Categorizing Software Applications for Maintenance

Collin McMillan[1], Mario Linares-Vásquez[2], Denys Poshyvanyk[1], Mark Grechanik[3]

[1]Department of Computer Science
The College of William and Mary
Williamsburg, Virginia, USA
{cmc, denys}@cs.wm.edu

[2]Department of Computer Science
Universidad Nacional de Colombia
Bogotá, Colombia
mlinaresv@unal.edu.co

[3]Accenture Technology Labs and
The University of Illinois at Chicago
Chicago, Illinois, USA
drmark@uic.edu

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. Unfortunately, for different legal and organizational reasons the source code is often not available, thus making it difficult to automatically categorize binary executables of software applications.*

In this paper, we propose a novel approach in which we use Application Programming Interface (API) calls from third-party libraries as attributes for automatic categorization of software applications that use these API calls. API calls can be extracted from source code and more importantly, from the byte-code of applications, thus making automatic categorization approaches applicable to closed source repositories. We evaluate our approach along with other machine learning algorithms for software categorization on two large Java repositories: an open-source repository containing 3,286 projects and a closed-source one with 745 applications. Our contribution is twofold: not only do 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, but also we carried out the first comprehensive empirical evaluation of automatic categorization approaches.

Index Terms— closed-source, open-source, software categorization, machine learning.

I. INTRODUCTION

Different software repositories 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 (i.e., units of functionality) allows stakeholders to decide what features they should implement in their own applications that belong to same groups or categories [7, 17] 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 [27, 29].

Automatic categorization of software applications in repositories is increasingly gaining acceptance since it reduces the manual effort significantly [4, 6, 7, 16, 17, 21, 24, 25]. 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, also called *attributes* of the applications, serve as the input to a machine learning algorithm that eventually 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.

A common key assumption for all existing automatic software categorization approaches is that the source code of open-source applications is always available. Unfortunately, it is often not true in case of commercial software development. Many organizations can release only the executable forms of their applications for various legal and organizational reasons. Furthermore, consulting companies such as Accenture, IBM, and HP Global Services do not own the source code that they produce – their clients do[1]. Therefore many commercial organizations cannot use the source code in software categorization approaches.

In reality, many consultants who build mission-critical software for different industries, especially for financial and biopharmaceutical companies, often work in "cleanrooms," where the source code is written and kept on company premises in physically secured environments [9, 15]. External consultants from outsourcing companies such as Accenture, IBM, and HP Global Services come to the cleanroom 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. Cleanroom development effectively negates the opportunity for consulting companies to accumulate

[1] Accenture policy 69 says 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.

knowledge about applications they build, and more importantly to use this knowledge for different software maintenance tasks.

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. Doing so is based on the fact that programmers typically build software using API calls from well-defined and widely used libraries [11]. The intuition behind our approach is that APIs are already grouped in packages and libraries based on their functionalities, and this grouping can be combined with 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. *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.*

In this paper, we investigate this question by empirically studying large sets of Java applications from different repositories and applying different machine learning algorithms to obtain the answer to this question with strong statistical significance. To our knowledge, this is the first time that different machine-learning approaches were thoroughly evaluated for software categorization on large application sets. All of our case study data is available at our online appendix[2]. Our contributions are the following.

- A new approach to software categorization based on the APIs used by the applications. We extracted the API information in two forms: as the API packages that contain calls used by applications and as the API classes. We found that using API packages results, on average, in 20% better predictions than using classes in terms of rate of true positives. Our approach is the first that is able to categorize applications in *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 contrasted the results of three different machine learning algorithms and three types of attribute, and show that *Support Vector Machines (SVM)* is the best-performing algorithm for categorization over these repositories.
- To demonstrate how competitive our approach is, we compared it with the closest baseline approach by Ugurel et al. [25] that has previously been tested on 330 applications from SourceForge and 1353 projects from IBiblio[5]. 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 whose calls are made in applications.

II. Background

In general, categorization is the task of assigning a finite set of *categories* to *software applications*. 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 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 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 the terms (i.e., the names of identifiers) are chosen by programmers of applications often arbitrarily, whereas the set of APIs is predefined.

A. Machine Learning Algorithms

In this paper, we focus on *supervised* machine learning algorithms, in which a *training set* of pre-categorized applications is used to predict the categories to which uncategorized applications belong. This paper deals with *multi-label* supervised learning, where an application is classified into *m* of *n* categories [23].

B. Attribute Selection

The attributes that are chosen to represent applications in classification are critical to producing quality predictions. Naturally, the attributes must be available and extractable from the software applications. In selecting attributes, we first choose which characteristics of the applications can be used as attributes that best distinguish the applications in each category. For example, words are a frequent choice of attribute used for categorization of text documents [4, 13,

[2] http://www.cs.wm.edu/semeru/catml/

[3] http://sharejar.com/

[4] http://sourceforge.net/

[5] http://www.ibiblio.org/

17, 25]. Also, the attributes must distinguish the applications in one category from the applications in a different category [28]. This selection is done automatically, during the training of the machine learning.

For example, selecting programming language keywords as attribute is a poor choice since all applications written in the same language are likely to share these keywords. Different strategies have been proposed for selecting the best subset of these attributes [12], though in general these strategies rely on the principle that certain attributes occur more often in applications that belong to certain categories. We use this principle in our approach – for example, APIs that come from a music library are likely to occur in application that processes music than in other types of applications.

III. Our Approach

Since a plethora of automatic categorization approaches use the same categorization process, this paper expands upon the work by Ugurel et al. [25] that is a popular baseline implementation 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 1353 from IBiblio). Our approach builds on this work by testing multiple types of attributes and algorithms. Another important different from previous studies is that our approach can be applied to closed-source repositories. To define the specifics of our approach that set it apart from other competitive approaches, we must answer the following four design questions.

Q1: 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.

Q2: What is an attribute?

Different approaches to software categorization use words from comments and identifiers as attributes that are extracted from the source code of 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 [25].

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 the source code. This necessity motivates us to select two more types of attributes from applications: API packages and API classes, whose API calls are invoked in applications. In this paper, we use these two types of attributes with different classification algorithms.

These two API-based attributes are both based on the API calls which applications use, but refer to different levels of granularity. API packages are one 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*. We use all packages regardless of how they are decomposed (e.g., a package containing another package). The API classes are the classes used in these packages and 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*. One important advantage to using packages and classes as attributes is that both of these can be extracted from Java byte code – it is not necessary to have the source code. The API packages and classes we detect in applications are from the Java SDK[6]. Given that the JDK is representative of the other third-party APIs, we expect our results to be general.

Q3: Which attributes do we use for categorization?

Since we use the baseline classification approach as *Expected Entropy Loss* (EEL) by Ugurel et al. [25], we briefly describe this approach here for reproducibility of our results. EEL is an almost decade old algorithm that is shown to be highly-effective for selecting the most-relevant attributes of systems in software repositories [25]. In EEL, words are selected from source code, and we adapt EEL as our attribute selector for API classes and packages 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.

We provide the following formulas for the reproducibility of our approach. Entropy 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:

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

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.

[6] http://www.oracle.com/technetwork/java/javase/downloads/

$$\Pr(C)=\frac{numberOf\ \mathrm{Re}\,latedAppli\ cations}{numberOfAp\ plications}$$

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

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

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

$$\begin{array}{lcc}
\Pr(\overline{a}) & = & 1-\Pr(a)\\
\Pr(C\mid a) & = & \dfrac{numberOf\ \mathrm{Re}\,latedApplicationsWithAttributeA}{numberOfApplicationsWithAttributeA}\\
\Pr(\overline{C}\mid a) & = & 1-\Pr(C\mid a)\\
\Pr(C\mid \overline{a}) & = & \dfrac{numberOf\ \mathrm{Re}\,latedApplicationsWithoutAttributeA}{numberOfApplicationsWithoutAttributeA}\\
\Pr(\overline{C}\mid \overline{a}) & = & 1-\Pr(C\mid \overline{a})
\end{array}$$

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

$$e(C)=-\Pr(C)\log\big(\Pr(C)\big)-\Pr(\overline{C})\log\big(\Pr(\overline{C})\big)$$

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

$$e_a(C)=-\Pr(C\mid a)\log\big(\Pr(C\mid a)\big)-\Pr(\overline{C}\mid a)\log\big(\Pr(\overline{C}\mid a)\big)$$

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

$$e_{\overline{a}}(C)=-\Pr(C\mid \overline{a})\log\big(\Pr(C\mid \overline{a})\big)-\Pr(\overline{C}\mid \overline{a})\log\Pr(\overline{C}\mid \overline{a})$$

Thus the *expected posterior entropy* is

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

And the *expected entropy loss* is

$$EEL(C,a)=e(C)-EPE(C,a)$$

Each attribute has a value for each category. Attributes with higher values of EEL for a category are more discriminatory and provide more information for the categorization. The attributes with highest EEL for each category are used to train categorization algorithms. In our case study, we select attributes only from the portion of the dataset used for training (see Section IV.B).

Q4: What machine learning algorithm do we use?

In this paper, we explore three popular machine learning algorithms: *Decision Trees*, *Naïve Bayes*, and *Support Vector Machines*. We chose to test these three algorithms because they have been widely used in text categorization [22].

Decision Trees (DT) uses a "divide and conquer" strategy to split the problem space into subsets. A DT is modeled like a tree where the root and the nodes are questions, and the arcs between nodes are possible answers to the questions. The leaves of the three 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 features and the output is a category. DTs were used for software categorization by Kawaguchi et al. [16].

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-label 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. [21].

Support Vector Machines (SVM) split the problem space into two possible sets by finding a hyper-plane which maximizes the distance with the closest item of each subset. The function that splits the hyper-plane is known as the *kernel* function. In this paper, we arrange multiple SVM classifiers in a *one-against-one* strategy to allow multi-label classification. In the one-against-one approach, each classifier is trained to recognize two classes [14]. We generate the predictions by a vote of the predictions from the classifiers following the technique of Ugurel et al. [25].

IV. Case Study Design

In Section V, we consider three types of attributes (terms, classes, and packages) and uses EEL to select a subset of those attributes for categorization. We also outlined three 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.

A. Settings of the Case Study

The settings of the case study include the applications we want to categorize and the implementation details behind the machine learning algorithms. While implementing our approach, we used the same implementations of machine learning algorithms as in [25] to allow direct comparison of with our techniques.

1) Software Repositories

We downloaded 8,310 Java applications from SourceForge (open source), and these applications are spread across the 22 categories in Table 1. We also downloaded 745 Java applications from Sharejar (closed source), and these applications are created for mobile phones and include only the compiled Java bytecode. The 13 Sharejar categories are listed in Table 2.

In general, projects may belong to one or more categories in the same repository, but not in cross-repositories. All categories from both repositories do not include any sub-categories. Also, in Sourceforge, we

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 1: SourceForge 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		

Table 2: Sharejar projects and categories

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.

2) Selection of Attributes

We extracted the package and class attribute sets from the Sharejar applications using JClassInfo[7], a byte-code analysis tool for Java. For the SourceForge repository, we built a tool based on PMD[8] to extract the API packages and classes directly from the applications' source code. PMD also allowed us to extract project terms from the source code.

Once we had the terms, packages, and classes from each project, it was necessary to select the attributes to be used for categorization. We use EEL to rank attributes for each category from the training set (see Section III). In keeping with the case study design from [25], we then select the attributes which best distinguish each category with the following procedure. First, we exclude attributes that occur in only one project. Second, we exclude attributes that occur in less than 7.5% of the projects in a category. Finally, we choose the top 100 remaining attributes for each category according to the attributes' EEL for the category.

3) Machine Learning Algorithms

We used WEKA[9] to implement the three machine learning algorithms for our approach. For SVM, we used the Weka Libsvm wrapper[10]. In all cases, we used WEKA's default parameters, which were the same as were used in [25].

B. Research Questions

Our goal is to determine how we can best categorize software based on the APIs used in each application. Therefore, we address 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 classes or API packages, is more effective for software categorization?

RQ$_3$: Are the API classes or API packages as effective attributes as words (e.g., identifiers, comments) from source code for software categorization?

The rationale behind RQ$_1$ is that we want to compare different machine learning approaches on large application sets and obtain quantitative measurements of how well it categorizes applications. We also want to know how different types of attributes affect the accuracy of our approach. Specifically, we extracted two types of data based on the APIs used in applications, and in RQ$_2$ we want to study which of these types produces the best accuracy. Similarly, the purpose of RQ$_3$ is to compare our approach, where attributes are API classes and packages, to competitive approaches that use words extracted from source code as attributes.

To respond to our research questions, we compare 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 train the machine learning algorithm and tested against the other four fifths. There are five iterations, and each section is used as the training set once. We chose 5-fold validation to reduce the computation time of our results as compared to 10-fold validation; recent studies have shown no statistical difference in the results from reduced iterations in validation [8].

C. Metrics and Statistical Analyses

1) Accuracy Metrics

The output of the machine learning algorithms is a set of predictions about the mapping of applications to categories. We evaluate these predictions using two metrics: true positive rate (TPR) and false positive rate (FPR). These metrics have been widely used as accuracy measures for machine learning [26], including the case study by Ugurel et al. [25]. The formulas for these metrics are as follows:

$$TPR = \frac{TP}{TP+FN}; \quad FPR = \frac{FP}{FP+TN};$$

[7] http://jclassinfo.sourceforge.net/

[8] http://pmd.sourceforge.net/

[9] http://weka.sourceforge.net/

[10] http://www.csie.ntu.edu.tw/~cjlin/libsvm/

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). TPR measures the proportion of true positives over the total number of predictions. FPR measures the proportion of false positives over all predictions.

2) *Testing Statistical Significance*

Our goal in our research questions is to compare the TPR and FPR of different algorithms using different types of attributes. Recent work in evaluating machine learning algorithms has suggested the *Friedman test* with *Nemenyi's post-hoc procedure* to establish statistical significance [5]. The Friedman test is a non-parametric test for comparing the accuracy of *k* classifiers over *N* datasets. If the null hypothesis is rejected using the Friedman test for multiple classifiers, we use Nemenyi's test to compare pairs of classifiers.

V. Case Study Results

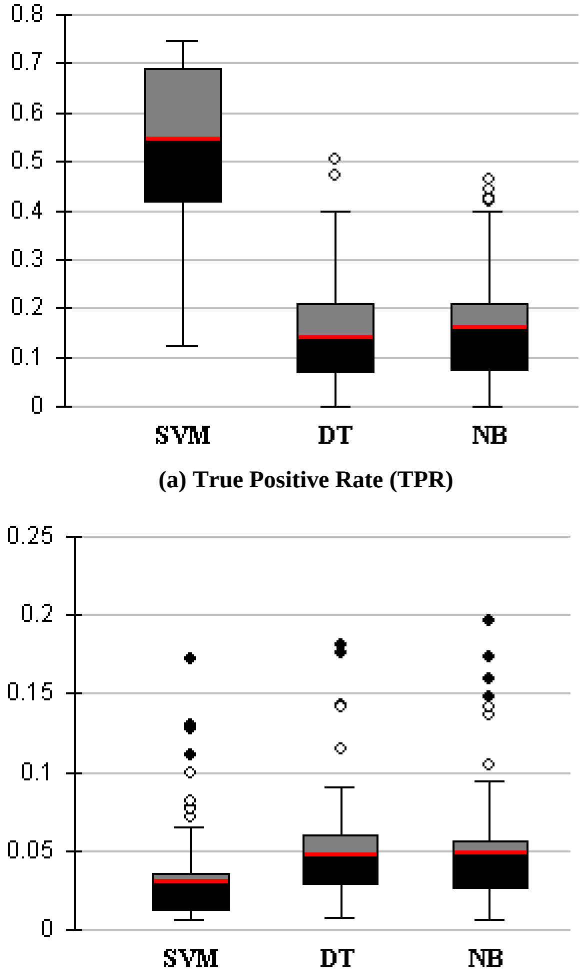
We conducted a 5-fold cross validation study using different configurations of type of attribute, repository, and machine learning algorithm. For every category in a repository, we calculate the TPR and FPR of the predictions for that category for a given configuration. Table 3 shows the configurations of our approach that we use to answer each research question.

Table 3: Configurations for our experiments. The last three 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
Sharejar	Classes	1, 2	1, 2	1, 2
Sharejar	Packages	1, 2, 3	1, 2, 3	1, 2, 3
Sourceforge	Classes	1, 2	1, 2	1, 2
Sourceforge	Packages	1, 2, 3	1, 2, 3	1, 2, 3
Sourceforge	Terms	1, 3	1, 3	1, 3

A. *RQ$_1$ – Machine Learning Algorithms*

Our approach relies on a supervised machine learning algorithm to interpret 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 [25]. In this paper, we contrast the results from three algorithms: SVM, DT, and NB.



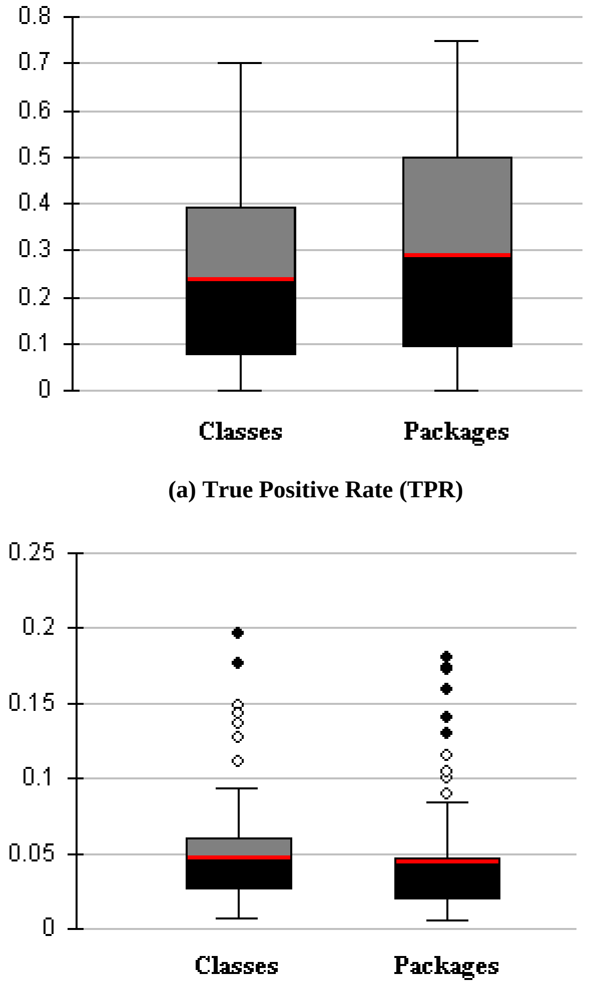
(a) True Positive Rate (TPR)

(b) False Positive Rate (FPR)

Figure 2. True and False positive rates for each algorithm over all types of attributes in all categories of both repositories. The red line is the median. The black box is the lower quartile. The gray box is the upper quartile. The thin line extends from the minimum to the maximum value.

Figure 2 shows a statistical summary of the TPR and FPR for our run of each algorithm. Each boxplot represents the TPR and FPR for one algorithm for each category in both Sourceforge and Sharejar, on all sets of attributes. We observe that the TPR for SVM is 54.53%, for DT is 14.11%, and for NB is 15.84%. The FPR for SVM is 3.11%, for DT is 4.76%, and for NB is 4.83%. We apply the Friedman test to test the statistical significance of the difference in these results. When testing TPR, the value of $Q_{critical}$ is 5.991, and at a 5% confidence level, $Q_{observed}$ equals 136.5. The value of p is less than 0.0001. When testing FPR, $Q_{critical}$ is 5.991, $Q_{observed}$ equals 129.6, and p is less than 0.0001. Therefore, we reject the null hypothesis that there is no significant difference of the values of TPR and FPR.

We applied Nemenyi's post-hoc test on the difference between specific pairs of algorithms with the following null hypotheses. We do not show any comparison of DT to NB because those algorithms performed less well than SVM.



(a) True Positive Rate (TPR)

(b) False Positive Rate (FPR)

Figure 3. True and False positive rates for classes and packages over three algorithms in all categories of both repositories.

H_0: There is no statistically-significant difference between the **TPR** of SVM and DT.

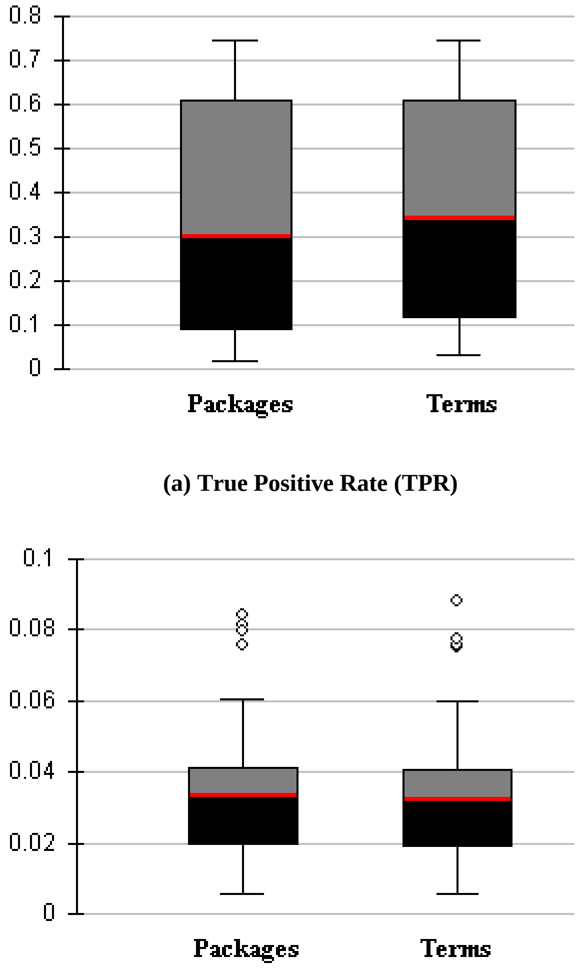
H_1: There is no statistically-significant difference between the **TPR** of SVM and NB.

H_2: There is no statistically-significant difference between the **FPR** of SVM and DT.

H_3: There is no statistically-significant difference between the **FPR** of SVM and NB.

Table 4. Nemenyi's test results RQ_1

H	$q_{critical}$	$q_{observed}$	Decision
H_0	26.59	140.5	Reject
H_1	26.59	132.5	Reject
H_2	26.59	141.5	Reject
H_3	26.59	118.0	Reject



(a) True Positive Rate (TPR)

(b) False Positive Rate (FPR)

Figure 4. True and False positive rates for packages and terms over three algorithms in all categories of Sourceforge.

The results for the tests are in Table 4. We reject all null hypotheses, meaning that the mean TPR and FPR given by SVM are statistically significantly higher than the results from DT or NB. 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 a dataset in our evaluation.**

B. 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 two types of attribute that have never been tested before for software categorization: API classes and packages. This section compares the quality of categorization when using each of these types of attribute. We did this comparison using multiple machine learning algorithms to minimize a threat to validity faced when using only one algorithm.

We used API classes and packages as the input to each of the machine learning algorithms, and computed the TPR and FPR in each category of both repositories. A statistical summary of the results is presented in Figure 3. Each boxplot represents one type of attribute. The average TPR for packages was 28.84% and the average FPR for packages was 4.75%. These values mean that about 29% of the predictions placed applications correctly into a category.

About 5% of predictions placed an application in wrong categories. The average TPR for classes was 23.84%, and average FPR for classes was 4.34%. The TPR for packages presents a roughly 20% improvement over classes. Also, the Friedman test shows that the difference in averages is statistically-significant: For TPR, $Q_{critical}$ is 3.841, $Q_{observed}$ equals 4.282, and p is 0.039. For FPR, $Q_{critical}$ is 3.841, $Q_{observed}$ equals 9.000, and p is 0.003. These values lead us to reject the null hypothesis stating that there is no significant difference between TPR and FPR for packages and classes. Therefore, we conclude that **API packages are more effective attributes than API classes for categorization of the applications in both repositories we used as a dataset.**

This conclusion means that the API packages that an application uses are specific to certain categories. Because the categories are groups based on functionality, the API packages relate to that functionality. Our finding reinforces the intuition that programmers will use APIs that relate to their tasks in that, in general, programmers cannot generate new API packages arbitrarily, as they can terms such as identifier names.

C. 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 [17, 25]. 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 contrast packages to single words from source code because we found packages to be the best-performing attributes (see Section V.B).

Figure 4 shows the TPR and FPR for all three machine learning algorithms using the packages 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 which do not own the source code that they develop. The Friedman test for TPR produces a $Q_{critical}$ of 3.841 and $Q_{observed}$ of 16.00. The value of p is less than 0.0001. Therefore, **we reject the null hypothesis that there is no statistically significant difference between the TPR when using packages or terms**.

The test results for FPR show that $Q_{critical}$ is 3.841, $Q_{observed}$ is 4.267, and p is 0.039. We reject the null hypothesis that there is no statistically significant difference between the FPR when using packages or terms. The average values in these cases are very similar, however: Using packages, the average TPR is 30.07% and FPR is 3.34%. Using terms, the TPR is 33.97% and FPR is 3.24%. Moreover, API packages are independent of applications that use them, and the semantics of the API calls in these packages is modular and precisely-defined. Therefore, even though the difference is statistically significant, the similarity of these values suggests that **packages are a good alternative to terms in the case when the terms are not available**.

VI. DISCUSSION AND FUTURE WORK

In Section V, we found that using packages as attributes outperformed classes, and that terms outperformed packages. One explanation for this result is that the packages are more specific to the categories than classes, and that terms are more specific than packages. We illustrate this explanation for this result in Table 5. The term “replyto” was the top feature according to EEL for the category `Email` in Sourceforge. Therefore, applications that contain “replyto” were more-likely to be categorized into the `Email` category. Similarly, the package `sun.net.www` was the top package for that category. We observe that 33 applications from Sourceforge contained the word “replyto”, while 300 used the API package `sun.net.www`. Eight applications in the category `Email` used each of those attributes. In this case, the term is a higher quality feature than the package because the term is more specific to the category than the package.

One key advantage to using API packages and classes 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 [19, 20]. Therefore, APIs are likely to be high-quality attributes for categorization, even if the terms are not.

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 [6]. 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 [27].

Extending the work of Anvik and Murphy [2], 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.

Table 5. Top term, class, and package of Email category of Sourceforge.

Type of Attribute	Attribute	Apps in Category with Attribute	Total Apps with Attribute
Term	replyto	8	33
Package	sun.net.www	8	300
Class	com.sun.jlex.internal.CEmit	8	300

VII. 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 and FPR 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 IV.A.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 and FPR 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 [25], and found comparable results. The fact that both repositories are in Java also introduces a threat to validity. We use API packages and classes, 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. 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.

VIII. RELATED WORK

Machine Learning has previously been used to categorize software systems. The work by Ugurel et al. is the most similar to ours in that we use supervised machine learning [25]. 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. 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.

MUDABlue is an information retrieval technique for software categorization [17]. 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 in our case study.

LACT is another system that relies on information retrieval to categorize software [24]. 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.

Bruno et al. [4] propose an approach for locating web services. Their approach takes a natural language query and uses SVM to match they 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.

Categorization has previously been used with other software artifacts in order to achieve some tasks related to software maintenance and evolution. Menzies et al. [18] 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 [1] 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. [13] 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.

Our work is related to Exemplar, a search engine that locates relevant applications [10] in that Exemplar matches query keywords to words in the documentation of API calls. However, Exemplar does not categorize software. Similarly, SSI is a technique for computing the similarity between source code based on API calls, but is used to locate source code using queries, not to categorize software [3].

IX. CONCLUSION

We present an approach for categorizing software applications in the context of maintenance tasks. We extract 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 three different machine learning algorithms and two software repositories, and compared our approach to the closest competing technique. We found that using API packages provided as good accuracy as using terms, even though the number of API packages is much smaller than the number of terms. Also, we found that our approach is applicable to repositories where the terms 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. Using our technique, maintainers of software can categorize applications even when the source code is not available, and use these categories to predict problems, or extract related bugs or features.

REFERENCES

[1] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y. Guéhéneuc, "Is it a bug or an enhancement?: a text-based approach to classify change requests," in *CASCON'08*.

[2] J. Anvik and G. Murphy, "Determining Implementation Expertise from Bug Reports," in *MSR'07*.

[3] S. Bajracharya, J. Ossher, and C. V. Lopes, "Leveraging Usage Similarity for Effective Retrieval of Examples in Code Repositories," in *FSE'10*.

[4] M. Bruno, G. Canfora, M. Di Penta, and R. Scognamiglio, "An Approach to support Web Service Classification and Annotation," in *EEE'05*.

[5] J. Demsar, "Statistical Comparisons of Classifiers over Multiple Data Sets," *Machine Learning Research,* vol. 7, pp. 1-30, 2006.

[6] G. A. Di Lucca, M. Di Penta, and S. Gradara, "An Approach to Classify Software Maintenance Requests," in *ICSM'02*.

[7] H. Dumitru, M. Gibiec, N. Hariri, J. Cleland-Huang, Mobasher B., Castro-Herrera C., and M. M., "On-demand Feature Recommendations derived from Mining Public Product Descriptions," in *ICSE'11*.

[8] C. X. J. Feng, Z. G. S. Yu, J. T. Emanuel, P. G. Li, X. Y. Shao, and Z. H. Wang, "Threefold versus fivefold cross-validation and individual versus average data in predictive regression modelling of machining experimental data," *Int. J. Comput. Integr. Manuf.,* vol. 21, pp. 702-714, 2008.

[9] M. Grechanik, C. Csallner, C. Fu, and Q. Xie, "Is Data Privacy Always Good For Software Testing?," in *ISSRE'10*.

[10] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby, "A Search Engine For Finding Highly Relevant Applications," in *ICSE'10*.

[11] M. Grechanik, C. McMillan, L. DeFerrari, M. Comi, S. Crespi, D. Poshyvanyk, C. Fu, Q. Xie, and C. Ghezzi, "An Empirical Investigation into a Large-Scale Java Open Source Code Repository," in *ESEM '10*.

[12] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection," *Machine Learning Research,* vol. 3, pp. 1157-1182, 2003.

[13] A. Hindle, D. M. Germán, M. W. Godfrey, and R. C. Holt, "Automatic classication of large changes into maintenance categories," in *ICPC'09*.

[14] C. Hsu and C. Lin, "A comparison of methods for multiclass suport vector machines," *IEEE Transactions on Neural Networks,* vol. 13, pp. 425-425, 2002.

[15] C. Jones, *Software Engineering Best Practices*. New York, NY: McGraw-Hill, 2010.

[16] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue, "Automatic Categorization Algorithm for Evolvable Software Archive," in *IWPSE'03*.

[17] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue, "MUDABlue: An automatic categorization system for Open Source repositories," *Journal of Systems and Software,* vol. 79, pp. 939-953, 2006.

[18] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *ICSM'08*.

[19] D. Ratiu and F. Deissenboeck, "From Reality to Programs and (Not Quite) Back Again," in *ICPC'07*.

[20] D. Ratiu and F. Deissenboeck, "How Programs Represent Reality (and How They Don't)," in *WCRE'06*.

[21] P. S. Sandhu, J. Singh , and H. Singh, "Approaches for Categorization of Reusable Software Components," *Journal of Computer Science,* vol. 3, pp. 266-273, 2007.

[22] F. Sebastiani, "Machine Learning in Automated Text Categorization," *ACM Computing Surveys,* vol. 34, pp. 1-47, 2002.

[23] M. Sokolova and G. Lapalme, "A systematic analysis of performance measures for classification tasks," *Information Processing and Management,* vol. 45, pp. 427-437, 2009.

[24] K. Tian, M. Revelle, and D. Poshyvanyk, "Using Latent Dirichlet Allocation for Automatic Categorization of Software," in *MSR'09*.

[25] S. Ugurel, R. Krovetz, C. Lee Giles 'z, D. M. Pennock, E. J. Glover, and H. Zha, "What's the code? automatic classification of source code archives," in *SIGKDD'02*.

[26] B. Újházi, R. Ferenc, D. Poshyvanyk, and T. Gyimóthy, "New Conceptual Coupling and Cohesion Metrics for Object-Oriented Systems," in *SCAM'10*.

[27] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller "How Long Will It Take to Fix This Bug?," in *MSR'07*.

[28] Y. Yang and J. O. Pedersern, "A Comparative Study on Feature Selection in Text Categorization," in *ICML'97*.

[29] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, " Cross-project defect prediction: a large scale experiment on data vs. domain vs. process," in *FSE'09*.