Automatic Classification of Software Artifacts in Open-Source Applications

Yuzhan Ma, Sarah Fakhoury
Michael Christensen, Venera Arnaoudova
School of Electrical Engineering and Computer Science
Washington State University
{yma1,sfakhour,mchriste,varnaoud}@eecs.wsu.edu

Waleed Zogaan
Mehdi Mirakhorli
Department of Software Engineering
Rochester Institute of Technology
{waz7355,mxmvse}@rit.edu

ABSTRACT

With the increasing popularity of open-source software development, there is a tremendous growth of software artifacts that provide insight into how people build software. Researchers are always looking for large-scale and representative software artifacts to produce systematic and unbiased validation of novel and existing techniques. For example, in the domain of software requirements traceability, researchers often use software applications with multiple types of artifacts, such as requirements, system elements, verifications, or tasks to develop and evaluate their traceability analysis techniques. However, the manual identification of rich software artifacts is very labor-intensive. In this work, we first conduct a large-scale study to identify which types of software artifacts are produced by a wide variety of open-source projects at different levels of granularity. Then we propose an automated approach based on Machine Learning techniques to identify various types of software artifacts. Through a set of experiments, we report and compare the performance of these algorithms when applied to software artifacts.

CCS CONCEPTS

• **Software and its engineering → Software organization and properties**; **Software notations and tools**; **Software libraries and repositories**;

KEYWORDS

Open-source software, machine learning, software artifacts.

ACM Reference Format:
Yuzhan Ma, Sarah Fakhoury, Michael Christensen, Venera Arnaoudova, Waleed Zogaan, and Mehdi Mirakhorli. 2018. Automatic Classification of Software Artifacts in Open-Source Applications. In *MSR '18: 15th International Conference on Mining Software Repositories, May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3196398.3196446

MSR '18, May 28–29, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5716-6/18/05…$15.00
https://doi.org/10.1145/3196398.3196446

1 INTRODUCTION

Empirical and data-centric research is largely enabled by the existence of datasets used to develop new research techniques or evaluate and compare existing ones. An example of data-centric research is the automated software requirements traceability. In this area, *training datasets* are needed to train trace-algorithms based on Machine Learning (ML) techniques. Researchers use labeled datasets of functional and non-functional requirements to train classification techniques to create traceability links between quality attributes and requirements document, design models and source code [7, 35, 38, 39, 46]. *Validation datasets* are needed to tune parameters of such trace-algorithms [6, 30, 35]. *Testing datasets* are used to test the performance of trace-algorithms on unseen data. Researchers use datasets to evaluate the accuracy of trace-algorithms based on Information Retrieval (IR) by establishing links between requirements and source code [11, 12, 21, 46, 53]. In other domains, researchers use design documents to create a ground truth software architecture model for an evolving software system [16].

Obtaining such software development datasets is one of the most frequently reported barriers for researchers in the software engineering domain [28, 44]. In recent years, with the advancement and popularity of the open-source approach to software development, researchers benefit from publicly available source code repositories [36]. Software artifacts, other than source code and issue tracking entities, can also provide a great deal of insight into a software system and facilitate knowledge sharing and information reuse. However, it can be a labor-intensive task to manually identify the types of artifacts available or lacking in a specific open-source project. Previous studies show that obtaining such artifacts from open-source projects is non-trivial and researchers lack appropriate automated support to identify, filter, and browse through such artifacts [57]. More importantly, we currently lack an in-depth understanding of the various types of software artifacts that are available in open-source projects. The common assumption is that open-source projects often lack software artifacts such as requirements and design documents.

In this paper, we aim at improving the understanding of open-source projects by investigating this common assumption, thus, answering two motivating questions:

> **Motivating Question #1:** What types of artifacts are created during open-source software development?

To this end, we conduct a large-scale empirical study involving 383 open-source software projects that are randomly sampled from GitHub. These projects are studied to obtain an empirically-based

understanding of the artifacts developed in open-source projects. Then we classify all artifacts contained in this sample of open-source projects using the proposed automatic approach. Results show that indeed open-source software projects often lack documentation related artifacts, which account for only 6.12% of the total number of software artifacts. Although the quantity of documentation related artifacts is low, 14.88% of the projects contain either design or requirement documents, which means that open-source projects could be a valuable resource for researchers interested in obtaining such artifacts.

Motivating Question #2: Can we automatically detect and categorize open-source software artifacts?

Using heuristics, we categorize artifacts into two groups: those that can be classified based on file name and extension alone (e.g., bat files) and those that require deeper analysis in order to be classified (e.g., text documents). We manually classify a sample of the artifacts from the second group and construct an oracle. During the manual classification, we identify features that are relevant for artifact classification. After this, we explore various ML algorithms for software artifact classification. Next, we report the performance of our approach on the validation and testing datasets and finally, we classify all artifacts present in the 383 open-source projects and report the prevalence of the different types of artifacts.

Our results show that we can successfully apply ML algorithms to text documents to classify software artifacts. Using ensemble techniques, such as voting, we are able to combine the predictive power of several algorithms that perform well on unique categories of software artifacts to create one classifier with improved performance across all categories. Our model achieves 85% precision and 82% recall when evaluated on the manually created oracle using 10-fold cross-validation. When applied to a testing dataset of unseen data gathered after the parameter tuning on the validation dataset, our approach achieves 76% precision and 75% recall.

The contributions of this work are as follows:

1. We provide insights into the types of artifacts created during open-source software development. Although documentation related artifacts only account for 6.12% of total software artifacts in open-source software projects, 14.88% of the projects contain either design or requirement documents, which is valuable resources for empirical studies that require such documents.
2. We propose a novel approach that utilizes heuristics and various ML classifiers that automatically classify software artifacts.
3. We supply a replication package [31], which includes (i) information about the sampled projects, (ii) an oracle of 208 manually classified documentation related software artifacts used for training, validation, and testing of the proposed approach, and (iii) the list of features used for the ML algorithms.

Paper Structure. Section 2 provides details regarding the study design and the automatic artifact classification approach. Section 3 presents the results, while Section 4 discusses related literature. Threats to validity are discussed in Section 5 and Section 6 concludes the paper and outlines directions for future work.

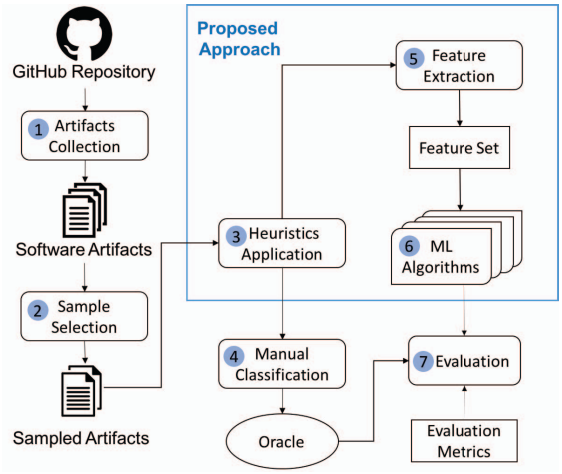

Figure 1: Approach overview.

2 STUDY DEFINITION AND DESIGN

The *goal* of this study is to investigate what types of artifacts are created during open-source software development. To achieve this goal, we propose an automatic approach for software artifact detection and classification using machine learning approaches. The *quality focus* is the performance of the proposed approach on artifact classification in terms of selected evaluation metrics such as precision and recall. The *perspective* of the study is that of researchers, who are interested in automatically obtaining software development artifacts that fit their research need. The evaluation is carried out in the *context* of open-source projects collected from GitHub [24]. More specifically, the study aims at addressing the following *research questions*:

- **RQ_1:** *How can software artifacts be categorized?* To answer this question we randomly sample from a large set of open-source projects and manually examine the type of artifacts available.
- **RQ_2:** *How accurate is the proposed approach for automatic software artifact classification?* We investigate the performance of the proposed approach using different evaluation metrics. We report results on validation and testing datasets using 10-fold cross-validation.
- **RQ_3:** *What types of artifacts are created during open-source software development?* We classify all artifacts present in the studied open-source projects and report the prevalence of the different types of artifacts.

Figure 1 depicts the overview of our approach, which is designed to automatically classify software artifacts leveraging (i) heuristics based on file names and extensions and (ii) existing ML algorithms. To answer $\mathbf{RQ_1}$, we collect a large set of diverse open-source projects and obtain a significant random sample of the projects. We identify the artifacts contained in the sampled projects and divide them into two groups by applying heuristics on file names and extensions. The first group contains artifacts that can be classified solely based on file names and extensions whereas the second group

contains artifacts that require deeper analysis in order to be classified. We manually classify a sample of the artifacts contained in the second group to construct an oracle of classified artifacts. During the manual classification, we also identify features that could be used to automate the artifact classification. For $\mathbf{RQ_2}$, we automate the feature extraction process and use various ML algorithms to automatically classify software artifacts belonging to the second group. Finally, to answer $\mathbf{RQ_3}$ we classify all artifacts of the studied open-source projects and report the frequency of occurrence of each type of artifact identified during the manual process.

The rest of this section is organized as follows: Section 2.1 provides details about the software systems used in this study. Section 2.2 describes the process that we followed to create the oracle. Section 2.3 describes the proposed automatic classification approach and section 2.4 lists the evaluation metrics used to evaluate the performance of the proposed approach for automatic artifact classification.

2.1 Subject Systems

We extract a large set of 91,108 open-source projects from GitHub making use of a code crawling application known as *GHTorrent* [19]. *GHTorrent* acts as a service to extract data and events, returning MongoDB data dumps. The dumps are composed of information about projects in the form of users, comments on commits, languages, pull requests, follower-following relations, and others.

To collect a significant sample of projects for our study, we randomly sample 383 projects from the collected open-source projects, ensuring 95% confidence level and 5% margin of error. All research questions are addressed using the sampled projects.

2.2 Oracle

To create an oracle of classified software artifacts, we manually examine a random set of artifacts from the 383 sampled projects. When the file name/extension are insufficient to classify an artifact, we analyze the file content. Two coders perform the classification of artifacts independently. An inter-rater reliability (IRR) analysis [23] is used to assess the degree to which coders consistently classify software artifacts. Both coders are Master students in Computer Science. Disagreements between the coders are resolved with discussions and when necessary a third coder is brought in. The category of artifacts are coded using categorical variables. The Cohen's kappa statistic measures the observed level of the agreement between coders for a set of nominal ratings and corrects for agreement that would be expected by chance, providing a standardized index of IRR that can be generalized across studies [23]. Possible values for kappa range from -1 to 1, with 1 indicating a perfect agreement, 0 indicating a completely random agreement, and -1 indicating a total disagreement. Landis and Koch [27] provide guidelines for interpreting kappa values as follows: values from 0.0 to 0.2 indicate slight agreement, values from 0.21 to 0.40 indicate fair agreement, 0.41 to 0.60 indicate moderate agreement, 0.61 to 0.80 indicate substantial agreement, and 0.81 to 1.0 indicate almost perfect or perfect agreement. The data in this study is collected through ratings provided by coders and has a significant impact on the computation and interpretation of our study. It is important that coders can independently reach similar conclusions about the types of software artifacts they identify because that confirms the established categories are well defined. Thus, we target at least substantial agreement, i.e., above 0.61.

2.3 Automatic Artifact Classification

To automate the software artifact classification process we identify heuristics based on file names and extensions (Section 2.3.1). For files that require further analysis we extract features (Section 2.3.2) that we use as input to machine learning algorithms (Section 2.3.3).

2.3.1 Heuristics Application. We utilize existing file name/extension categorization [10] and we randomly sample a portion of the most frequently occurring extensions to confirm the correctness of such categorization. In addition to file extension, we expect the file name to provide useful information in artifacts identification as well. For example, testing code is often organized under directory with names contain "test" or "tests" and files with .wav extension can be automatically identified as audio file. Such identification is assumed to be correct by construction. On the other hand, some files, such as .txt, can not be identified without examining the file content.

2.3.2 Feature Creation Process. Generating a set of features for text classification problems could be achieved with the use of various information retrieval techniques. For instance, one could use a Vector Space Model [43] and use a weighting schema such as Term Frequency-Inverse Document Frequency (TF-IDF) [45] to automatically extract the most important terms in a document. Other, more sophisticated techniques that could be used are Latent Semantic Indexing (LSI) [14] and Latent Dirichlet Allocation (LDA) [3]. Information retrieval techniques are most useful when the characteristics of the documents that we are working on are unknown. In other words, we rely on the technique to identify hidden patterns that characterize each document.

Instead, we decided to use the knowledge gained through the manual validation process of artifacts and thus manually creating the set of features that characterize each type of artifact. Because an optimal set of features cannot be determined a priori, the two annotators generate an initial set of features and iteratively refine the set through discussions. This manual approach gives us more flexibility in determining the relevant set of features, while harnessing the knowledge gained during the oracle creation process.

2.3.3 Machine Learning Algorithms. We select seven different machine learning approaches belonging to three different categories: decision trees, Support Vector Machines, and Bayesian Networks. Research has shown that these algorithms perform well for text classification problems [25, 32, 34, 49]. We use the implementations provided through Weka [22] and evaluate the classifiers using 10-fold cross-validation. In other words, we evaluate the predictive models by partitioning the original sample into 10 equal sized sub samples, performing the analysis on one subset, and validating the analysis on the other. The validation is repeated 10 times to obtain an average estimate of the predictive model. We briefly describe the selected algorithms and the parameter tuning that we performed:

(1) **Random Forest** [4] averages the predictions of a number of tree predictors where each tree is fully grown and is based on independently sampled values. The large number of trees avoids over fitting. Random Forest is known to be robust

to noise and to correlated variables. We use the function randomForest (package randomForest) with the number of trees being 500 as a starting point, which has shown good results in previous works [52]. We tune the parameters for the number of trees varying from 500 to 1000 and for the features explored at each branch from the default value: $(\log_2(\#predictors)+1)$ to 20% of the total number of features with a step of 0.05.

(2) **Sequential Minimal Optimization (SMO)** is an implementation of John Platt's sequential minimal optimization algorithm to train a support vector classifier. We use RBF kernel, Polynomial kernel, and the Pearson VII function-based universal kernel (PUK) [50] in combination with this classifier. We tune the exponent parameter of the classifier varying from 1.0 to 4.0 with a step of 0.5, the gamma parameter from 0 to 1 with a step of 0.05, and the cost parameter from 1 to 50 with a step of 1.

(3) **Multinomial Naïve Bayes** is a specific version of Naïve Bayes, created for improved performance on text classification problems [34]. Naïve Bayes is the simplest probabilistic classifier applying Bayes' theorem. It makes strong assumptions on the input: the features are considered conditionally independent among each other. We explore the performance of the classifier using kernel estimator and supervised discretization.

(4) **J48** is an implementation of the C4.5 decision tree. This algorithm produces human understandable rules for the classification of new instances. The implementation provided through Weka offers three different approaches to compute the decision trees, based on the type of the pruning techniques: pruned, unpruned, and reduced error pruning. We tune the parameter for the minimum number of instances at each leaf from 1 to 8 with a step of 1.

(5) **Ensemble Learning** is used to combine individual classifiers with the aim of obtaining better overall predictive performance. We use the majority vote algorithm provided through Weka. The majority vote approach considers the votes of each classifier for the label of an instance and uses the label agreed upon by the majority.

2.4 Evaluation

We evaluate the performance of the automatic artifact classification approach using the following evaluation metrics:

2.4.1 Precision. Precision is defined as the percentage of artifact predicted as belonging to the categories that are correct with respect to the oracle, $Precision = TP/(TP + FP)$, where TP and FP are the number of true and false positives, respectively.

2.4.2 True Positive Rate (TPR). TPR or relative recall is calculated as the ratio between the number of true positives and the total number of positive events, i.e., $TPR = TP/(TP + FN)$. In the context of this study, the TPR indicates how many of the manually known software artifacts are correctly discovered.

2.4.3 F-Score. Precision and recall are inversely related, thus, it is difficult to compare results of the model using the two metrics. F-score is used to aggregate both measures into a single value. F-score is the harmonic mean of the precision and recall, i.e., $F = 2*Precision*TPR/(Precision+TPR)$. F-score reaches its best value at 1 (perfect precision and recall) and worst at 0.

2.4.4 Area Under the Receiver Operating Characteristic (ROC) curve. ROC is a plot of the true positive rate against the false positive rate at various discrimination thresholds. The area under ROC is close to 1 when the classifier performs better and close to 0.5 when the classification model is poor and behaves like a random classifier.

2.4.5 Matthews Correlation Coefficient (MCC). MCC is a measure used in machine learning to assess the quality of a two-class classifier especially when the classes are unbalanced [33].

$$MCC = \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP+FP)(FN+TN)(FP+TN)(TP+FN)}}$$

Values range from -1 to 1, where 0 indicates that the approach performs like a random classifier. Other correlation values are interpreted as follows: $MCC < 0.2$: low, $0.2 \leq MCC < 0.4$: fair, $0.4 \leq MCC < 0.6$: moderate, $0.6 \leq MCC < 0.8$: strong, and $MCC \geq 0.8$: very strong [8].

2.4.6 Micro and Macro Average. There are different ways to average results of a multi-class classifier. Macro-average treats each class with equal weight and is calculated as the average of the metrics computed within each class. Micro-average gives each individual instance equal weight so that the largest classes have most influence. It is computed by aggregating the outcomes across all classes and computing a metric with aggregated outcomes. We report all evaluation metrics along with both micro and macro average.

3 RESULTS AND ANALYSIS

In this section, we report the results of our study, with the aim of answering the research questions formulated in Section 2.

3.1 RQ_1: How can software artifacts be categorized?

We extracted 91,108 open-source projects in various programming languages from GitHub between April and October 2015. To achieve 95% confidence level and 5% margin of error, we randomly select 383 applications and study software artifacts in those projects. The size of the selected subjects, in terms of Lines Of Code (LOC), ranges from 2 to 12 million LOC. Table 1 provides descriptive statistics of the sampled projects[1]. In addition, Figure 2 shows the distribution of the primary programming language across the projects, i.e., the language with the highest number of LOC.

We identify the following artifact types only by file names and extensions as shown in Table 2: application, archive, audio, disk image, font, image, project, source code, testing code, and miscellaneous. Some file extensions can be associated with multiple file types. For example, png can be Portable Network Graphics Image or Corel Paint Shop Pro Browser Catalogue, i.e., an image file or a documentation file. We randomly sample 5 instances of such extensions and assign them to one file type based on their file content. In addition to extensions we separate testing code from source code,

[1]LOC is computed using CLOC [9] which counts blank lines, comment lines, and physical lines of source code separately. We report the physical lines of source code.

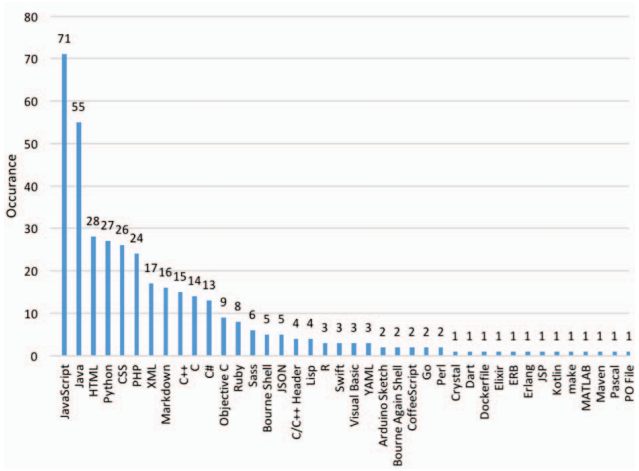

Figure 2: Distribution of primary languages in the sampled projects.

Table 1: Statistics for the size of the sampled projects.

	LOC
Min	2
Q1	374
Median	1,264
Mean	102,986
Q3	9,127.5
Max	12,609,300

by verifying if one of the following keywords appears in file name or directory: "test", "tests", and "mock". Another heuristic is used to identify miscellaneous files based on the number of words in the file. Through experiments, we observe that a threshold of 30 offers a good compromise between precision and recall.

We analyze the file extensions associated with open-source projects. There are 234,296 artifacts with 1,217 distinct files extensions in the sampled projects, excluding hidden files. However, the top 38 most frequent file extensions occur in more than 95% of the projects and account for over 76% of the total artifacts. Table 3 shows the top 38 most frequent file extensions along with the number of projects that contain files with these extensions and the number of files with these extensions in the sampled projects. "Num. of Projects" (%) reports on the number (percentage) of sampled applications that contain files of each extension. "Num. of Files" (%) reports on the number (percentage) of files with each extension in the sampled applications. "Cum. %" reports the cumulative % of the artifacts. For instance, the first row shows that 1) 383 out of the 383 sampled projects, i.e., 100%, contain files without extension and 2) 13,706 out of 234,296 artifacts, i.e., 5.85%, have no extension. The extensions highlighted in gray are documentation related files that are not identified by the heuristics shown in Table 2. Since it is not feasible to manually go through every single file, we sampled 2% of files with the highlighted extensions.

Table 2: Heuristics applied to identify types of non-documentation related artifacts.

Artifact Type	Heuristic
Application	.bat .cmd .exe .ser .swf
Archive	.a .gz .jar .pack .zip
Audio	.kt .mp3 .ogg .wav
Disk Image	.scl
Font	.eot .otf .ttf .woff
Image	.blp .bmp .dds .gif .ico .jpeg .jpg .png .psd .rs .svg .tga .tif .xpm
Project	.csproj .pbxproj .vcproj .vcxproj
Source Code	.as .asm .c .cc .class .coffee .cpp .cs .cshtml .css .ctp .cxx .d .dll .ebuild .ejs .el .erb .erl .f .f90 .go .gradle .groovy .h .haml .hpp .hs .i .java .js .jsp .less .lua .m .mo .o .php .phpt .phtml .pl .pm .pp .py .pyc .r .rb .s .scala .scss .scssc .sh .smali .so .sql .swift .t .tcl .ts .vb .vim .rkt
Testing Code	if a file is classified as code, we further examine if "test", "tests", and/or "mock" is contained in fully qualified file name, ex. ProjectName/src/test/file.java
Miscellaneous	non-readable files non-English files insufficient information (files with ≤ 30 words)

To create an oracle of documentation related files, two coders manually and independently classify 894 randomly selected artifacts. 149 out of 894 sampled artifacts are documentation related files. During this manual classification process, we iteratively refine and consolidate the initial list of categories as needed. The initial IRR value is 0.64 and it is calculated for a set of 115 artifacts. The two coders then discuss the discrepancies to reach an agreement. The subsequent IRR value increased to 0.786 for the next 115 artifacts, which indicates substantial agreement [27]. Since kappa shows substantial agreement, the remaining software artifacts categorization was conducted by only 1 coder. Our manual analysis led to the creation of a taxonomy of documentation related artifacts with 7 distinct categories. A description of each category follows:

(1) **Contributors' Guide** contain information targeting the contributors to the project such as how to begin contributing to the project, the review process, tips on debugging, etc.
(2) **Design Documents** contain information about the design of the project, such as design patterns and design decisions, underlying project framework and architecture, as well as version compatibility details.
(3) **License** contain information about copyright and the type of licenses the project operates under.
(4) **List of Contributors** contain information about and credit to the authors and maintainers of the project, including author names, their roles, and contact information.
(5) **Release Notes** are usually documents shared with end users or clients and outline specific version changes, bug fixes, or enhancements made to the project.
(6) **Requirement Documents** often contain functional and non-functional requirements, use cases, and other software specifications that target expected user interactions.
(7) **Setup Files** contain all artifacts that have to do with project setup. Examples include manifest files, make files, configuration files, and version requirement files.

Table 3: Extension distribution in the sampled projects.

File Extension	Num. of Projects	%	Num. of Files	%	Cum. %
no extension	383	100.00%	13,706	5.85%	5.85%
md	262	68.41%	1,853	0.79%	6.64%
html	162	42.30%	3,203	1.37%	8.01%
txt	162	42.30%	7,828	3.34%	11.35%
png	153	39.95%	8,450	3.61%	14.96%
js	152	39.69%	9,921	4.23%	19.19%
css	132	34.46%	1,405	0.60%	19.79%
xml	115	30.03%	6,147	2.62%	22.41%
json	109	28.46%	1,542	0.66%	23.07%
jpg	97	25.33%	1,300	0.55%	23.63%
java	79	20.63%	3,582	1.53%	25.15%
ico	65	16.97%	96	0.04%	25.20%
svg	59	15.40%	435	0.19%	25.38%
sh	58	15.14%	1,265	0.54%	25.92%
gif	56	14.62%	2,614	1.12%	27.04%
properties	53	13.84%	164	0.07%	27.11%
py	53	13.84%	15,147	6.46%	33.57%
h	49	12.79%	48,448	20.68%	54.25%
php	42	10.97%	2,645	1.13%	55.38%
jar	42	10.97%	260	0.11%	55.49%
ttf	42	10.97%	131	0.06%	55.55%
yml	39	10.18%	264	0.11%	55.66%
woff	37	9.66%	79	0.03%	55.69%
eot	36	9.40%	78	0.03%	55.73%
pdf	35	9.14%	400	0.17%	55.90%
rb	32	8.36%	2,267	0.97%	56.86%
scss	29	7.57%	780	0.33%	57.20%
c	28	7.31%	43,056	18.38%	75.57%
sln	28	7.31%	100	0.04%	75.62%
lock	27	7.05%	41	0.02%	75.63%
conf	26	6.79%	270	0.12%	75.75%
bat	26	6.79%	51	0.02%	75.77%
plist	25	6.53%	266	0.11%	75.88%
cpp	25	6.53%	1,581	0.67%	76.56%
cache	23	6.01%	59	0.03%	76.58%
log	22	5.74%	133	0.06%	76.64%
config	21	5.48%	77	0.03%	76.67%
map	20	5.22%	123	0.05%	76.73%

Table 4: Sample list of features.

Document Type	#	Example Features
Contributors' Guide	26	contribute, welcome, checkout, severity
Design Document	10	architecture, design, framework, layer
License	30	disclaimer, free, law, reproduction
List of Contributors	18	authors, instructions, maintainers, thank
Release Notes	30	added, bug, date, fixed, improve, version
Requirement Document	10	feature, functionality, support, requirement
Setup Files	25	build, configure, defaults, ignore, manifest

During the manual classification, we identify 342 unique features that characterize the categories in the above taxonomy. Some of those are based on their frequency of occurrence in artifacts, while others are identified by the coders. We observe that five features are not present in any of the files in our oracle. We remove those features and retain the remaining 337 features that we will use for the automatic artifact classification. Table 4 shows examples of the features we used to identify each category and the distribution of artifacts in our oracle. The complete list of features can be found in our online replication package. Based on the in-depth analysis and manual classification of 894 artifacts, the following conclusion was drawn:

> **RQ$_1$** Summary: Some software artifacts can be categorized solely using heuristics based on file names and extensions. However, other artifacts that are documentation related require deeper analysis and identification of characterizing features to be classified.

3.2 RQ$_2$: How accurate is the proposed approach for automatic software artifact classification?

In this section we evaluate the performance of the automatic artifact classification. We do not evaluate the classification of non-document related artifacts, i.e., those listed in Table 2 as those are correct by construction. Table 5 contains the results of applying ML algorithms using 10-fold cross-validation. Results per class as well as the micro and macro averages across classes are reported. Overall, Naïve Bayes Multinomial has the best performance with a micro average precision of 0.80, 0.76 recall, 0.76 F-measure, 0.73 MCC, and 0.95 ROC. The high values for MCC and ROC indicate that the classifier performs very well on the validation dataset.

Values in bold indicate the best performance achieved per class for both precision and recall. For example, at 0.74, J48 is able to achieve the highest precision for the class Release Notes relative to the other classifiers. However, at 0.83, Naïve Bayes Multinomial achieves the highest recall for Release Notes. Each algorithm achieves the best precision and recall performance for at least one class, therefore, different algorithms may be better suited to classify instances from different classes. Using ensemble techniques, such as voting we are able to combine the predictive power of several algorithms that perform well on unique classes, to create one classifier with improved performance across all classes.

Table 6 contains the results of classifiers used in Table 5 combined using ensemble learning. Specifically, the classifiers are combined using majority vote. Results in Table 5 indicate that Naïve Bayes Multinomial performs the best on several different classes, therefore we increase the weight of its vote during classification by two to create a weighted majority vote, which has shown to be effective in similar text classification research [40]. As compared to the best performing single classifier, majority vote yields a micro average precision of 0.85, which is a 5% increase, recall increases by 6% to 0.82, F-Measure increases by 7% to 0.83. MCC increases by 7% to 0.80 and ROC decreases to 0.90, which still indicates strong performance.

Requirement Document is the class with the lowest performance using both single classifiers and voting. However, using voting we are able to achieve a better balance between precision and recall. The best precision and recall for the class are both at 0.40 for single classifiers, however, with ensemble learning precision drops by only 0.01 and recall increases by 0.26. Overall, voting improves the performance in terms of precision and recall across all classes. The only exception is with the class Setup Files for which SMO Polynomial Kernel is able to achieve a 3% higher precision and

Table 5: Performance of individual classifiers and 10-fold cross-validation on the training dataset.

Classifier	Parameters	Class	Precision	Recall	F-Measure	MCC	ROC
Naïve Bayes Multinomial	Default	Requirement Document	0.35	**0.70**	0.47	0.45	0.92
		Design Document	0.63	0.50	0.56	0.53	0.93
		Release Notes	0.69	**0.83**	0.76	0.69	0.97
		Setup Files	**0.86**	0.48	0.62	0.59	0.94
		License	**0.94**	**1.00**	0.97	0.96	1.00
		List of Contributors	**0.89**	**0.89**	0.89	0.87	0.99
		Contributors' Guide	**0.86**	**0.69**	0.77	0.73	0.89
		Micro Average	0.80	0.76	0.76	0.73	0.95
		Macro Average	0.74	0.73	0.72	0.69	0.95
SMO Poly Kernel	Default	Requirement Document	**0.40**	0.40	0.40	0.36	0.86
		Design Document	0.43	0.30	0.35	0.32	0.91
		Release Notes	0.70	0.77	0.73	0.66	0.90
		Setup Files	0.77	**0.92**	0.84	0.81	0.96
		License	**0.94**	0.97	0.95	0.94	0.99
		List of Contributors	0.82	0.78	0.80	0.77	0.95
		Contributors' Guide	0.81	0.65	0.72	0.68	0.87
		Micro Average	0.75	0.76	0.75	0.71	0.93
		Macro Average	0.69	0.68	0.69	0.65	0.92
Random Forest	#Trees 500	Requirement Document	**0.40**	0.20	0.27	0.25	0.91
		Design Document	**1.00**	0.30	0.46	0.53	0.97
		Release Notes	0.65	0.73	0.69	0.60	0.94
		Setup Files	0.71	0.88	0.79	0.74	0.95
		License	0.91	1.00	0.95	0.94	1.00
		List of Contributors	0.87	0.72	0.79	0.77	0.98
		Contributors' Guide	0.64	**0.69**	0.67	0.59	0.93
		Micro Average	0.74	0.74	0.72	0.69	0.96
		Macro Average	0.74	0.00	0.69	0.63	0.95
J48	MinNumObj 4	Requirement Document	0.27	0.30	0.29	0.23	0.70
		Design Document	0.67	**0.60**	0.63	0.61	0.84
		Release Notes	**0.74**	0.67	0.70	0.63	0.87
		Setup Files	0.45	0.52	0.48	0.37	0.76
		License	0.93	0.93	0.93	0.92	0.98
		List of Contributors	0.43	0.50	0.46	0.38	0.79
		Contributors' Guide	0.55	0.46	0.50	0.41	0.82
		Micro Average	0.62	0.61	0.62	0.55	0.84
		Macro Average	0.58	0.57	0.57	0.51	0.82

13% higher recall. Despite this, comparing the micro average for all classes of SMO Polynomial Kernel to the ensemble approach, the performance trade off is a 10% increase in precision, 6% increase in recall in favor of the ensemble approach.

In order to evaluate the model generated by the majority vote algorithm, we run the classifier on a newly generated oracle, the testing dataset, and analyze the results. Table 7 contains the results of the classifier on the second oracle of 59 data points. Overall, results for classes Contributors Guide, List of Contributors, Design Documents, License, and Setup Files are very similar, in term of F-Measure, MCC and ROC, to the performance obtained on the first oracle. Release Notes and Requirement Documents are two categories that perform significantly worse with 0.35 decrease in precision for Release Notes and 0.52 decrease in recall for Requirement Documents. The results for these two classes affect the overall micro and macro averages. 3 out of 7 instances from the Requirement Document class are categorized as Release Notes and 2 out of 10 instances of Release notes are categorized as Requirement Documents. We investigate the ML features across the different types of artifacts to understand the drop of performance in the testing dataset. Our analysis leads to two observations. First, we note that there is a significant decrease in the number of documents containing the features for Requirement Documents in the testing dataset. The second observation is that there is an increased overlap of features between Requirement Documents and Release Notes in the testing dataset. One explanation could be due to the fact that the features we manually created are not representative of Requirement Documents. Another explanation could be due to fact that Requirement Documents in the second oracle are considerably smaller in size compared to the Requirement Documents in the first oracle. Thus, there might not be enough textual content, i.e., features, in the second oracle for the ML algorithms to perform well. We plan to further investigate and try to improve the performance of ML features regarding the Release Notes and Requirement Document artifacts in our future work by adding more documents to the training set and by comparing the performance of manually

extracted features to that of automatically extracted features using information retrieval approaches.

RQ$_2$ Summary: Combining different ML algorithms through ensemble learning, we are able to automatically classify documentation related software artifacts with an average precision of 85% and recall of 82% using 10-fold cross-validation on the validation dataset—an oracle of 149 data points. Using the same classifier on a testing dataset of 59 new data points, our approach achieves an average precision of 76% and a recall of 75%.

3.3 RQ$_3$: What types of artifacts are created during open-source software development?

To explore the types of artifacts created during open-source software development, we run our classification approach on the entire sample set of 383 projects. Table 8 contains the predicted distributions of various documentation and non-documentation related artifacts created during open-source project development. "Num. of Projects" (%) reports on the number (percentage) of sampled applications that contain each type of artifact. Overall, the most common type of artifacts are source code, setup, miscellaneous, and archive, which are identified in over 50% of the applications. The least common type of artifacts are disk image and audio, which are identified in less than 5% of the applications.

"Num. of Files" (%) in Table 8 reports on the number (percentage) of artifacts from each category across all sampled applications. There is a total of 87,619 software artifacts in the sample applications. We observe that documentation related artifacts make up only 6.12% of all files. Further more, design documents and requirement documents only make up 0.42% and 0.68% respectively. Setup files account for 3.57% of the total artifacts. As expected, source code makes up 56.79% of the entire artifacts collection.

Focusing on documentation, we observe that 5.74% and 10.18% of the projects contain design and requirement documents, respectively. Taking into consideration that 4 projects contain both design and requirement documents, the combination of projects that contain either type makes up 14.88% of the sampled applications (22+39-4=57). Although documentation related artifacts only accounts for a small portion of the available artifacts, open-source projects can still be a good resource for researchers for such artifacts.

RQ$_3$ Summary: Using our automatic artifact classification approach, we confirm that open-source projects provides a variety of software artifacts. Approximately 14.88% of the projects contain either design or requirement documents.

4 RELATED WORK

This section discusses relevant literature. Section 4.1 discusses related work using open-source software as a dataset, Section 4.2 discusses related work pertaining to the categorization of software artifacts, and Section 4.3 discusses related work in text classification.

4.1 Open-Source Software as a Dataset

Godfrey and Tu [18] focus on the evolution of open-source software development and examine 96 releases of the Linux operating system kernel. This study aims to compare the evolutionary narratives of open-source with commercially developed systems. However, only files with ".c" and ".h" extensions are examined. Other source artifacts such as configuration files and documentation are ignored.

Behnamghader et al. [2] introduce a framework for conducting large-scale replicable empirical studies of architectural changes across different versions of 23 open-source software systems. The findings of this work bring new insights about the frequency of architectural changes in software systems.

Munaiah et al. [36] propose a framework that help researchers to identify GitHub repositories which contain engineered software projects. The proposed work defines dimensions that are used to classify software engineered projects through validating the existence of such dimensions in GitHub repositories.

Tian et al. [48] propose a technique using LDA to automatically categorize open-source applications. The proposed technique, called LACT, is evaluated in two studies and the results show that LACT is able to effectively and automatically categorize software systems regardless of their programming language.

Vendomo et al. [51] conduct an empirical study aiming at identifying and automatically detecting exceptions in open-source software licenses by relying on machine learning. They analyze the source code of 51K projects written in six programming languages and identify 14 different license exception types.

Zogaan et al. [56] present an empirical study and propose two automated techniques to generate traceability training datasets from technical programming websites and open-source software repositories. The proposed techniques use both Web-Mining and Big-Data Analysis to generate the training datasets and categorize them based on tactic-related code-artifacts. In their Big-Data approach, they use machine learning classifiers to detect tactic-related files that could be used as training datasets.

Caniell et al. [5] present a dataset that contains source code and related metadata of FOSS history for the Debian operating system. This dataset contains over 30 million code files in C and C++ along with their related metadata files.

In addition, there are a number of projects in the area of mining open-source software repositories [15, 55] with primarily focus on studying the source code and coding issues. There is a limited experimental research on using such resource to generate scientific datasets with diverse artifacts.

Our study complement and advances existing work. We propose an automated approach based on heuristics and machine learning techniques to identify various types of software artifacts that could assist researchers and practitioners in multiple sub-domains of software engineering to find appropriate datasets that fit their need.

4.2 Categorization of Software Artifacts

Robles et al. [41] analyze source code artifacts from versioning repositories beyond source code and provide insights into software projects from both a technical and management point of view. Robles et al. [42] propose a semi-automatic approach that determines the availability and quantity of documentation and source code comments in a libre software package. In both studies, only file extensions and names are utilized to identify the different types of files. Our approach is complementary to this study since we use file content in addition to file name and extension when classifying artifacts. We use manually extracted features and machine

Table 6: Performance of the classifiers using ensemble learning and 10-fold cross-validation on the training dataset.

Classifier	Class	Precision	Recall	F-Measure	MCC	ROC
Majority Vote	Release Notes	0.85	0.84	0.84	0.81	0.90
(2*Naïve Bayes Multinomial,	Contributors' Guide	0.90	0.78	0.84	0.81	0.88
SMO Poly Kernel,	List of Contributors	0.99	0.86	0.92	0.91	0.93
J48, and Random Forest)	Design Document	0.74	0.51	0.60	0.59	0.75
	License	0.98	1.00	0.99	0.99	1.00
	Requirement Document	0.39	0.66	0.49	0.46	0.79
	Setup Files	0.74	0.79	0.77	0.72	0.87
	Micro Average	0.85	0.82	0.83	0.80	0.90
	Macro Average	0.80	0.78	0.78	0.76	0.87

Table 7: Performance of the classifiers using ensemble learning and 10-fold cross-validation on the testing dataset.

Classifier	Class	Precision	Recall	F-Measure	MCC	ROC
Majority Vote	Release Notes	0.50	0.80	0.62	0.54	0.82
(2*Naïve Bayes Multinomial,	Contributors' Guide	0.90	0.82	0.86	0.83	0.90
SMO Poly Kernel,	List of Contributors	0.86	1.00	0.92	0.92	0.99
J48, and Random Forest)	Design Document	1.00	0.40	0.57	0.62	0.70
	License	1.00	0.90	0.95	0.94	0.95
	Requirement Document	0.33	0.14	0.20	0.15	0.55
	Setup Files	0.75	0.90	0.82	0.78	0.92
	Micro Average	0.76	0.75	0.73	0.70	0.85
	Macro Average	0.76	0.71	0.70	0.68	0.83

Table 8: Distribution of the different types of software artifacts in the sampled projects.

Software Artifacts	Category	Num. of Projects	%	Num. of Files	%
Documentation	Design Documents	22	5.74%	371	0.42%
	List of Contributors	33	8.62%	134	0.15%
	Requirement Documents	39	10.18%	592	0.68%
	Contributors' Guide	54	14.10%	389	0.44%
	License	84	21.93%	259	0.30%
	Release Notes	93	24.28%	489	0.56%
	Setup Files	235	61.36%	3,130	3.57%
Subtotal				5,364	6.12%
Non-Documentation	Disk Image	1	0.26%	4,209	4.80%
	Audio	5	1.31%	83	0.09%
	Project	25	6.53%	68	0.08%
	Font	31	8.09%	201	0.23%
	Application	32	8.36%	121	0.14%
	Testing Code	92	24.02%	3,766	4.30%
	Image	126	32.90%	10,212	11.66%
	Source Code	217	56.66%	49,680	56.70%
	Misc	236	61.62%	13,380	15.27%
	Archive	236	61.62%	535	0.61%
Subtotal				82,255	93.88%
Total				87,619	100%

learning algorithms to classify documentation related artifacts thus proposing a fully automated approach.

Gousios and Zaidman [20] introduce pullreqs, a dataset of almost 900 OSS GitHub projects and 350,000 pull requests that are used to study the pull request distributed development model. The main focus of their study is to understand the principles that guide pull-based development. Do et al. [13] design and construct an infrastructure to support controlled experimentation with testing techniques. The infrastructure includes artifacts (programs, versions, test cases, faults, and scripts) that enable researchers to perform controlled experimentation and replications. While these studies provide artifacts that can be used to improve the understanding of one aspect of OSS development, we complement these works by automatically detecting and categorizing multiple OSS artifacts, which can be beneficial to various OSS development activities.

Mirakhorli and Cleland-Huang [35] present an approach using ML to discover architectural tactics in code. The ML classifier is trained using code snippets extracted from OSS systems to automatically detect and categorize code-related files that contain ten common architectural tactics. Our study is not limited to a specific artifact type. Instead, we categorize both documentation related and non-documentation related artifacts, including but not limited to code related files.

Kalliamvakou et al. [26] conduct a study to understand the characteristics of the repositories and users in GitHub. They analyze a GHTorrent dump [17] to identify a set of perils that software engineering researchers should consider when utilizing GitHub repositories in their studies. While this study focuses on the projects and users characteristics, we analyze and classify software artifacts.

4.3 Text Classification

Linares-Vásquez et al. [29] extract APIs used by applications as attributes for categorization of their application domain and explore the performance of five different ML algorithms. The performance of the algorithms when using API methods and packages as features is compared to the performance of the algorithms when using terms from source code. Results show that the accuracy when using API methods and packages is as good as the accuracy when using terms

from source code. The best results, i.e., 0.67 average precision and 0.67 average recall, are achieved using SVM with a linear kernel.

Abu-Nimeh et al. [1] explore the performance of six ML algorithms for email text classification. They report that RF and logistic regression are among the top performing classifiers on their dataset, however all approaches achieve high performance. RF outperforms all other classifiers with an error rate of 7.72% when legitimate and phishing emails are weighted equally. Logistic regression outperformed all classifiers achieving the minimum weighted error rate of 3.82% when applying cost-sensitive measures.

Ye et al. [54] explore the performance of Naïve Bayes, SVM, and character based N-gram model for sentiment classification on text-based travel reviews. Their results indicate that SVM preforms the best, however, on large datasets all three algorithms achieve an accuracy of at least 0.80.

Pascarella and Bacchelli [37] propose a taxonomy and an automated approach using ML to classify comments in source code. They use the Naïve Bayes Multinomial algorithm, which is shown to achieve a weighted average TPR of 0.85 on the validation dataset and 0.74 on a cross-project validation dataset.

Similarity, our approach for overall software artifact classification achieves good performance, i.e., 0.82 weighted average recall and 0.75 on unseen dataset.

5 THREATS TO VALIDITY

This section discusses threats to validity that can affect our study.

Threats to *conclusion validity* relate to issues that could affect the ability to draw correct conclusion about relations between the treatment and the outcome of an experiment. One issue related to conclusion validity is the representativeness of sample used to validate the availability of documentation for open-source projects. We analyzed a random sample considering 95% confidence level and ± 5% margin of error. Another threat to validity might be related to the identification of categories for software artifacts. We use Cohen's kappa to ensure consistent rating between the two coders. Lastly, we report results using appropriate diagnostics for the performance of the ML algorithms, such as ROC and MCC and when discussing findings we keep into account acceptable ranges for ROC and MCC (i.e., ROC $\geq$ 0.5 and MCC $>$ 0).

Threats to *internal validity* concern the relation between the independent and dependent variables and factors that could have influenced the relation with respect to the causality. As explained in Section 2.3.3, ML algorithms are trained with manual tuning of some parameters. It is possible that better results could be obtained by employing automatic parameter tuning tools, such as Auto-WEKA [47]. This would simply mean that our results represent a lower-bound. Another threat is the calibration of the threshold used to identify files with insufficient information. Indeed, different values could have produced different results and could have affected the assessment of the proposed approach. Although the threshold is experimentally determined, this does not guarantee that the choice is optimal for the entire universe of software artifacts.

Threats to *construct validity* concern the relation between theory and observation. In this study, such threats are mainly due to measurement errors. As for precision and relative recall, the manual validation could be affected by subjectiveness of the coders or human error. If we conduct the experiment with different coders, the results might not be the same. To mitigate these threats, the oracle was created by two persons independently and, in case of different classification, a discussion, and if needed a third person was asked to perform the classification.

Threats to *external validity* concern the generalizability of the findings outside the experimental settings. Potential threats to external validity in this study include the selection of sampled open-source applications, which may not be representative of the studied population. To minimize this threat, we aimed to extract applications of various size and programming languages from GitHub. During the extraction process, we did not filter out projects based on project quality or project characteristics such as engineered projects as defined by Munaiah et al. [36]. Thus, the availability of artifacts in engineered projects might differ from what we observe in our random sample. Another potential threat is the manual nature of the feature creation process that could lead to overfitting the features to a particular dataset. However, to mitigate this threat we sample artifacts from over 300 projects which greatly reduces the chance of overfitting features. Additionally, we test the generalizability of our results on a second, unseen oracle, i.e., the testing dataset, to ensure acceptable accuracy of the classification model on new data. One further issue is the size of the dataset we used. Our results are reported on about 208 total data points (159 for training and validation, 59 for testing). The size of our dataset might be considered small. However, it is tied to the manual effort required to classify those software artifacts.

Threats to *reliability validity* concern the ability to replicate a study with the same data and to obtain the same results. We use open-source software projects whose source code is available. Moreover, we provide all necessary details to replicate the analysis in our online replication package [31].

6 CONCLUSION AND FUTURE WORK

This paper presents an automated approach to classify open-source software artifacts. The proposed approach is rigorously evaluated and results indicate that a combination of ML algorithms using ensemble learning outperforms individual classification techniques. Our approach is applied on 383 randomly selected open-source projects to investigate what types of software artifacts are generated in open-source projects. Results of this empirical study indicate that besides source code, around 14.88% of open-source projects contain other forms of artifacts such as requirements and design documents that are of interest to software engineering researchers.

Work-in-progress includes building an add-on for GitHub to identify and visualize which artifacts are in place and which ones are missing for a particular project. We envision that the add-on can be useful to evaluate the quality of a project and to determine whether it satisfies certain documentation standards. Additionally, we will investigate the use of information retrieval approaches for automatic feature extraction of Release Notes and Requirement Document artifacts and we will construct a larger oracle to improve the performance of the classifiers. Finally, we plan to expand the approach to other forms of artifacts and explore multi-label text classification techniques.

REFERENCES

[1] Saeed Abu-Nimeh, Dario Nappa, Xinlei Wang, and Suku Nair. 2007. A comparison of machine learning techniques for phishing detection. In *Proceedings of the Anti-phishing Working Groups eCrime Researchers Summit*. 60–69.

[2] Pooyan Behnamghader, Duc Minh Le, Joshua Garcia, Daniel Link, Arman Shahbazian, and Nenad Medvidovic. 2017. A large-scale study of architectural evolution in open-source software systems. *Journal of Empirical Software Engineering (EMSE)* 22, 3 (2017), 1146–1193.

[3] David M Blei, Andrew Y Ng, and Michael I Jordan. 2003. Latent dirichlet allocation. *Journal of machine Learning research* 3, Jan (2003), 993–1022.

[4] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.

[5] Matthieu Caneill, Daniel M. Germán, and Stefano Zacchiroli. 2017. The Debsources Dataset: two decades of free and open source software. *Journal of Empirical Software Engineering* 22, 3 (2017), 1405–1437.

[6] Jane Cleland-Huang, Adam Czauderna, Marek Gibiec, and John Emenecker. 2010. A Machine Learning Approach for Tracing Regulatory Codes to Product Specific Requirements. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 155–164.

[7] Jane Cleland-Huang, Raffaella Settimi, Oussama BenKhadra, Eugenia Berezhanskaya, and Selvia Christina. 2005. Goal-centric Traceability for Managing Non-functional Requirements. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 362–371.

[8] Jacob Cohen. 1988. *Statistical power analysis for the behavioral sciences*.

[9] Al Danial. 2018. cloc. (2018). Retrieved March 19, 2018 from https://github.com/AlDanial/cloc

[10] DataTypes.net. 2018. The most recent filename extension database. (2018). Retrieved March 19, 2018 from https://datatypes.net/

[11] Andrea De Lucia, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. 2006. Can Information Retrieval Techniques Effectively Support Traceability Link Recovery?. In *Proceedings of the International Conference on Program Comprehension (ICPC)*. 307–316.

[12] Diana Diaz, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Silvia Takahashi, and Andrea De Lucia. 2013. Using code ownership to improve IR-based Traceability Link Recovery. In *Proceedings of the International Conference on Program Comprehension (ICPC)*. 123–132.

[13] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Journal of Empirical Software Engineering (EMSE)* 10, 4 (2005), 405–435.

[14] Susan T Dumais. 2004. Latent semantic analysis. *Annual review of information science and technology* 38, 1 (2004), 188–230.

[15] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen. 2014. Mining Billions of AST Nodes to Study Actual and Potential Usage of Java Language Features. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. 779–790.

[16] Joshua Garcia, Ivo Krka, Chris Mattmann, and Nenad Medvidovic. 2013. Obtaining Ground-Truth Software Architectures. *Proceedings of the International Conference on Software Engineering (ICSE)* (2013).

[17] GHTorrent. 2018. Downloads 2014-01-02. (2018). Retrieved March 19, 2018 from http://ghtorrent-downloads.ewi.tudelft.nl/mysql/mysql-2014-01-02.tar.gz

[18] Michael W. Godfrey and Qiang Tu. 2000. Evolution in open source software: A case study. In *Proceedings of the International Conference on Software Maintenance (ICSM)*. 131–142.

[19] Georgios Gousios. 2013. The GHTorrent dataset and tool suite. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*. Piscataway, NJ, USA, 233–236.

[20] Georgios Gousios and Andy Zaidman. 2014. A Dataset for Pull-based Development Research. In *Proceedings of the Conference on Mining Software Repositories (MSR) (MSR 2014)*. 368–371.

[21] Jin Guo, Natawut Monaikul, Cody Plepel, and Jane Cleland-Huang. 2014. Towards an Intelligent Domain-specific Traceability Solution. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. 755–766.

[22] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. 2009. The WEKA Data Mining Software: An Update. *SIGKDD Explorations Newsletter* 11, 1 (2009), 10–18.

[23] Kevin A Hallgren. 2012. Computing inter-rater reliability for observational data: an overview and tutorial. *Tutorials in quantitative methods for psychology* 8, 1 (2012), 23.

[24] GitHub Inc. 2018. GitHub. (2018). Retrieved March 19, 2018 from https://github.com/

[25] Thorsten Joachims. 1998. Text categorization with support vector machines: Learning with many relevant features. In *Proceedings of the European conference on machine learning*. 137–142.

[26] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. 2016. An in-depth study of the promises and perils of mining GitHub. *Journal of Empirical Software Engineering* 21, 5 (2016), 2035–2071.

[27] Richard J Landis and Gary G Koch. 1977. The measurement of observer agreement for categorical data. *Biometrics* (1977), 159–174.

[28] Gernot A. Liebchen and Martin Shepperd. 2008. Data Sets and Data Quality in Software Engineering. In *Proceedings of the International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*. 39–44.

[29] Mario Linares-Vásquez, Collin McMillan, Denys Poshyvanyk, and Mark Grechanik. 2014. On using machine learning to automatically classify software applications into domain categories. *Journal of Empirical Software Engineering (EMSE)* 19, 3 (2014), 582–618.

[30] Sugandha Lohar, Sorawit Amornborvornwong, Andrea Zisman, and Jane Cleland-Huang. 2013. Improving Trace Accuracy Through Data-driven Configuration and Composition of Tracing Features. In *Proceedings of the Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 378–388.

[31] Yuzhan Ma. 2018. Online Replication Package. (March 2018). https://github.com/MaggieMa21/SoftwareArtifactsClassification_MSR2018

[32] Larry M Manevitz and Malik Yousef. 2001. One-class SVMs for document classification. *Journal of machine Learning research* 2, Dec (2001), 139–154.

[33] Brian W Matthews. 1975. Comparison of the predicted and observed secondary structure of T4 phage lysozyme. *Biochimica et Biophysica Acta (BBA* 2, 405 (1975), 442–451.

[34] Andrew McCallum, Kamal Nigam, et al. 1998. A comparison of event models for naive bayes text classification. In *AAAI-98 workshop on learning for text categorization*, Vol. 752. 41–48.

[35] Mehdi Mirakhorli and Jane Cleland-Huang. 2016. Detecting, Tracing, and Monitoring Architectural Tactics in Code. *Transactions on Software Engineering (TSE)* 42, 3 (2016), 205–220.

[36] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating GitHub for engineered software projects. *Journal of Empirical Software Engineering (EMSE)* 22, 6 (2017), 3219–3253.

[37] Luca Pascarella and Alberto Bacchelli. 2017. Classifying code comments in Java open-source software systems. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*. 227–237.

[38] Dan Port, Allen Nikora, Jane Huffman Hayes, and LiGuo Huang. 2011. Text Mining Support for Software Requirements: Traceability Assurance. In *Proceedings of the Hawaii International Conference on System Sciences (HICSS)*. 1–11.

[39] Mona Rahimi, Mehdi Mirakhorli, and Jane Cleland-Huang. 2014. Automated extraction and visualization of quality concerns from requirements specifications. In *Proceedings of the International Requirements Engineering Conference (RE)*. 253–262.

[40] KR Remya and JS Ramya. 2014. Using weighted majority voting classifier combination for relation classification in biomedical texts. In *Proceedings of the International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT)*. 1205–1209.

[41] Gregorio Robles, Jesus M Gonzalez-Barahona, and Juan Julian Merelo. 2006. Beyond source code: the importance of other artifacts in software development (a case study). *Journal of Systems and Software* 79, 9 (2006), 1233–1248.

[42] Gregorio Robles, Jesus M Gonzalez-Barahona, and Juan Luis Prieto. 2006. Assessing and evaluating documentation in libre software projects. In *Workshop on Evaluation Frameworks for Open Source Software (EFOSS)*.

[43] Gerard Salton, Anita Wong, and Chung-Shu Yang. 1975. A vector space model for automatic indexing. *Commun. ACM* 18, 11 (1975), 613–620.

[44] Martin Shepperd, Qinbao Song, Zhongbin Sun, and Carolyn Mair. 2013. Data Quality: Some Comments on the NASA Software Defect Datasets. *Transactions on Software Engineering (TSE)* 39, 9 (2013), 1208–1215.

[45] Karen Sparck Jones. 1972. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation* 28, 1 (1972), 11–21.

[46] Hakim Sultanov and Jane Huffman Hayes. 2013. Application of reinforcement learning to requirements engineering: requirements tracing. In *Proceedings of the International Requirements Engineering Conference (RE)*. 52–61.

[47] Chris Thornton, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. 2013. Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 847–855.

[48] Kai Tian, Meghan Revelle, and Denys Poshyvanyk. 2009. Using latent dirichlet allocation for automatic categorization of software. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*. 163–166.

[49] SL Ting, WH Ip, and Albert HC Tsang. 2011. Is Naive Bayes a good classifier for document classification. *International Journal of Software Engineering and Its Applications* 5, 3 (2011), 37–46.

[50] Bülent Üstün, W. J. Melssen, and Lutgarde M C Buydens. 2006. Facilitating the application of Support Vector Regression by using a universal Pearson VII function based kernel. *Chemometrics and Intelligent Laboratory Systems* 81, 1 (2006), 29–40.

[51] Christopher Vendome, Mario Linares-Vásquez, Gabriele Bavota, Massimiliano Di Penta, Daniel German, and Denys Poshyvanyk. 2017. Machine learning-based detection of open source license exceptions. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 118–129.

[52] Elaine J Weyuker, Thomas J Ostrand, and Robert M Bell. 2010. Comparing the effectiveness of several modeling methods for fault prediction. *Journal of Empirical Software Engineering (EMSE)* 15, 3 (2010), 277–295.
[53] Suresh Yadla, Huffman Jane Hayes, and Alex Dekhtyar. 2005. Tracing requirements to defect reports: an application of information retrieval techniques. *Innovations in Systems and Software Engineering (ISSE)* 1, 2 (2005), 116–124.
[54] Qiang Ye, Ziqiong Zhang, and Rob Law. 2009. Sentiment classification of online reviews to travel destinations by supervised machine learning approaches. *Expert systems with applications* 36, 3 (2009), 6527–6535.
[55] Jiaxin Zhu, Minghui Zhou, and Audris Mockus. 2014. Patterns of Folder Use and Project Popularity: A Case Study of Github Repositories. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*. Article 30, 4 pages.
[56] Waleed Zogaan, Ibrahim Mujhid, Joanna C. S. Santos, Danielle Gonzalez, and Mehdi Mirakhorli. 2017. Automated Training-set Creation for Software Architecture Traceability Problem. *Journal of Empirical Software Engineering (EMSE)* 22, 3 (2017), 1028–1062.
[57] Waleed Zogaan, Palak Sharma, Mehdi Mirahkorli, and Venera Arnaoudova. 2017. Datasets from Fifteen Years of Automated Requirements Traceability Research: Current State, Characteristics, and Quality. In *Proceedings of the International Requirements Engineering Conference (RE)*. 110–121.