all the research questions we used the statistic $\hat{p}$ as effect size estimator.

[13] We did not use typical effect size estimators such as Cohen's d, because we do not assume that the samples are drawn from normal distributions.

Table 6. Null hypotheses and p-values for **RQ_1** using the Nemenyi's test, and non-parametric effect sizes $\widehat{p}$ (based on Mann-Whitney's test). Each cell lists a null hypothesis that formulates that there is not significant difference between the values of a metric achieved with SVM and the values of the same metric achieved with other machine-learning algorithm. For example $\mathbf{H_0^1}$ formulates that there is no significant difference between the precision of SVM and DT. The Bonferroni corrected significance level (for Nemenyi's test) was 0,0018.

	P-values and effect sizes			
Comparison	**PRC**	**REC**	**F-MEASURE**	**FPR**
DT vs. SVM	H_0^1, p< 0,0001, $\widehat{p}$=0,8930	H_0^8, p < 0,0001, $\widehat{p}$=0,9734	H_0^{15}, p < 0,0001, $\widehat{p}$=0,9445	H_0^{22}, p < 0,0001, $\widehat{p}$=0,2625
NB vs. SVM	H_0^2, p< 0,0001, $\widehat{p}$=0,9107	H_0^9, p < 0,0001, $\widehat{p}$=0,9787	H_0^{16}, p < 0,0001, $\widehat{p}$=0,9361	H_0^{23}, p < 0,0001, $\widehat{p}$=0,2358
JRip vs. SVM	H_0^3, p< 0,0001, $\widehat{p}$=0,9072	H_0^{10}, p < 0,0001, $\widehat{p}$=0,9393	H_0^{17}, p < 0,0001, $\widehat{p}$=0,9630	H_0^{24}, p < 0,0001, $\widehat{p}$=0,2734
IBK3 vs. SVM	H_0^4, p< 0,0001, $\widehat{p}$=0,9109	H_0^{11}, p < 0,0001, $\widehat{p}$=0,9717	H_0^{18}, p < 0,0001, $\widehat{p}$=0,9507	H_0^{25}, p < 0,0001, $\widehat{p}$=0,2647
IBK5 vs. SVM	H_0^5, p< 0,0001, $\widehat{p}$=0,9152	H_0^{12}, p < 0,0001, $\widehat{p}$=0,9703	H_0^{19}, p < 0,0001, $\widehat{p}$=0,9531	H_0^{26}, p < 0,0001, $\widehat{p}$=0,2611
IBK7 vs. SVM	H_0^6, p< 0,0001, $\widehat{p}$=0,9151	H_0^{13}, p < 0,0001, $\widehat{p}$=0,8048	H_0^{20}, p < 0,0001, $\widehat{p}$=0,9520	H_0^{27}, p < 0,0001, $\widehat{p}$=0,2691
IBK9 vs. SVM	H_0^7 p< 0,0001, $\widehat{p}$=0,9117	H_0^{14}, p < 0,0001, $\widehat{p}$=0,9738	H_0^{21}, p < 0,0001, $\widehat{p}$=0,9540	H_0^{28}, p < 0,0001, $\widehat{p}$=0,2742

5 Case Study Results

5.1 RQ_1 – Machine-learning Algorithms

Our approach relies on a supervised machine-learning algorithm to extract a categorization model from the attributes and assign each application to one or more categories. Related work has studied only one supervised algorithm for software categorization, that is, SVM (Ugurel et al. 2002). In this paper, we contrast the results from five algorithms: SVM, DT, NB, RIPPER, and IBK with several values for *k* (3, 5, 7, and 9)[14].

Figure 2 shows a statistical summary of the PRC, REC, F-Measure and FPR for our run of each algorithm. Each boxplot represents the distribution of a metric for one algorithm for each category in both SourceForge and Sharejar, on all sets of attributes. We observe that the average values and medians of SVM outperform the other algorithms. Moreover, 75% of the values achieved with SVM (for each metric) outperform the Interquartile Range of the other algorithms[15]. We applied the Friedman's test to test the statistical significance of the difference in these results of each classifier. When testing the four metrics at a 5% confidence level, we found that p-values (two-tailed) are less than 0.0001[16]. Therefore, we reject the null hypothesis that there is no significant difference of the values of TPR, PRC, F-Measure, and FPR.

[14] For the *k* parameter in the IBK algorithm we used odd values to avoid tied votes.

[15] In Figures 2.a, 2.b, 2.c the values of SVM above the first quartile are higher than the third quartile of the other algorithms. In Figure 2.d the values of SVM below the third quartile are lower than the first quartile of the other algorithms

[16] The $Q_{critical}$ and $Q_{observed}$ values are provided in our online appendix.

Table 7. Null hypotheses and p-values for **RQ_2** using the Nemenyi's test, and non-parametric effect sizes $\hat{p}$ (based on Mann-Whitney's test). Each cell lists a null hypothesis that formulates that there is not significant difference between the values of a metric achieved with a dataset type and the values of the same metric achieved with other dataset type. For example **H_0^{29}** formulates that there is no significant difference between the precision of Methods and Classes. The Bonferroni corrected significance level (for Nemenyi's test) was 0,0167.

	P-values and effect sizes			
Comparison	**PRC**	**REC**	**F-MEASURE**	**FPR**
Methods vs. Classes	H_0^{29}, p< 0,0001, $\hat{p}$=0,6993	H_0^{32}, p< 0,0001, $\hat{p}$=0,6182	H_0^{35}, p< 0,0001, $\hat{p}$=0,6300	H_0^{38}, p=0.087, $\hat{p}$=0,5290
Methods vs. Packages	H_0^{30}, p< 0,0001, $\hat{p}$=0,5777	H_0^{33}, p=0.022, $\hat{p}$=0,5533	H_0^{36}, p= 0,009, $\hat{p}$=0,5349	H_0^{39}, p< 0,0001, $\hat{p}$=0,5637
Classes vs. Packages	H_0^{31}, p< 0,0001, $\hat{p}$=0,4052	H_0^{34}, p< 0,0001, $\hat{p}$=0,4358	H_0^{37}, p< 0,0001, $\hat{p}$=0,4088	H_0^{40}, p< 0,0001, $\hat{p}$=0,5460

Table 8. Null hypotheses and p-values for **RQ_2** using the Nemenyi's test, and non-parametric effect sizes $\hat{p}$ (based on Mann-Whitney's test) on Sharejar results. The Bonferroni corrected significance level (for Nemenyi's test) was 0,0167.

	P-values and effect sizes			
Comparison	**PRC**	**REC**	**F-MEASURE**	**FPR**
Methods vs. Classes	H_0^{29}, p< 0,0001, $\hat{p}$=0,6587	H_0^{32}, p=0.556, $\hat{p}$=0,4456	H_0^{35}, p< 0,0001, $\hat{p}$=0,5465	H_0^{38}, p< 0,0001, $\hat{p}$=0,5978
Methods vs. Packages	H_0^{30}, p< 0,0001, $\hat{p}$=0,7126	H_0^{33}, p=0.155, $\hat{p}$=0,4858	H_0^{36}, p< 0,0001, $\hat{p}$=0,6146	H_0^{39}, p< 0,0001, $\hat{p}$=0,5934
Classes vs. Packages	H_0^{31}, p< 0,0001, $\hat{p}$=0,5826	H_0^{34}, p=0.044, $\hat{p}$=0,5311	H_0^{37}, p=0.038, $\hat{p}$=0,5618	H_0^{40}, p=0.239, $\hat{p}$=0,4912

Table 6 lists the null hypotheses we used for the Nemenyi's post-hoc test on the difference between specific pairs of algorithms[17]. The p-values obtained and the effect sizes are also listed in Table 6. For all the hypotheses we found p-values lower than the significance level suggested by the Bonferroni correction (0.0018). For PRC, REC, and F-Measure, we found that the differences between the values reported by LIBSVM and the others algorithms are higher than 0.9; for FPR the effect size is small, with probabilities lower than 0.3. It means that there are significant differences between the performance measures of SVM compared to DT, NB, JRip, and IBK. Therefore, we reject all the null hypotheses (H_0^1 to H_0^{28}), meaning that the mean TPR, FPR, and PRC given by SVM are statistically significantly higher than the results from DT, NB, JRip, or IBK. However, for FPR, we found that there are small differences between using LIBSVM or the other algorithms.

We ran the same procedure using Sharejar and SourceForge as independent samples. In both cases the average, median and IQR of SVM outperforms the other algorithms in the four metrics. For Sharejar the average values of metrics achieved with Libsvm are 40.04% (PRC), 43.62%(REC), 40.24%(F-Measure), and 6.11%(FPR); for SourceForge the average values are 54.45%(PRC), 60.56%(REC), 55.31% (F), and 2.18%(FPR). Although SVM provides the highest

[17] We do not show any comparison of between DT, NB, JR,IP, and IBK because those algorithms performed less well than SVM

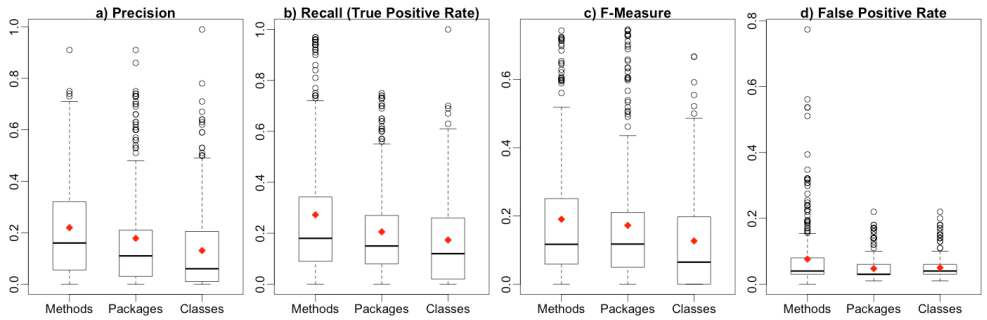

Figure 3. Performance metrics for methods, classes and packages over five algorithms in all categories of both repositories.

average values of performance, using SVM to categorize software applications in Sharejar provide PRC and REC that are below the 50%; for SourceForge the average values are higher than 50%. These values cannot be considered as outstanding performance measures for software categorization; using a random binary classifier (with the same probability for each class) could provide the same results without using a training procedure.

The results of the Nemenyi's tests using Sharejar and SourceForge as independent samples are similar to using a sample composed of Sharejar and SourceForge measures. We reject all the null hypotheses, meaning that the mean PRC, REC, F-Measure, and FPR given by SVM are statistically significantly higher than the results from DT, NB, JRip, or IBK.

Therefore, we answer RQ_1 by concluding that SVM is the most-effective machine-learning algorithm for categorization of the applications in both repositories we used as datasets in our evaluation.

5.2 RQ_2 – API-based Attributes of Applications

The quality of the results can be strongly affected by the attributes, which are used as input to the machine-learning algorithm. In this paper we propose three types of attributes that have never been tested before for software categorization: API methods, classes, and packages. This section compares the quality of categorization when using each of these types of attribute. We did this comparison using *Decision Trees*, *Naïve Bayes*, *Support Vector Machines, IBK (with k =3,5,7,9), and RIPPER* to minimize a threat to validity faced when using only one algorithm.

We used API methods, classes, and packages as the input to each of the machine learning algorithms, and computed the PRC, REC, F-Measure, and FPR in each category of both repositories. In our experiment, we grouped the results from several different machine learning algorithms. These are the results shown in Figure 3 and discussed below. An alternative approach would have been to examine the results for just one machine learning algorithm. The alternative design would have the advantage of focusing closely on a single algorithm, and could increase the usefulness of the results for that algorithm. While this advantage is important, we felt that it was outweighed by the need to maximize the generality of our results, since we aim to study how the attributes can be used for different categorization tasks.

A statistical summary of the results is presented in Figure 3. Each boxplot represents one type of attribute. We observe that the average PRC for packages is 17.82%, for classes is 13.03% and for methods is 21.97%. Methods is the API-based attribute with highest precision and classes is the lowest one; the average value of PRC for packages presents a roughly 37% improvement over classes, and for methods presents a roughly 23% improvement over packages. For REC, the average value for packages is 20.57%, for classes is 17.32%, and for methods is 27.21%. API methods presents a roughly 32% improvement over packages, and packages presents a roughly 20% improvement over classes. For F-Measure, we observe that the average value for packages is 17.23%, for classes is 12.68% and for methods is 19.03%. API methods provide a 10% improvement of accuracy over packages and a 50% accuracy improvement on classes. For FPR, we observe that the average value for packages is 4.74%, for classes is 4.98% and for methods is 7.62%. Thus, packages have the lowest FPR and methods the highest one. These values mean that about 27% of the predictions placed applications correctly into a category (TPR) using API

Table 9. Null hypotheses and p-values for **RQ_2** using the Nemenyi's test, and non-parametric effect sizes $\hat{p}$ (based on Mann-Whitney's test) on SourceForge results. The Bonferroni corrected significance level (for Nemenyi's test) was 0,0167.

	P-values and effect sizes			
Comparison	**PRC**	**REC**	**F-MEASURE**	**FPR**
Methods vs. Classes	H_0^{29}, p< 0,0001, $\hat{p}$=0,7293	H_0^{32}, p< 0,0001, $\hat{p}$=0,7238	H_0^{35}, p< 0,0001, $\hat{p}$=0,7093	H_0^{38}, p< 0,0001, $\hat{p}$=0,4717
Methods vs. Packages	H_0^{30}, p=0.915, $\hat{p}$=0,5774	H_0^{33}, p=0.074, $\hat{p}$=0,5774	H_0^{36}, p=0.183, $\hat{p}$=0,4795	H_0^{39}, p=0.003, $\hat{p}$=0,5485
Classes vs. Packages	H_0^{31}, p< 0,0001, $\hat{p}$=0,3294	H_0^{34}, p< 0,0001, $\hat{p}$=0,3294	H_0^{37}, p< 0,0001, $\hat{p}$=0,2905	H_0^{40}, p< 0,0001, $\hat{p}$=0,5874

methods, about 5% of predictions placed an application in wrong categories (FPR) using API packages, and 21.97% of the positive predictions were correct (PRC) using methods.

Therefore, there is a difference between performance measures for the three API-based attributes, and the Friedman's test shows that the difference in averages is statistically significant. When testing the four metrics at a 5% confidence level, we found that p-values (two-tailed) are less than 0.0001. These values suggest us to reject the null hypothesis stating that there is no significant difference between PRC, REC, F-Measure and FPR for packages, classes and methods. Then, according to the procedure proposed in (Demsar 2006), we applied Nemenyi's post-hoc test on the difference between specific pairs of API-based attributes. Table 7 lists the null hypotheses we used for the Nemenyi's post-hoc test. For the case of using Sharejar and SourceForge data we found that p-values for the comparison of REC of API methods vs. API packages, and FPR of API methods vs. API classes, are lower than the Bonferroni corrected significance level (0.0167). Lowest values of effect sizes are between API classes and API packages, and we found that probabilities of randomly selecting highest values between the samples in the hypotheses (H_0^{29} - H_0^{40}) are not higher than 0.7. Therefore, all the hypotheses for **RQ_2** are rejected except for H_0^{33} and H_0^{38}.

We applied the same procedure for Sharejar and SourceForge datasets and the results for the tests are in Tables 8 and 9. According to Nemenyi's tests in the three cases (Sharejar+SourceForge, Sharejar, SourceForge) there is no significant difference between REC of methods and packages. In SourceForge there is no significant difference between PRC of packages and methods, but there is significant difference in FPR values; packages outperform FPR of methods and classes, and methods have the highest FPR. As in the case of using both dataset as a whole, we also found that highest differences are between API methods and API classes; and lowest values of effect sizes are between API classes and API packages.

In Sharejar there is no significant difference between the REC of API-based attributes, however there is significant difference in PRC of the three attributes, and API methods have the highest average with the lowest variance. For FPR, classes and packages belong to the same group and are significantly different from methods. Thus, In the case of samples including Sharejar and SourceForge results, methods and packages outperformed REC and PRC of classes, and methods outperformed REC and PRC of packages. Again, lowest values of effect sizes are between API classes and API packages; and highest values between API methods and API classes.

Although there is no significant difference between REC and PRC of methods and packages, **API packages are more effective attributes in SourceForge** because they provide the lower values of FPR. **The terms in the names of API methods in Sharejar provide the best precision value, but packages have the highest REC with a lower FPR than methods. However, comparing API classes and API packages in SourceForge we found the lowest effect sizes; it means that API classes could provide same PRC and REC than API packages. In Sharejar the probabilities of differences between API classes and API packages are higher than 0.5; therefore API packages in Sharejar could provide better performance than API classes.**

API methods and packages are more effective attributes than API classes for categorization of the applications in both repositories we used as a dataset. However, methods have highest values of REC and PRC, and packages and classes have the lowest FPR values. Our result is somewhat surprising, in that one may expect fine-grain attributes, such as methods, to always be more telling than course-grain ones, such as classes. While methods do in fact outperform classes and packages, using API packages outperforms classes. This means that the programmers must use API classes in a variety of their projects, but select specific methods for the goals of each category. Along this vein of thought, programmers would tend to include packages only when those are needed in a category.

Table 10. Null hypotheses and p-values for **RQ_3** using the Wilcoxon Rank test, and non-parametric effect sizes $\hat{p}$ (based on Mann-Whitney's test). Each cell lists a null hypothesis that formulates that there is not significant difference between the values of a metric achieved with a dataset type and the values of the same metric achieved with other dataset type. For example $\mathbf{H_0}^{41}$ formulates that there is no significant difference between the precision of Methods and Terms.

	P-values			
Comparison	**PRC**	**REC**	**F-MEASURE**	**FPR**
Methods vs. Terms	$\mathbf{H_0}^{41}$, p=0,583, $\hat{p}$=0,4881	$\mathbf{H_0}^{43}$, p=0,303, $\hat{p}$=0,4928	$\mathbf{H_0}^{45}$, p= 0,060, $\hat{p}$=0,5301	$\mathbf{H_0}^{47}$, p=0,131, $\hat{p}$=0,4928
Packages vs. Terms	$\mathbf{H_0}^{42}$, p=0,853, $\hat{p}$=0,4925	$\mathbf{H_0}^{44}$, p<0.0001, $\hat{p}$=0,5815	$\mathbf{H_0}^{46}$, p= 0,446, $\hat{p}$=0,5195	$\mathbf{H_0}^{48}$, p=0,699, $\hat{p}$=0,5815

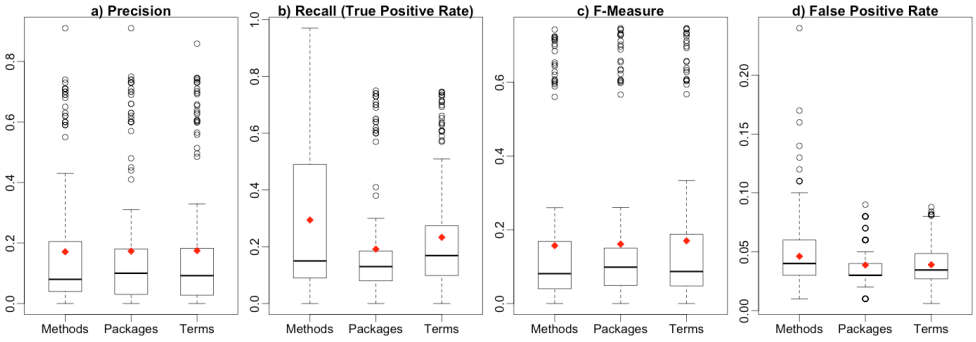

Figure 4. Performance metrics for terms, methods and packages over five algorithms in all categories of both repositories.

5.3 RQ_3 – API-based and Text-based Attributes

Case studies by other researchers studied the use of source code terms, and various combinations of these terms (e.g., bigrams, phrases, etc.), as attributes (Ugurel et al. 2002; Kawaguchi et al. 2006). These studies found that single words were the most-effective terms to use as attributes. This paper builds on this previous work by comparing the use of terms against the use of API-based attributes. Specifically, we compare packages and methods to single words from source code because we found them to be the API-based attributes with best REC and PRC values (see Section 5.2). For RQ_3, we compare methods versus terms, and packages versus terms using the Wilcoxon rank test. The rationale for this is that the Wilcoxon rank test is suggested to compare two samples over *N* datasets (Demsar 2006).

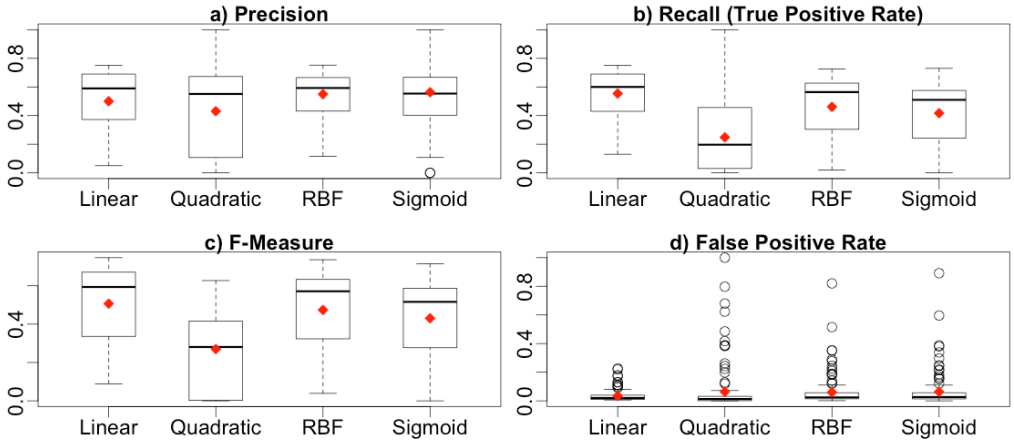

Figure 5. Performance metrics for terms, classes, methods and packages over four kernel types in all categories of both repositories.

Figure 4 shows the PRC, REC, F-Measure, and FPR for all five machine learning algorithms using the packages, methods and terms as attributes. These attributes come only from the applications in SourceForge because it was only possible to extract the terms from those applications – we had only byte-code for applications from Sharejar, which is quite typical for large development companies that do not own the source code that they develop. Using packages, the average PRC is 17.29%, REC is 19.16%, F-Measure is 16.12%, and FPR is 4.45%. Using methods, the PRC is 17.09%, REC is 29.40%, F-Measure is 15.69%, and FPR is 5.90%. Using terms, the PRC is 17.48%, TPR is 23.31%, F is 17.00%, and FPR is 4.85%.

Table 10 lists the null hypotheses and p-values of the Wilcoxon Rank test for **RQ_3**. The null hypotheses state that there is no significant difference between two samples. We only found significant difference with a p-value < 0.0001 and it was for REC of packages and terms, however, we did not find significant difference for PRC. For the comparison between methods and terms, in the four metrics we found values that suggest no significant difference.

According to the p-values of the Wilcoxon Rank tests, we cannot reject the null hypothesis that there is no statistically significant difference between the PRC, REC, F-Measure, and FPR when using methods or terms. Moreover, we can reject the null hypothesis that there is no statistically significant difference between the REC when using packages or terms, but we cannot reject the null hypothesis that there is no statistically significant difference between the PRC, F-Measure and TPR when using packages or terms. Therefore, we conclude that packages and methods are good alternatives to terms in the case when the terms are not available.

5.4 RQ_4 – Kernel Types

The baseline approach proposed in (Ugurel et al. 2002) uses a linear kernel function with a cost parameter equals to 100. For RQ_4 we want to evaluate what kernel function is the best model to separate the categories in order to improve the categorization performance, Thus, we compared the performance of linear kernel function with quadratic, RBF, and sigmoid kernel functions using the default parameters in WEKA and the same cost parameter proposed in (Ugurel et al. 2002). Figure 5 shows a statistical summary of the PRC, REC, F-Measure, and FPR for our run of SVM with different kernel types. Each boxplot represents a metric for one kernel for each category in both SourceForge and Sharejar, on all sets of attributes. We observe that the averages values of PRC are 50.00% for the linear kernel, 43.06% for the quadratic, 54.96% for the RBF, and 56.32% for the Sigmoid. In this case the sigmoid kernel function has the highest average value of PRC, however, the RBK kernel shows a similar distribution. The average REC values are 55.37% for linear, 24.83% for quadratic, 46.12% for RBF, and 41.70% for sigmoid. The average REC of the linear kernel outperforms the other types of kernels and the REC values of the linear kernel are

Table 11. Null hypotheses and p-values for **RQ_4** using the Nemenyi's test, , and non-parametric effect sizes $\hat{p}$ (based on Mann-Whitney's test). Each cell lists a null hypothesis that formulates that there is not significant difference between the values of a metric achieved with two different kernel types. For example **H_0^{49}** formulates that there is no significant difference between the precision of Linear and Quadratic kernels. The Bonferroni corrected significance level (for Nemenyi's test) was 0,0083.

	P-values			
Comparison	**PRC**	**REC**	**F-MEASURE**	**FPR**
Quadratic vs. Linear	H_0^{49}, p=0.007, $\hat{p}$=0,5622	H_0^{52}, p< 0,0001, $\hat{p}$=0,8601	H_0^{55}, p< 0,0001, $\hat{p}$=0,7933	H_0^{58}, p=0.215, $\hat{p}$=0,6226
Sigmoid vs. Linear	H_0^{50}, p=0.543, $\hat{p}$=0,4637	H_0^{53}, p< 0,0001, $\hat{p}$=0,6205	H_0^{56}, p< 0,0001, $\hat{p}$=0,5641	H_0^{59}, p< 0,0001, $\hat{p}$=0,4429
RBF vs. Linear	H_0^{51}, p=0.382, $\hat{p}$=0,4754	H_0^{54}, p< 0,0001, $\hat{p}$=0,6955	H_0^{57}, p< 0,0001, $\hat{p}$=0,6268	H_0^{60}, p< 0,0001, $\hat{p}$=0,4322

more grouped around the mean (the variance is the lowest for the four types of kernels). The F-Measure confirms the results of PRC and REC with the linear kernel providing the highest accuracy and the quadratic kernel the lowest accuracy. The average values of FPR are 3.39% for the linear kernel, 6.36% for the quadratic, 6.01% for the RBF, and 6.36% for the sigmoid; the linear kernel has the lowest FPR value with the lowest variance. Thus, we applied the Friedman's test to validate if the differences between the distributions of the performance measures are statistically significant. For PRC we got a p-value of 0.003 and for the other metrics p-values less than 0.0001. These values lead us to reject the null hypothesis stating that there is no significant difference between TPR, FPR, and PRC of the four types of kernels.

After the Friedman's test, we applied the Nemenyi's post-hoc test on the difference between specific pairs of kernel types to validate if there are significant differences between the linear kernel and the other. We choose the linear kernel as the baseline for the test, because it performs better that the other kernels for TPR and FPR. Table 11 lists the null hypotheses we used for the Nemenyi's post-hoc test, and the p-values. We cannot reject H_0^{50}, H_0^{51}, and H_0^{58} because their p-values are greater than the Bonferroni corrected significance level (0.0083). Although there are significant differences between the performance of the linear kernel and the other, using the REC and F as performance measures, there is no significant difference between the distributions of the FPR values obtained with the linear and quadratic kernels. Using PRC as performance measure, we cannot reject H_0^{49}, but we reject H_0^{50} and H_0^{51}; this means that there are no significant differences between the linear kernel and the RBF and Sigmoid kernels. Effect sizes for REC and F-Measure suggest that the highest differences are between Linear and Quadratic Kernels, and Linear and RBF Kernels; for FPR, the effect sizes between Linear and RBF, and Linear and Sigmoid are lower than 0.5. Therefore, we consider that Sigmoid could provide as good results as Linear Kernel.

Therefore, we can conclude that the linear kernel is the most-effective kernel for categorization of the applications in both repositories we used as datasets in our evaluation with average values of 50.00% (PRC), 55.37% (REC), 50.68% (F-Measure), and 3.39% (FPR).

6 Discussion and Future Work

SVM is a widely used algorithm and has been reported to outperform machine-learning classifiers in several domains such as text categorization (Leopold and Kindermann 2002), developer recommendation (Anvik et al. 2006; Anvik and Murphy 2011), and software categorization (Ugurel et al. 2002; McMillan et al. 2011). We confirmed this by using a one-against-all approach

Table 12. F-measure of linear SVM for each category and each attribute type in SourceForge

Category	F-Measure			
	Terms	Methods	Classes	Packages
1. Bioinformatics	0.743	0.722	0.272	0.743
2. Chat	0.745	0.743	0.169	0.742
3. Communications	0.605	0.589	0.149	0.601
4. Compilers	0.706	0.698	0.329	0.712
5. Database	0.693	0.648	0.213	0.689
6. Education	0.633	0.620	0.178	0.633
7. Email	0.746	0.718	0.168	0.746
8. Frameworks	0.724	0.724	0.304	0.727
9. Front-ends	0.712	0.704	0.241	0.702
10. Games	0.730	0.715	0.189	0.728
11. Graphics	0.603	0.604	0.217	0.607
12. Indexing/Search	0.733	0.721	0.214	0.732
13. Internet	0.608	0.599	0.163	0.604
14. Interpreters	0.638	0.628	0.198	0.635
15. Mathematics	0.655	0.653	0.238	0.658
16. Networking	0.568	0.560	0.124	0.567
17. Office/Business	0.594	0.595	0.271	0.598
18. Scientific	0.658	0.644	0.196	0.654
19. Security	0.627	0.607	0.188	0.622
20. Testing	0.733	0.714	0.170	0.730
21. Visualization	0.607	0.603	0.324	0.605
22. WWW/HTTP	0.698	0.684	0.172	0.696

to learn a multi-label model from the SourceForge and Sharejar datasets. This configuration provides better results for multi-class and multi-label problems than the other algorithms (NB, DT, JRip, IBK) because using a binary SVM classifier for each category allows the model to build better hyperplanes that maximizes the margin used to distinguish the categories.

6.1 Attributes and Classifier Accuracy

We found in SourceForge that using packages and methods as attributes provide better PRC and REC than classes, and terms outperformed packages and methods. Measuring the accuracy with the F-Measure we also found that classes achieved the lowest values, terms achieved the highest values, and packages and methods are good alternatives to terms. Tables 12 and 13 list the F-Measure values of linear SVM for each category and each attribute in SourceForge and Sharejar. It is clear that there is a difference between classes and the other types of attributes for each category; the values of terms, packages and methods outperform the values of classes for each category. One explanation for this result is that the packages and methods are more specific to the categories than classes, and that terms are more specific than packages. It means that some attributes should appear only in the applications that are grouped in a category, or these attributes should not appear across several categories because they do not provide enough information to the classifier to distinguish between the categories. To explore this, we computed the number of attributes with top values of EEL that are overlapped in the categories and plotted the average values in Figure 6.

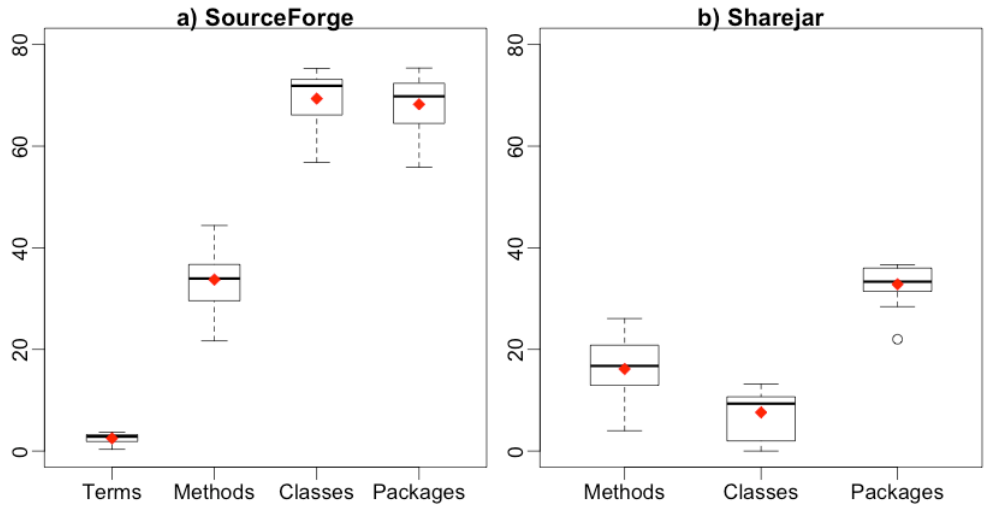

Figure 6. Averages values of overlapped top-100 EEL attributes in categories of both repositories

For SourceForge, terms extracted from source code is the attribute with the lowest number of attributes shared between categories and classes is the one with the highest number of overlapped attributes; thus identifiers in methods and variables are specific to categories with a few terms that are common to the categories, and classes are less specific. For example, the `Email` category has an average of 3.71 overlapped terms with other categories, such as `hashcode`, `locally`, `traffic`, `etched`, `crif`, and `timezone`. The `hashcode` term is also a top EEL attribute in `Networking`, `Office/Business`, and `Security` categories; `locally` is also in `Post-Office`, `Office/Business`, and `Networking`; `etched` is also in `Database`, `Frameworks`, `Games`, `Internet`, and `Networking`; `crif` is also in `Communications`, `Frameworks`, and `Security`. The Office /Business category has an average of 75.29 classes shared with other categories; the classes `javax::swing::plaf::basic::BasicTableUI::Handler`, `com::sun::java::util::jar::pack::ClassReader`,`com::sun::org::apache::xalan::internal::xsltc::compiler::IdKeyPattern`, and `sun::misc::ServiceConfigurationError` are overlapped with the other 21 categories. For packages the situation is similar to classes, the average number of overlapped attributes for each category is slightly lower than classes in most of the categories; however the accuracy of packages outperforms classes. We conjecture that the accuracy difference is due to the fact that the variety of overlapped packages is lower than classes (e.g., the number of unique packages overlapped with other categories is 83, and the number of unique classes overlapped with other categories is 87).

For Sharejar, linear SVM with methods provided the highest values of accuracy for each category, and packages provided the lowest values. Thus, identifiers extracted from methods, and classes are more specific than packages for categories in Sharejar applications. Again, to explain these results we computed the average number of overlapped top-100 EEL attributes by category (Figure 6). Packages with top EEL values are more overlapped than the other attributes, and classes are the attributes with the lowest overlapping. For example the localization category has an average of 36.67 overlapped packages such as, `javax.microedition.rms`, `javax.microedition. midlet`, `javax.microedition.io`, `javax.microedition.lcdui`; 21 of 46 packages are overlapped with the other 12 categories, and 8 packages are overlapped with 11 categories. The reason is that importing these packages provides access to features (midlets, foundation profiles, CLDC)[18] that are required or mandatory to implement J2ME applications. Therefore, the extensions of the J2ME API have the packages that could provide the

[18] http://www.oracle.com/technetwork/java/javame/javamobile/documentation/index.html (verified on 05/07/2012)

Table 13. F-Measure of linear SVM for each category and each attribute type in Sharejar

Category	F-Measure		
	Methods	**Classes**	**Packages**
1. Chat & SMS	0.597	0.592	0.539
2. Dictionaries	0.354	0.354	0.089
3. Education	0.405	0.400	0.299
4. Free Time	0.420	0.423	0.310
5. Internet	0.485	0.479	0.406
6. Localization	0.507	0.522	0.462
7. Messengers	0.346	0.341	0.250
8. Music	0.598	0.554	0.490
9. Science	0.409	0.409	0.188
10. Utilities	0.487	0.482	0.365
11. Emulators	0.370	0.374	0.330
12. Programming	0.323	0.323	0.200
13. Sports	0.458	0.458	0.297

information to the classifiers to distinguish between the Sharejar categories. However, these packages were designed to provide low level functions in most of the cases or are related to specific deployment environments (e.g., com.motorola.io, com.siemens.mp.io), and using these packages is not enough to describe application domains. For methods and classes, the overlap is lower, and classes and terms extracted from methods are more specific for the categories. However, we conjecture that the APIs' design at methods and class levels are not as good as those in JDK to be pertinent descriptors of application domains.

6.2 Expected Entropy Loss and Term-based/API-based attributes

According to EEL definition (Section 3), attributes with higher values of EEL are more discriminatory and provide more information for the categorization. Therefore, the higher the values of EEL in the attributes selected to train the classifiers, the higher the accuracy of the categorization because the classifiers are able to better distinguish among different categories. We explored this relation, using the distribution of top attributes in both repositories. Figures 7 and 8 show average values of EEL of the top-100 attributes in both SourceForge and Sharejar. Additionally, Figures 9 and 10 show the distribution of EEL values for top-100 attributes. In some categories the number of top terms is less than 100, because the number of packages or classes in the category is less than 100. The distributions of EEL values for top attributes, in both repositories, are skewed to the left and not unimodal. This confirms that some attributes are more descriptive for some categories and many others are commonly shared by some categories. The multi-modal behavior is more evident in the case of terms (SourceForge) and methods (Sharejar). API methods are the attributes in Sharejar with the highest average value of EEL and lowest variance for every category (Figure 8). Top identifiers in terms (SourceForge) also have the highest average of EEL and the lowest variance for each category (Figure 7); however, average values for SourceForge are lower than Sharejar. This is because the number of applications and categories in Sharejar are lower than in SourceForge, and Sharejar applications use APIs with features related to mobile devices. API calls in Sharejar are more specific that API calls in SourceForge.

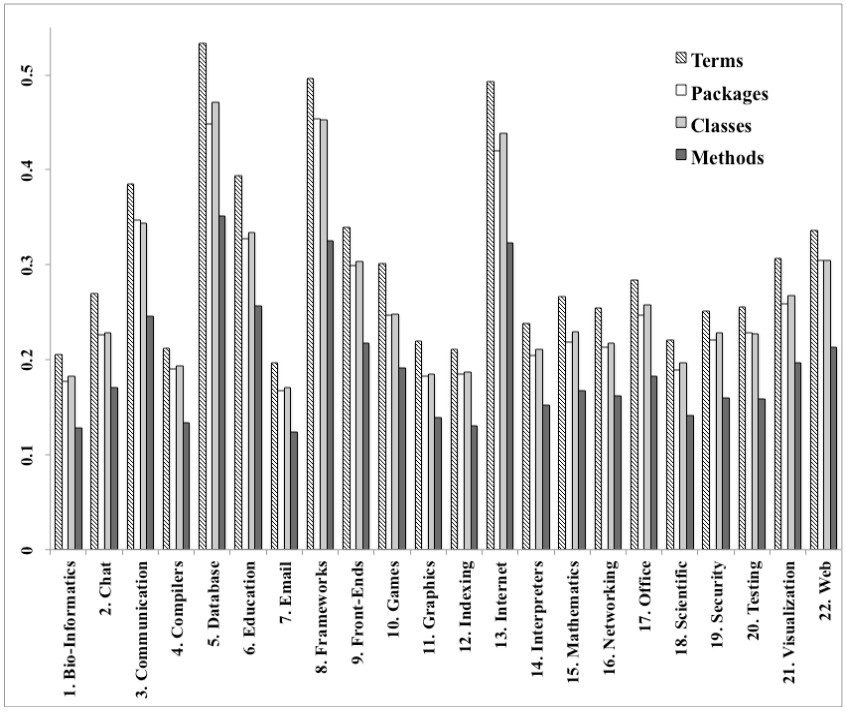

Figure 7. Average EEL of the 100 attributes with the highest EEL values in all categories of SourceForge.

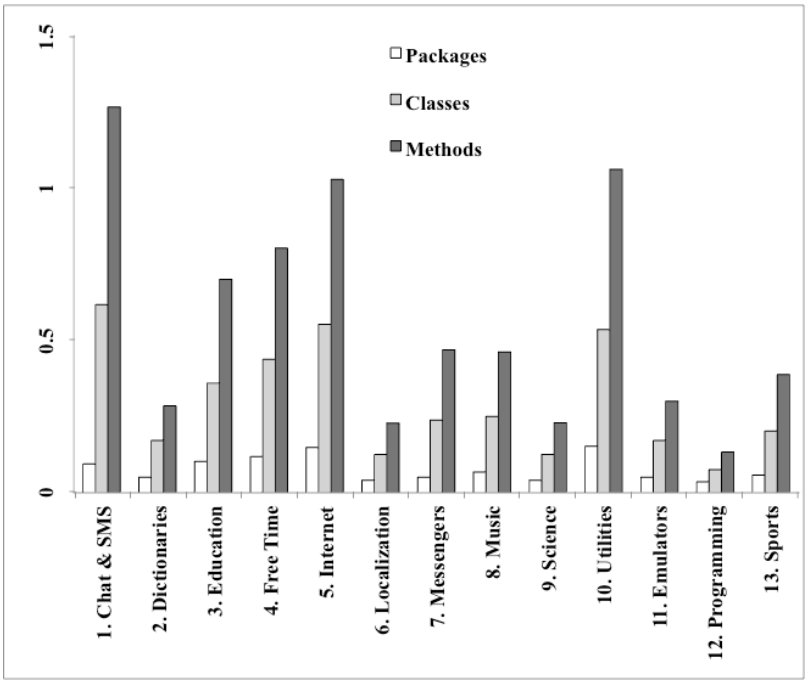

Figure 8. Average EEL of the 100 attributes with the highest EEL values in all categories of Sharejar.

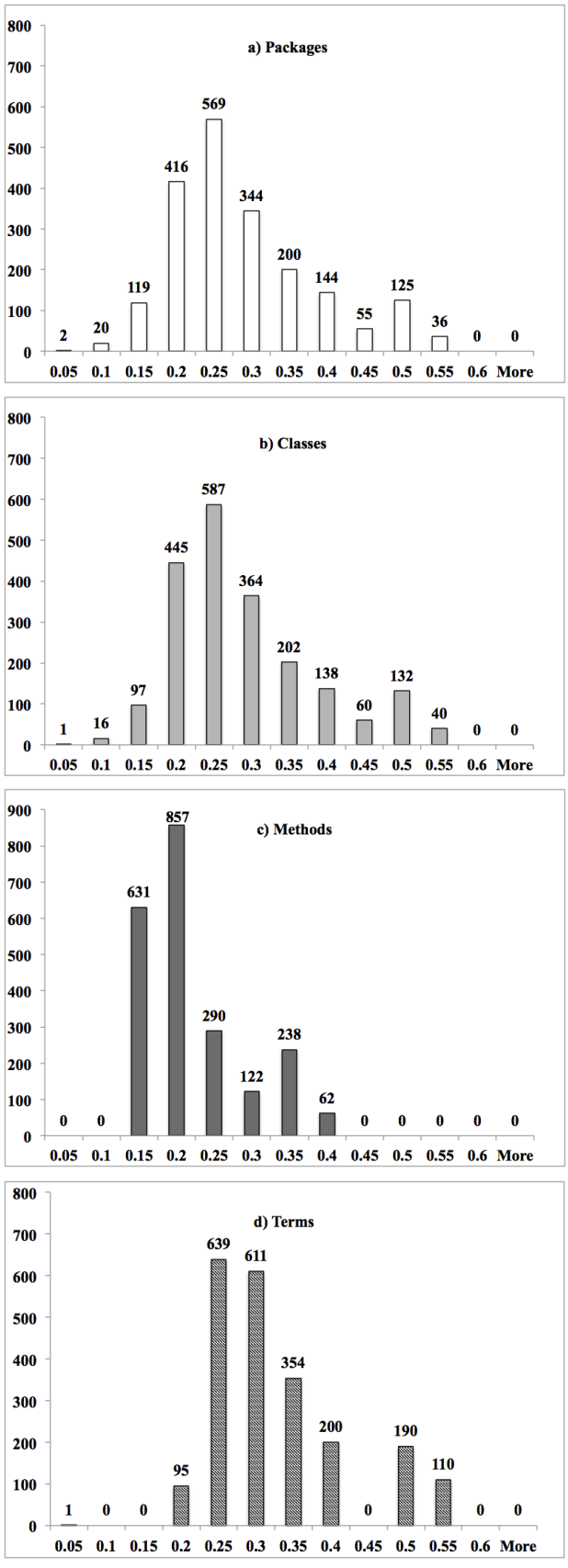

Figure 9. Histogram of EEL for top attributes in SourceForge

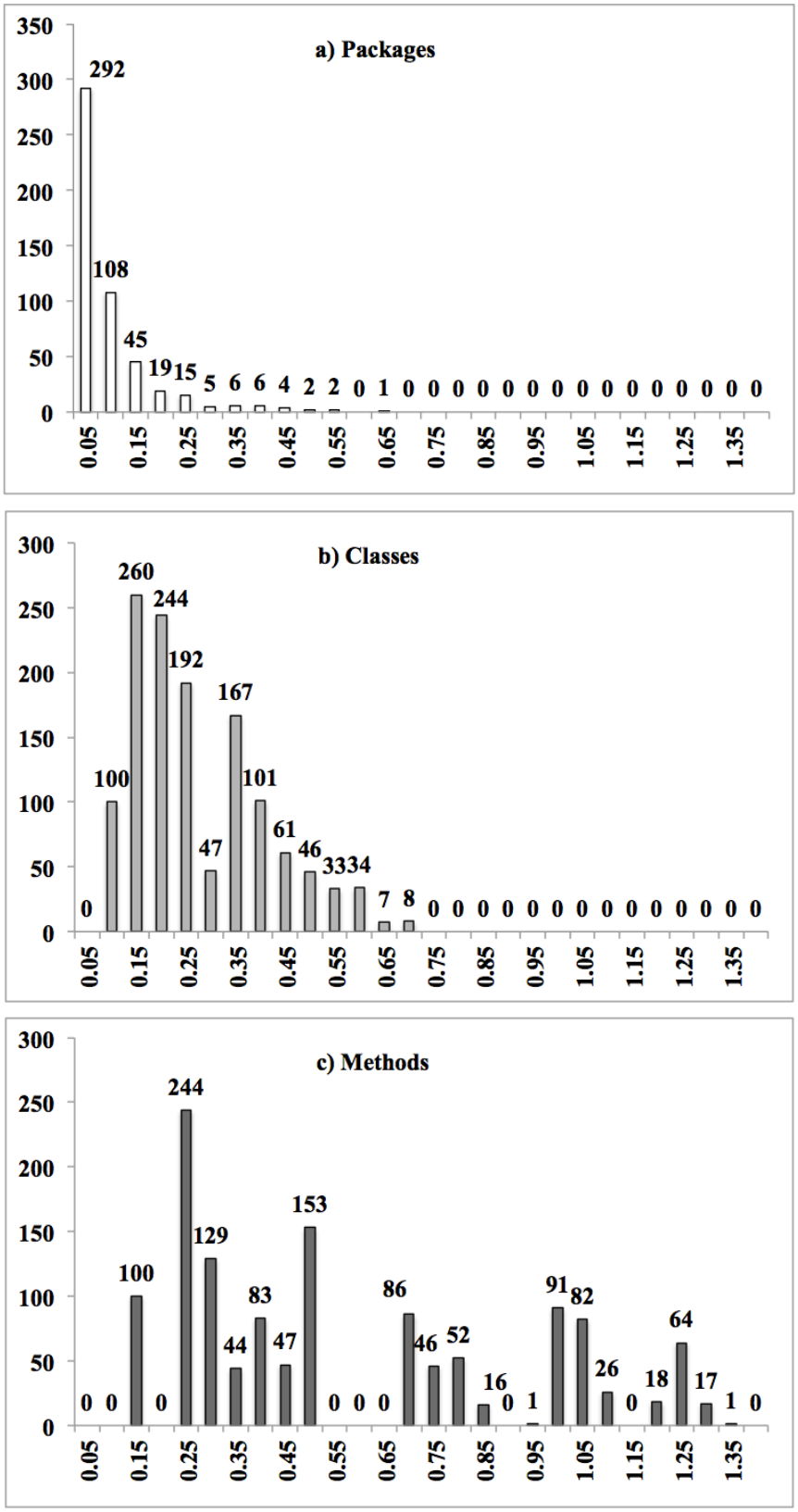

Figure 10. Histogram of EEL for top attributes in Sharejar

Figure 9 shows multi-modal distributions for the top EEL values in SourceForge. The distribution of methods has EEL values that are lower than EEL values of terms, and terms in SourceForge are the best option for categorization because they provide more information and are more related to categories. The percentage of attributes, in the terms dataset, with EEL higher or equals than 0.5 is higher than in the other datasets (Terms = 12%, API packages = 7%, API classes = 8%, API classes = 0%). Top attributes in classes and packages have similar distributions, and effect sizes for PRC, REC and F-Measure are lower than 0.35 (Table 9). However, the effect size for FPR is close to 0.6, and we guess that the small differences (e.g., number of attributes with EEL equal to 0.45, 0.5, 0,55) between the distributions for the top EEL values in classes and packages, contributes to this effect size.

Figure 10 shows a clear difference between packages and classes in Sharejar. API classes have higher EEL values than API packages, and according to effect sizes (Table 8) the probabilities of getting PRC, REC, and F-Measure values using API classes, are greater than the performance values obtained with API packages. Therefore, classes in Sharejar could provide better performance than packages. This case differs from SourceForge results, and the reason is that the packages in Sharejar are not enough to describe features of categories, meanwhile the diversity of classes provide more information to the categorization (Table 4, Section 4.1.2).

6.3 Applications and Future work

The best results were achieved using SVM with a linear kernel on API-terms of the SourceForge datasets [19]. Our API-based categorization provided REC and FPR values that outperform the best values reported by Ugurel et al. (Ugurel et al. 2002). Figure 11 depicts the distribution of REC and FPR of our API-based results on SourceForge dataset, and the distribution of REC and FPR values of the single words-based approach (SW) reported in (Ugurel et al. 2002). API-classes provided the worst average values, however, the average and medians of the other API-based approaches (terms, methods, and packages) outperform SW, using a higher number of applications and categories. This leads us to believe that our API-based categorization could be used as a model to recommend the categories that could be assigned to software applications. For example, a developer uploading a new system to SourceForge can use the list of categories provided by our approach as a recommendation list that is based on the knowledge extracted from previous selections of other developers; then, according to the agreement of the developer with the recommendations, the final selection could be used as new knowledge in a further training of the classifier. If the developer agrees with the list, the model will be reinforced, it not new knowledge will provide more information to the classifier to avoid the overfitting of the model.

Our results shed additional light on how categorizing software applications can be useful for software maintenance. Di Lucca et al. use automatic classification of software maintenance requests to route them to specialized maintenance teams (Di Lucca et al. 2002). With our approach, these requests can be mapped to application categories, and then similar requests and solutions can be located in these categories enabling stakeholders to address maintenance requests faster and within budget (Weiss et al. 2007). Additionally, extending the work of Anvik and Murphy (Anvik and Murphy 2011), where implementation expertise of developers is inferred from bug reports, our approach can complement this work by classifying expertise of developers by categories of applications with which they deal.

Future work should be dedicated to explore better methods for automatic categorization that improve our results. In spite the fact that our results outperform the best values reported by Ugurel et al. (Ugurel et al. 2002), REC and PRC values are not higher than 60% on average. A possible explanation for these rather low values is that several attributes are shared across different categories (Sections 6.1 and 6.2) and our approach to categorize multi-label datasets using single–label classifiers is not able to identify the relationships in those attributes. A single-label classifier learns how a subset of attributes allows identifying a category from the rest, but does not identify how those attributes are used to identify other categories. According to (Ji et al. 2008), there are shared spaces between attributes in different categories that exhibit semantic correlation; therefore, it is essential to exploit the correlation information between categories to provide high performance values in categorization processes using multi-label classifiers.

[19] 0.671 (Average PREC), 0.671 (Average REC), 0.671 (Average F-measure), 0.024 (Average FPR)

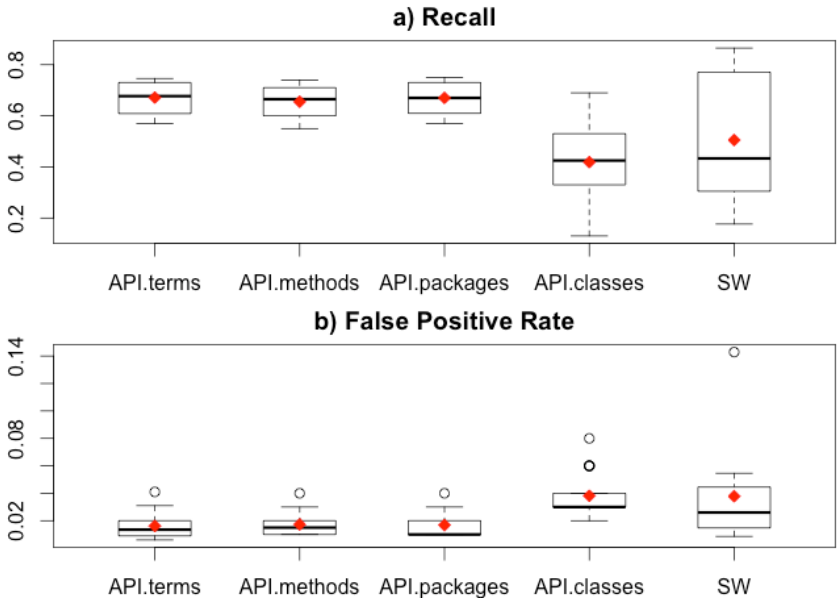

Figure 11. Boxplot of performance metrics (REC and FPR) of API-based categorization and Single Words-based categorization

We used multi-class algorithms to categorize multi-label datasets; we transformed the multi-label datasets into multi-class ones creating new categories. This process does not take into account the correlation between the categories loosing information that is useful for categorization of multi-label datasets. We guess that using multi-label algorithms could increase the TPR and PRC in our results. Therefore our future work consists in comparing multi-label algorithms with SVM using different techniques for attributes selection and the same approach for extracting attribute in closed-source repositories. Additionally, topic-modeling techniques has been used in several researches related to maintenance tasks; one additional research question that we want to address as future work is whether topics extracted with LDA can be more effective for selecting descriptive attributes as compared to EEL.

7 Threats to Validity

Certain threats to validity affect the results of our case study and our ability to generalize these results. Internal threats include the attributes we use for categorization. The terms that programmers write in source code may be arbitrary, and the existence of a term in a project may be coincidental. The API classes and packages are less likely to occur coincidentally, because APIs provide functionality that the programmer wanted to use. In this case, the TPR, FPR, and PRC we report from the terms could be too high or low as compared to classes or packages. We minimize this threat by using 5-fold cross validation.

Another internal threat to validity is the set of categories we use. For SourceForge, our approach considers only top-level categories with more than 100 applications (see Section 4.1). We do not explore why these categories are the largest, and our results could be affected by certain "popular" categories: applications may be more likely to occur in these categories purely by chance. We minimized this threat by including all the categories from Sharejar, although we compute the TPR, FPR and PRC separately. That is, an application from Sharejar cannot be placed into a category from SourceForge.

One external threat to validity is our choice of repositories. Further work is needed to reproduce our case study on other datasets, and we cannot guarantee that our results will apply to all possible software repositories. We minimized this threat in two ways: First, we used two different repositories. Second, we duplicated the case study design from previous work (Ugurel et al. 2002), and found comparable results. The fact that both repositories are in Java also introduces

a threat to validity. We use API packages, classes and methods, but other programming languages may not have a similar hierarchical organization of APIs.

Finally, there is a threat that applications are incorrectly assigned to categories in subject repositories. It means that a training set may be compromised, and it is very difficult to determine it with any certainty. For example, registration process of new applications in SourceForge includes the process of manually categorizing the projects using the *Trove Software Map*[20]*(TSM)*. The TSM has categories that allow developers to categorize applications using domain topics, programming language, operating systems, among others. However, the topics predefined in TSM do not represent all the possible domains, and in some cases developers select categories that do not describe as the features provided by the applications. If this is the case, then all approaches introduce a similar level of imprecision, and a relative comparison of these approaches may still be valid.

8 Related Work

Machine learning has previously been used to categorize software systems. Kawaguchi et al. (Kawaguchi et al. 2003) use a Decision Tree on 41 software systems from SourceForge that were categorized manually by developers into six groups (board game, compiler, database, editor, video conversion, and xterm) . Their model uses 3-gram representations of filenames as attributes. Although, we could not compare our approach to the Decision Tree based model proposed in (Kawaguchi et al. 2003) because we extract terms and API calls from applications, we use the same Decision Tree algorithm with our attributes selection strategy. Sandhu et al. (Sandhu et al. 2007) propose an unsupervised approach using Naïve Bayes classification and a hybrid model of Naïve Bayes with LSA; the model categorize 63 components belonging to six categories or domains that were defined manually. Therefore we cannot compare our model to the one proposed in (Sandhu et al. 2007) because ours is supervised and we use 4,031 applications (745 from Sharejar and 3,286 from SourceForge) that are categorized in a predefined list of categories.

The work by Ugurel et al. is the most similar to ours in that we use supervised machine learning techniques (Ugurel et al. 2002). We have replicated Ugurel's study in this paper and compared our approach to it on a large repository of open-source projects. Ugurel et al. uses a SVM implementation for programming language and application topic classification of open-source systems using single labels. Their model includes feature selection with EEL and categorization with SVM. We expanded this work by evaluating multiple machine learning algorithms and types of attributes with multiple categories.

MUDABlue is an information retrieval technique for software categorization (Kawaguchi et al. 2006). MUDABlue uses Latent Semantic Indexing (LSI) and clustering for automatic software categorization of 41 programs selected from SourceForge. MUDABlue uses identifiers as features. Unlike our approach, MUDABlue automatically generates categories based on these features instead of placing projects into existing categories. Therefore, we could not directly compare our approach to MUDABlue.

LACT is another system that relies on information retrieval to categorize software (Tian et al. 2009). LACT uses Latent Dirichlet Allocation (LDA) over the same dataset as Kawaguchi et al. in order to infer topics to which applications belong. Like MUDABlue, LACT automatically generates categories for projects, meaning that we could not compare our approach to LACT.

Kelly et al. (Kelly et al. 2011) use topic modeling techniques (LDA) to semi-automatically identify common topics from the source code of software applications. Like LACT, Kelly et al. use topics clustering to identify commonalities and variability across the applications, and topics are extracted as collections of frequent words in the clusters.

Bruno et al. (Bruno et al. 2005) propose an approach for locating web services. Their approach takes a natural language query and uses SVM to match the query to related web services. Also, Bruno et al. find relationships among web services via automatic categorization. Their approach uses words as attributes. These words come from any documentation of the web service. In principle, our approach is similar in that we test SVM and words for categorization, though we also perform a case study with many machine-learning algorithms with APIs as attributes.

[20] http://sourceforge.net/apps/trac/sourceforge/wiki/Software%20Map%20and%20Trove (verified on 05/07/2012)

Categorization has previously been used with other software artifacts in order to achieve some tasks related to software maintenance and evolution. Menzies et al. (Menzies and Marcus 2008) present an automated method named SEVERIS, for assigning severity levels to defect reports. SEVERIS extracts words from issues reports and selects most relevant by using a measure of information gain (InfoGain). SEVERIS build rules set between the terms and the severity levels (categories) in order to assign the severity of new reports, which is different from our approach in that we aim to categorize whole applications.

Antoniol et al. (Antoniol et al. 2008) use machine learning classifiers in order to categorize descriptions of "issues" posted in bug tracking systems. The objective of categorization is to classify issues into types of activities (e.g., bug fixing, feature enhancement, etc.). Issues are modeled using words as attributes. Antoniol et al. use three different machine learning algorithms: logistic regression, Naïve Bayes and Decision Trees. Unlike our approach, their technique focuses on categorizing issues in applications.

Hindle et al. (Hindle et al. 2009) propose to use machine learning for categorizing commits (e.g., from CVS) into categories of maintenance tasks (e.g., corrective, adaptive, etc.). The words in the commit messages are used as sets of attributes. Hindle et al. use seven classifiers for the categorization: J.48, Naïve Bayes, SMO, KStar, IBK, JRip and ZeroR. They performed an evaluation of these algorithms, but unlike this paper, only used one type of attribute.

Schuler et al. (Schuler et al. 2007) proposed a dynamic birthmark generation technique for software applications. The birthmark is built as a set of short call sequences received by API objects. The approach could be considered as a type of categorization, because applications with the same birthmark are likely to share a common origin. Thus a birthmark is a label (category) that identifies intrinsic properties of executable files that are hard to modify but easy to validate. Like our approach, their technique uses an API-based approach, collecting sequences of method calls from the Java Platform Standard API on jar files. However, dynamic birthmark generation technique requires executing application, which would be prohibitive in the context of our problem setting.

Our work is related to Exemplar, a search engine that locates relevant applications (Grechanik et al. 2010) in that Exemplar matches query keywords to words in the documentation of API calls. Although, Exemplar does not categorize software, it also explores the idea that applications contain functional abstractions in a form of API calls whose semantics are defined precisely. Based on this, Exemplar is a code search engine that identifies API calls in software applications and uses the documentation of the APIs to retrieve relevant applications. Similarly, *Structural Semantic Indexing* (SSI) is a technique for computing the similarity between source code based on API calls, however, it is used to locate source code using queries, not to categorize software (Bajracharya et al. 2010). As in the case of Exemplar and our study, SSI is based on the heuristic that source code entities (classes, methods, etc.) that use APIs in similar ways, are semantically related because they do similar things.

9 Conclusions

We present an approach for categorizing software applications in the context of maintenance tasks. We extracted the APIs used by applications as attributes for categorization. Our technique differs from previous approaches in that we do not rely on words extracted from the source code of applications, meaning that we can support software maintenance tasks over both open- and closed-source repositories. We built and tested our ideas with five different machine-learning algorithms (SVM, Naïve Bayes, Decision Tree, JRIP, and IBK) and two software repositories, and compared our approach to the closest competing technique. We found that using API methods and packages provided as good accuracy as using terms, even though the number of API packages and methods is smaller than the number of terms. One key advantage to using API packages, classes, and methods as attributes is that these attributes are more stable than terms across many programs. Recent work has found that APIs are more likely to represent domain concepts in applications than terms (Ratiu and Deissenboeck 2006; Ratiu and Deissenboeck 2007). Hence, APIs are likely to be high-quality attributes for categorization, even if terms are not.

Also, we found that our approach is applicable to repositories where the terms in the source code are not available. Ours is the first study that thoroughly evaluated different machine learning algorithms and types of attributes for the purposes of software categorization. The average accuracy (F-measure, precision, recall) of the best configuration of our technique is around 60%, suggesting that developers can use it to obtain a list of possible domain categories, which software applications could be assigned to, even when the source code is not available. Knowing these

categories could be used to predicting problems and quality attributes, or extracting related bugs or features.

Acknowledgements

We are grateful to anonymous EMSE and ICSM'11 reviewers for their relevant detailed comments and suggestions, which helped us in significantly improving the initial version of this paper. We also would like to thank Marty White from the College of William and Mary for his useful suggestions that we implemented in this paper. This work is supported in part by NSF CCF-1016868, CCF-0916260, CCF-0916139, CCF-1017633, CCF-1218129 and Accenture. Any opinions, findings and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

References

Aha D. W., Kibler D. and Albert M. K. (1991) Instance-Based Learning Algorithms. Machine Learning **6**: 37-66.

Alpaydin E. (2010) Introduction to Machine Learning, The MIT Press.

Antoniol G., Ayari K., Di Penta M., Khomh F. and Guéhéneuc Y.-G. (2008) Is it a bug or an enhancement?: a text-based approach to classify change requests. 18th Conference of the Centre for Advanced Studies on Collaborative Research Meeting of Minds (CASCON'08), Ontario, Canada, 304-318.

Anvik J., Hiew L. and Murphy G. C. (2006) Who should fix this bug? 28th International Conference on Software Engineering (ICSE'06), 361-370.

Anvik J. and Murphy G. C. (2011) Reducing the Effort of Bug Report Triage: Recommenders for Development-oriented Decisions ACM Transactions on Software Engineering and Methods **20**(3).

Bajracharya S., Ossher J. and Lopes C. V. (2010) Leveraging Usage Similarity for Effective Retrieval of Examples in Code Repositories. 18th International Symposium on the Foundations of Software Engineering (FSE'10).

Bruno M., Canfora G., Di Penta M. and Scognamiglio R. (2005) An Approach to support Web Service Classification and Annotation. IEEE International Conference on e-Technology, e-Commerce and e-Services (EEE'05), 138 - 143

Bugde S., Nagappan N., Rajamani S. and Ramalingam G. (2008) Global Software Servicing: Observational Experiences at Microsoft. 2008 IEEE International Conference on Global Software Engineering (ICGSE '08). , 182-191.

Cohen W. W. (1995) Fast Effective Rule Induction. 12th International Conference on Machine Learning, 115-123.

Crammer K. and Singer Y. (2003) A Family of Additive Online Algorithms for Category Ranking. Journal of Machine Learning Research **3**(6): 1025-1058.

de Carvalho A. C. P. L. F. and Freitas A. A. (2009) A Tutorial on Multi-label Classification Techniques. Foundations of Computational Intelligence. A. Abraham, A.-E. Hassanien and V. Snásel, Springer-Verlag. **5**.

Demsar J. (2006) Statistical Comparisons of Classifiers over Multiple Data Sets. Journal of Machine Learning Research **7**: 1-30.

Di Lucca G. A., Di Penta M. and Gradara S. (2002) An Approach to Classify Software Maintenance Requests. IEEE International Conference on Software Maintenance (ICSM'02), Montréal, Québec, Canada, 93-102.

Dit B., Guerrouj L., Poshyvanyk D. and Antoniol G. (2011) Can Better Identifier Splitting Techniques Help Feature Location? 19th IEEE International Conference on Program Comprehension (ICPC'11), Kingston, Ontario, Canada, 11-20.

Dumitru H., Gibiec M., Hariri N., Cleland-Huang J., Mobasher B., Castro-Herrera C. and Mirakhorli M. (2011) On-demand Feature Recommendations Derived from Mining Public Product Descriptions. 33rd IEEE/ACM International Conference on Software Engineering (ICSE'11), Honolulu, Hawaii, USA, 181-190.

Feng C.-X. J., Yu Z.-G. S., Emanuel J. T., Li P.-G., Shao X.-Y. and Wang Z.-H. (2008) Threefold versus fivefold cross-validation and individual versus average data in predictive

regression modelling of machining experimental data. International Journal of Computer Integrated Manufacturing **21**(6): 702-714.

Frakes W., Prieto-Diaz R. and Fox C. (1998) DARE: Domain analysis and reuse environment. Annals of Software Engineering **5**: 125-141.

Grechanik M., Csallner C., Fu C. and Xie Q. (2010) Is Data Privacy Always Good For Software Testing? 21st IEEE International Symposium on Software Reliability Engineering (ISSRE'10), San Jose, California, USA, 368-377.

Grechanik M., Fu C., Xie Q., McMillan C., Poshyvanyk D. and Cumby C. (2010) A Search Engine For Finding Highly Relevant Applications. 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10), Cape Town, South Africa, 475-484.

Grechanik M., McMillan C., DeFerrari L., Comi M., Crespi S., Poshyvanyk D., Fu C., Xie Q. and Ghezzi C. (2010) An Empirical Investigation into a Large-Scale Java Open Source Code Repository. 4th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '10), Bolzano-Bozen, Italy.

Grissom, R. J., & Kim, J. J. (2012) Effect sizes for research: Univariate and multivariate applications. (2nd ed.). New York, NY: Taylor & Francis.

Guyon I. and Elisseeff A. (2003) An Introduction to Variable and Feature Selection. Journal of Machine Learning Research **3**: 1157-1182.

Hindle A., Germán D. M., Godfrey M. W. and Holt R. C. (2009) Automatic Classification of Large Changes into Maintenance Categories. 17th IEEE International Conference on Program Comprehension (ICPC'09), Vancouver, Canada, 30-39.

Hsu C. and Lin C. (2002) A Comparison of Methods for Multiclass Support Vector Machines. IEEE Transactions on Neural Networks **13**(2): 415-425.

Ji S., Tang L., Yu S., Ye J. (2008) Extracting Shared Subspace for Multi-label Classification. 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'08), Las Vegas, Nevada, USA, 381-389.

Jones C. (2010) Software Engineering Best Practices. New York, NY, McGraw-Hill.

Kang K. C., Cohen S., Hess J., Novak W. and Peterson A. (1990). Feature-oriented domain analysis (FODA) feasibility study Pittsburgh, Pennsylvania, USA, Carnegie Mellon University, Software Engineering Institute

Kawaguchi S., Garg P. K., Matsushita M. and Inoue K. (2003) Automatic Categorization Algorithm for Evolvable Software Archive. 6th International Workshop on Principles of Software Evolution (IWPSE'03), 195-200.

Kawaguchi S., Garg P. K., Matsushita M. and Inoue K. (2006) MUDABlue: An automatic categorization system for Open Source repositories. Journal of Systems and Software **79**(7): 939-953.

Kelly M. B., Alexander J. S., Adams B. and Hassan A. E. (2011) Recovering a Balanced Overview of Topics in a Software Domain. 11th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'11), Williamsburg, VA, USA, to appear.

Leopold E. and Kindermann J. (2002) Text Categorization with Support Vector Machines. How to Represent Texts in Input Space ? Machine Learning **46**(1): 423-444.

Lorena A. C. and De Carvalho A. C. P. L. F. (2004) Comparing Techniques for Multiclass Classification Using Binary SVM Predictors. Third Mexican International Conference on Artificial Intelligence (MICAI'04), Mexico City, Mexico, Springer, 272-281.

McMillan C., Grechanik M., Poshyvanyk D., Xie Q. and Fu C. (2011) Portfolio: Finding Relevant Functions And Their Usages. 33rd IEEE/ACM International Conference on Software Engineering (ICSE'11), Honolulu, Hawaii, USA, 111-120.

McMillan C., Linares-Vásquez M., Poshyvanyk D. and Grechanik M. (2011) Categorizing Software Applications for Maintenance. 27th IEEE International Conference on Software Maintenance (ICSM'11), Williamsburg, Virginia, USA, 343-352.

Menzies T. and Marcus A. (2008) Automated Severity Assessment of Software Defect Reports. IEEE International Conference on Software Maintenance (ICSM'08), Beijing, China, 346-355.

Poshyvanyk D. and Grechanik M. (2009) Creating and Evolving Software by Searching, Selecting and Synthesizing Relevant Source Code. 31st IEEE/ACM International Conference on Software Engineering (ICSE'09), Vancouver, British Columbia, Canada, 283-286.

Prieto-Diaz R. (1990) Domain Analysis: An Introduction. ACM SIGSOFT Software Engineering Notes **15**(2): 47-54.

Ratiu D. and Deissenboeck F. (2006) How Programs Represent Reality (and How They Don't). 13th Working Conference on Reverse Engineering (WCRE'06), 83 - 92.

Ratiu D. and Deissenboeck F. (2007) From Reality to Programs and (Not Quite) Back Again. 15th IEEE International Conference on Program Comprehension (ICPC'07), Banff, Alberta, Canada, 91-102.

Sandhu P. S., Singh J. and Singh H. (2007) Approaches for Categorization of Reusable Software Components. Journal of Computer Science **3**(5): 266-273.

Schuler D., Dallmeir V. and Lindig C. (2007) A dynamic birthmark for java. Twenty-second IEEE/ACM International Conference on Automated software Engineering (ASE 2007), Atlanta, Georgia, USA, 274-283.

Sim S. E., Umarji M., Ratanotayanon S. and Lopes C. V. (2011) How Well Do Search Engines Support Code Retrieval on the Web? ACM Transactions on Software Engineering and Methodology (TOSEM) **21**(1).

Tian K., Revelle M. and Poshyvanyk D. (2009) Using Latent Dirichlet Allocation for Automatic Categorization of Software. 6th IEEE Working Conference on Mining Software Repositories (MSR'09), Vancouver, British Columbia, Canada, 163-166.

Ugurel S., Krovetz R. and Giles C. L. (2002) What's the Code ? Automatic Classification of Source Code Archives. Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-2002), Edmontong, Alberta, Canada, 632-638.

Újházi B., Ferenc R., Poshyvanyk D. and Gyimóthy T. (2010) New Conceptual Coupling and Cohesion Metrics for Object-Oriented Systems. 10th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'10), Timişoara, Romania, 33-42.

Weiss C., Premraj R., Zimmermann T. and Zeller A. (2007) How Long Will It Take to Fix This Bug? 4th IEEE International Workshop on Mining Software Repositories (MSR'07), Minneapolis, MN, 1-8.

Zhang M.-L. and Zhou Z.-H. (2005) A k-nearest neighbor based algorithm for multi-label classification. IEEE International Conference on Granular Computing, Beijing, China, 718-721.

Zhang M.-L. and Zhou Z.-H. (2006) Multi-label Neural Networks with Applications to Functional Genomics and Text Categorization. IEEE Transactions on Knowledge and Data Engineering **18**(10): 1338-1351.

Zimmermann T., Nagappan N., Gall H., Giger E. and Murphy B. (2009) Cross-project Defect Prediction: A Large Scale Experiment on Data vs. Domain vs. Process. ESEC/SIGSOFT FSE 2009, Amsterdam, The Netherlands, 91-100.