New Trends in Intelligent Software Methodologies, Tools and Techniques
H. Fujita et al. (Eds.)
IOS Press, 2017

doi:10.3233/978-1-61499-800-6-88

Software Categorization Using Low-Level Distributional Features

Zalán BODÓ [a], Bipin INDURKHYA [b]

[a] *Romanian Institute of Science and Technology, Cluj-Napoca, Romania / Faculty of Mathematics and Computer Science, Babeş–Bolyai University, Cluj-Napoca, Romania; bodo@rist.ro*

[b] *Romanian Institute of Science and Technology, Cluj-Napoca, Romania / Department of Cognitive Science, Jagiellonian University, Cracow, Poland; indurkhya@rist.ro*

Abstract. In recent years, there has been a growing interest in applying deep learning techniques for automatic generation of software. To achieve this ambitious objective, a number of smaller research goals need to be reached, one of which is automatic categorization of software, used in numerous tasks of software intelligence. We present here an approach to this problem using a set of low-level features derived from lexical analysis of software code. We compare different feature sets for categorizing software and also apply different supervised machine learning algorithms to perform the classification task. The representation allows us to identify the most relevant libraries used for each class, and we use the best-performing classifier to accomplish this. We evaluate our approach by applying it to categorize popular Python projects from Github.

Keywords. software categorization, text categorization, software intelligence, distributional features, feature selection

1. Introduction

In recent years, there has been much interest in applying the techniques of machine learning for modeling the process of software generation with the goal of being able to generate software automatically or semi-automatically [9,1,8]. We are also conducting research in this area, which is focused on using deep-learning and cognitive modeling approaches for software generation. In a recent publication of our research group [19], we used a visual representation of the memory dynamics of programs to cluster and classify sorting algorithms. Towards the grandiose goal of automatic software generation, we are exploring the idea of using low-level static features of software for categorization, where low-level means that only lexical analysis of the programs is performed. Such features in turn can be used to make recommendations to programmers about software libraries anticipated as most likely to be important. In this paper, we present our preliminary results of this research.

From amongst the applicative goals of the above-mentioned research project, we can mention the development of an intelligent system that assists programmers by autocomplete suggestions, best practice and other similar recommendations [26,12]. One such important recommendation would be a list of the most relevant libraries for the ac-

tual software project, given some keywords or a textual description. Automatic labeling can also be of great help in searching for similar projects. Good representations can lead to good similarity measure, which is a central problem of automatic code generation. Low-level features are considered to be important, because a wide range of programming languages can be efficiently covered using a relatively small set of regular expressions.

In this paper we present the selection of useful features using only lexical analysis of program code; these features are then used to automatically label software and also to group them into categories.

The paper is structured as follows: Section 2 enumerates some of the existing categorization approaches for software projects. In Section 3 we present our approach: the low-level representation of software for categorization and feature identification. Section 4 describes the method used for identifying the most relevant features. In Section 5, we present our experiments, results and discussion. Finally, Section 6 presents the conclusions and possible extensions of this research.

2. Categorizing Software

Software categorization is an important problem of software intelligence systems [5]: categorization is usually performed using statistical machine learning techniques; but, before doing this, a suitable representation of software is needed. In [5], the authors focus on four related problems that use categorization (malware, plagiarism, theft and clone detection), and present possible representations and similarity metrics or distances for measuring software proximity. For example, a software can be represented as a string of the source code, and distance can be measured using the longest common subsequence algorithm; for vector-based representation one can use the cosine similarity, for sets the Jaccard index, and for graphs or trees the graph edit distance.

The multi-label approach MUDABlue presented in [15] uses as features the identifiers (function, class, variable, etc. names), and applies latent semantic analysis (LSA) for dimensionality reduction and better representation [7].

In [14], three different approaches are compared: a similarity-based technique using the number of code clone lines, a supervised learning system using decision trees and character-level trigrams of filenames of the source code files as features, and an LSA-based approach described in [15].

The work presented in [18] traces API calls in binary executables, and uses these as features for classification. It compares three machine learning algorithms, support vectors machines (SVMs), decision trees and naive Bayes classifiers. A great advantage of their method is that it can be applied also on closed-source software repositories. Feature selection is accomplished using expected entropy loss.

In [24], latent Dirichlet allocation was used for categorization, using identifiers and (tokenized) comments as features.

From the literature one can observe that categorization is usually done using some *low-level* features. This can be explained by the fact that in many application domains, programming paradigms do not change; therefore using control or data flow graphs might not explain the expected effects. Another disadvantage of using more complex analysis and program representation is the increased computational demand and the difficulty to adapt to new programming languages. However, before moving ahead with such assump-

tions, a thorough investigation of these representations and the underlying challenges is needed.

We compare here different low-level representations for software categorization, and use the trained classifier to identify relevant features for every category. The described system can be easily extended or modified to support a wide range of programming languages.

3. Categorization Using Low-Level Features

Comparing different feature sets for representing software project, our goal is to demonstrate empirically that using the distribution of the used libraries[1], one can obtain a better representation than using all lexical tokens as in information retrieval; in other words, the distribution of the used libraries induces a better similarity measure.[2] Also, besides improving the prediction performance of the system, we work in a significantly lower-dimensional space. The system built in this way can be used for multiple problems: for automatically categorizing software projects and also for identifying the most influential modules for different categories.

In text categorization, and more generally in information retrieval [23,17], the *bag-of-words* model is one of the most popular representation; despite its simplicity, it performs surprisingly well. The simplicity lies in the independence assumption regarding the words: no language grammar or text semantics is taken into consideration, not even the ordering of the words. Spam filtering provides a good example domain [2] where the occurrences of certain words or simple combinations can be good indicators of the document's topics.

A bag-of-words equivalent representation for software projects would be taking all the tokens returned by the lexical analyzer and using them as features. Of course, depending on the programming language, some of the evidently irrelevant tokens can be neglected from the beginning: for example the symbols representing code blocks, separators of the variables in a method call, braces, etc. We call these features *low-level*, since from the classical phases of program analysis (lexical, syntactic and semantic) [16] we perform only the first step of analysis.

In this study, we focus on the software projects written in Python. For processing the Python programs, we defined the following feature sets:

- FS_1 Using only `token.NAME` token types (`tokenize` and `token` Python modules): this type represents all named tokens like variable, method, class, etc. names. The idea behind this representation is that operators, constants, etc. are probably less relevant when comparing two programs or projects. This representation is similar to [15].
- FS_2 Using all the tokens retrieved by the Python tokenizer, except the following four token types: `token.N_TOKENS` (comments), `token.NEWLINE`, `token.INDENT`, `token.DEDENT`.

[1] Throughout the paper we will use the terms library, module and import interchangeably.

[2] The selection of specific keyword types is somewhat equivalent to part-of-speech tagging-based term selection in information retrieval and natural language processing.

FS_3 Using only the imported packages and modules with their names as features. The underlying idea is that if one wants to measure similarity between software projects, it might be a good idea to examine the imported modules, since projects having similar profiles should use about the same set of modules. Evidently, observing only the imports is just a rough approach, a better method should also consider the usage patterns of the imported libraries. This is partly accomplished by the next feature set.

FS_4 Using the imported modules and the trace of usage frequencies. The central question here is how to trace the usage of different modules? Since we only perform lexical analysis of the code, we can only observe method calls and class instantiations. Hence, after obtaining the names of the imported modules, we search for method and class names in the code to find calls, object creations and count these too. This approach resembles the method of [18].

In FS_3 and FS_4, if an imported module was found, the top-level and all possible submodules are similarly included in the feature set.[3]

4. Feature Identification Using Support Vector Machines

Feature selection in machine learning means dimensionality reduction by selecting some of the most relevant dimensions or features of the input space to represent our data [10]. Feature extraction, however, uses the input features and some function of these to create new ones; see for example PCA or kernel PCA [22]. Selection is usually preferred over extraction if interpretation of the reduced dimensions is needed. Selecting features is performed prior to the application of the learning algorithm to filter out irrelevant dimensions and noise, thereby improving the performance of the method. However, sometimes the goal is not only to build a good predictor, but to find the most important features, based on which (good) predictions can be made—we call this *feature identification*.

Based on the experiments presented in Section 5, and considering the reported results from the literature [11], SVMs were chosen for performing feature identification.

In linear classifiers, the decision function has the following form:

$$f(\mathbf{x}) = \mathbf{w}'\mathbf{x} + b \tag{1}$$

where $\mathbf{w}$ is the normal vector to the separating hyperplane, also called the *weight vector*, while b is the offset. In such models, the weights can be considered as importance factors for the features: the higher the weight (in absolute value), the more important the corresponding feature. Thus, if n features are needed, we can simply choose those having the n highest weights [11].

In SVMs the optimal parameters $(\mathbf{w}, b)$ are found by maximizing the margin of the separating hyperplane, i.e. $2/\|\mathbf{w}\|$. In order to deal with non-linearity, kernel functions are introduced, that is the input vectors are implicitly mapped to a so-called feature space using $k(\mathbf{x}, \mathbf{z}) = \phi(\mathbf{x})'\phi(\mathbf{z})$, where k and ϕ denotes the kernel function and the feature mapping, respectively. The above procedure is not limited to the linear case: in order to

[3] More precisely, if `x.y.z` was imported, we include `x`, `x.y` and `x.y.z` as well; when searching for a method call or class instantiation, we similarly update the occurrence of every higher-level module.

find the most relevant features, one can observe the change in the resulting weight vector as described by SVM Recursive Feature Elimination [11,21].

SVM is originally a binary classification algorithm, but multi-class formulations of the maximum margin separation principle also exist; see for example [25]. However, because of efficiency reasons, different combination schemes of binary classifiers are usually applied as *one-versus-rest* and *one-versus-one* approaches [13]. The task is to select the most relevant features for every category. In one-versus-rest, having c classes, c classifiers are built: each model is trained by considering the samples of the i-th class as positives and all the other samples as negatives. In this case we can select in a straightforward manner the highest weighted feature for each class.

For the one-versus-one scheme, we use a voting system. In this approach, a separate classifier is built for every class pair, and every classifier will possess an initially empty bag of features. For every weight vector, we select the highest n positive and lowest n negative weights, and the corresponding feature counts are incremented for the actual positive and negative classes. After going through all $c(c-1)/2$ weight vectors, we can rank the most relevant features for every class by taking into account the received votes.

5. Experiments, Results and Discussion

5.1. The Dataset

The first problem needed to be tackled was to collect usable data for the experiments. Usable data in our case means open software code and labels, categories attached to the projects. For this we studied three of the most popular software project hosting sites: Github[4], Bitbucket[5] and Sourceforge[6]. In order to gather statistics about the projects, all of the above services offer REST APIs. After studying the provided functionalities we discovered the following:

- None of the APIs offer topic information about the projects.
- Only Sourceforge was designed from the beginning to support project categories; here the projects are organized into a hierarchy, but the REST API cannot be used to retrieve this.
- Only Github and Bitbucket offers the possibility to query projects based on different properties (e.g. number of stargazers, date of creation, etc.).
- There are no category or topic labels on Bitbucket at all.
- On 31.01.2017, or possibly earlier, but announced on this date, Github introduced topic labels for projects.[7] One can add topics on the main page of the project using the "Add topics" links. It is also mentioned in the post that suggested topics can appear when assigning a topic to a repository, and this is the result of the application of machine learning and natural language processing methods. These keywords can be changed—added, deleted, modified—at any time, not only when creating a new project. However, probably since this is a newly introduced feature,

[4] `https://github.com/`

[5] `https://bitbucket.org/`

[6] `https://sourceforge.net/`

[7] Introducing Topics, `https://github.com/blog/2309-introducing-topics`, date of retrieval: 10.04.2017

Figure 1. Word cloud showing the labels attached to Python projects in our Github-based dataset.

only a fraction of the existing projects have category labels, moreover the API does not support querying these yet.

Taking into consideration the above findings, we decided to use Github and utilize the existing topic tags by retrieving and processing the HTML pages of the projects.

Currently, Github hosts more than 19.4 million active repositories[8], only a fraction of which represent public (open source) projects. To cut off less popular projects with no relevant usage or activity, we narrowed down the number of open projects using the stargazers count: as of 27.02.2017, using the search qualifier `stars:>=100`, we managed to collect 78 518 metadata records about Github-hosted projects. The metadata database contains the following fields: 0: language, 1: total downloads count, 2: commit counts for the last 52 weeks, 3: stargazers (stars) count, 4: watchers count, 5: list of topics, 6: description of project, 7: number of contributors, 8: total number of contributions, 9: date of project creation.[9]

In this stage of the research, we focused on one programming language only: we selected those projects for which Python was given as primary language, resulting in 7590 projects. This number decreased further when selecting only labeled projects, arriving at a final number of 1351. Then different labels were counted and some of these categories were selected for the experiments described below.

5.2. Experimental Results and Discussion

For performing the categorization experiments, we selected some of the most frequent labels. The different labels found in our database are shown by the word cloud in Figure 1. Excluding the labels denoting programming languages, we defined the following three experiments:

(1) binary classification using 'machine-learning' (67) and 'security' (54 projects) labels;
(2) binary classification using 'linux' (45) and 'windows' (22 projects) labels;
(3) multi-class classification using the first 16 most frequent labels: 'django' (101), 'machine-learning' (67), 'deep-learning' (62), 'tensorflow' (60), 'security' (54), 'linux' (45), 'cli' (44), 'flask' (36), 'data-science' (29), 'api' (25), 'terminal' (25), 'visualization' (24), 'git' (24), 'windows' (22), 'aws' (21), 'music' (20 projects).

[8]The state of the Octoverse 2016, `https://octoverse.github.com/`, date of retrieval: 10.04.2017
[9]The dataset was made available for download at `http://www.cs.ubbcluj.ro/~zbodo/github.html`.

Table 1. 'machine-learning' versus 'security' classification results.

Classifier	Normalization	FS_1	FS_2	FS_3	FS_4
SVM	n.n.	73.50	59.67	75.90	75.16
	ℓ_1	55.33	67.14	55.33	62.63
	ℓ_2	86.76	67.14	**90.86**	**90.06**
Random forest	n.n.	82.95	63.23	81.28	82.28
	ℓ_1	83.28	62.63	81.94	77.15
	ℓ_2	84.28	63.23	82.46	79.96
Decision tree	n.n.	**88.36**	52.08	**89.20**	87.53
	ℓ_1	87.63	53.73	**90.86**	**90.03**
	ℓ_2	86.76	53.73	**90.86**	**90.03**
k-NN	n.n.	69.43	65.60	81.03	81.76
	ℓ_1	69.30	65.60	73.43	76.80
	ℓ_2	76.80	64.28	**89.23**	87.56

For performing the categorization, we used the following classifiers, using the *scikit-learn* Python library[10]:

(a) linear SVM [3] (one-versus-one scheme for the multi-class case)
(b) random forest [4] (10 estimators)
(c) decision tree [20] (Gini impurity)
(c) k-nearest neighbors [6] ($k = 5$)

The collected data was not separated into training and test sets, but the 5-fold cross validation was used to measure the performance. The parameters of the used models, however, were not optimized.

The present classification problem is a *multi-label* one, that is an example can have more than one label: a project can belong to 'machine-learning', 'big-data', 'data-science', etc. category as well. Therefore, every project was included as many times as the number of different labels it had. However, we did not take into account the multi-label property when considering the output of the classifiers. Similarly, no advanced performance indicators such as precision, recall, F-measure or ROC AUC [17] were used; the results presented in Table 1–3 are 5-fold cross-validation accuracy results. In every table, we highlighted the results having one of the three highest integer parts.

Table 4 shows the relevant library features found by one-versus-one linear SVM.[11] In the table, only the first ten distinct top-level module/package names were left in decreasing order of their received votes from the binary classifiers. If less than ten libraries are listed, this does not mean error but zero votes for the other modules.

The feature sets, after processing all the downloaded Python projects (1351), have the following cardinalities:

FS_1	1 478 134
FS_2	5 239 581
FS_3	226 211
FS_4	226 211

[10] `http://scikit-learn.org/`, version 0.18.1.

[11] We performed experiments also using one-versus-rest SVMs with seemingly inferior results, therefore we decided not to publish it here.

Table 2. 'linux' versus 'windows' classification results.

Classifier	Normalization	FS_1	FS_2	FS_3	FS_4
SVM	n.n.	59.67	59.67	61.09	59.56
	ℓ_1	**67.14**	**67.14**	**67.14**	**67.14**
	ℓ_2	**67.14**	**67.14**	**65.71**	**65.60**
Random forest	n.n.	63.82	62.92	62.63	62.63
	ℓ_1	62.63	62.06	62.63	62.63
	ℓ_2	62.63	62.61	63.63	62.63
Decision tree	n.n.	47.80	52.08	58.24	58.13
	ℓ_1	50.87	53.73	56.81	50.87
	ℓ_2	55.38	53.73	56.81	47.69
k-NN	n.n.	**65.71**	**65.60**	**65.60**	**67.14**
	ℓ_1	58.13	**65.60**	53.73	59.56
	ℓ_2	**64.17**	**64.28**	**65.60**	**67.14**

Table 3. Results of the 16-class experiment.

Classifier	Normalization	FS_1	FS_2	FS_3	FS_4
SVM	n.n.	34.14	31.41	35.21	36.27
	ℓ_1	15.32	15.32	24.27	29.58
	ℓ_2	37.63	23.82	**43.85**	**44.15**
Random forest	n.n.	36.63	36.20	37.48	38.85
	ℓ_1	37.21	36.38	38.84	37.94
	ℓ_2	37.60	35.60	38.24	38.85
Decision tree	n.n.	36.41	35.05	33.53	35.05
	ℓ_1	35.35	35.05	35.96	38.54
	ℓ_2	37.02	35.05	36.57	37.33
k-NN	n.n.	25.79	17.44	35.05	**39.75**
	ℓ_1	29.74	26.55	32.61	35.35
	ℓ_2	31.86	27.31	34.43	36.56

The sets F_3 and F_4 are, indeed, smaller than the "naive" ones by an order of magnitude. The equal dimensionality of F_3 and F_4 is due to the fact that F_4 does not introduce new features, but finds a few more occurrences of the same features.

As expected, the results in Table 1–3 show that FS_1 performs better than FS_2, while FS_3 and FS_4 perform better than the first two sets. It is also evident from the experiments that discriminating between 'machine-learning' and 'security' is a much simpler problem than differentiating between 'linux' and 'windows', both denoting operating systems, and therefore having many common properties. This can also be seen from Table 4, showing four common modules within the first ten most relevant features. The 16-class experiments might not seem very promising at the first sight, but one has to keep in mind that a random classifier—taking into account the class distribution as well—would produce an accuracy of about 8%. One also has to take into account the fact that the top 16 classes include the 'linux' and 'windows' categories too, and might be other similar cases as well, which evidently deteriorates the overall accuracy. In the next section, we discuss possible future improvements of the present system.

Table 4. Relevant libraries found by feature identification.

django	`django, dotenv, configurations, signal, os, pyinstrument, sys, models`
machine-learning	`digits, tensorflow, os, numpy, sklearn, google, scipy, keras, __future__, datetime`
deep-learning	`gym_starcraft, torchcraft_py, keras, cv2, tensorflow, random, numpy, config, layers, os`
tensorflow	`time, tensorflow, six, keras, tarfile, __future__, numpy, sys, distkeras, re`
security	`time, socket, metrics, json, anchore, subprocess, afl_utils, sys, glob, urllib2`
linux	`gdb, Xlib, sys, os, time, click, subprocess, pygments, prompt_toolkit, ctypes`
cli	`json, sys, os, yaml, colorama, cliff, click, subprocess, logging, prompt_toolkit`
flask	`flask, enum, redis, json, requests, zappa, click, config, datetime, nets`
data-science	`cliff, datetime, html, re, feedparser, progressbar, time, locate_datasets, tensorlayer, pylab`
api	`urllib2, bs4, PIL, json, re, slacker, os, tumblpy, quora, exceptions`
terminal	`termfeed, colorama, os, urllib, re, PIL, selenium, sys, setuptools, time`
visualization	`subprocess, tempfile, json, logging, numpy, ast, setuptools, Constant, sys, gdal`
git	`git, email, sys, subprocess, colorama, git_multimail, klaus, ssl, pbr, lib`
windows	`Xlib, threading, ctypes, sublime, re, os, pyxhook, sys, cget, qtodotxt`
aws	`boto, argparse, util, sys, chalice, aq, re, time, django, nose`
music	`setuptools, re, random, database, unittest, mutagen, musthe, librosa, os, requests`

The fact that FS_4 induces no significant performance improvement over FS_3 is good news for us: even the naive trace implemented in FS_4 would be quite complicated to realize in languages like C or C++. In these programming languages, external knowledge is needed to determine the method or class to which a library belongs.

A threat to the validity of the experiments is whether the sample we were working on is representative. We tried to minimize this by not applying any filtering when selecting the software projects, which were filtered only by the number of Github stargazers (to filter out small, unpopular projects) and programming language. However, We applied another filter by category labels, omitting the projects without such keywords. At this stage, we did not want to introduce additional noise in our system by using automatically extracted keywords (e.g. from project descriptions). Another threat can be in the selection of categories. To minimize this, we selected the most frequent categories, having at least 20 examples. Only category labels denoting programming languages were filtered out, thereby arriving at 16 categories.

6. Conclusion and Future Work

We presented here our preliminary results in studying open source software repositories by comparing four low-level distributional representations for categorizing software. The resulting classifier can also be used to identify relevant libraries for specific software categories.

The classifier built can be used for automatically categorizing projects, or simply recommending topic labels, whilst categories can greatly help when searching for software projects. Also, the generated vectors can be used directly for calculating software resemblance using the cosine similarity [17]. Given simple keywords or a textual description of the project topics, our system is able to recommend libraries for the developers; this feature can be extremely helpful in situations when programmers have less experience in the given domain.

As mentioned in Section 5.1, we only used labeled projects in the experiments. Using the short textual description of the projects, we can assign labels to originally unlabeled software, using for example the *tf-idf* weighting [23,17] or other keyword selection methods.

Because category predictions should not be influenced by software length, we experimented different normalization schemes, as shown in the tables of Section 5.2. Before normalization, we could use other prior statistical information, e.g. idf, regarding the imported libraries.

As described in Section 3, for an import we included the top-level and all possible submodules. Prior to categorization or any other operation on the generated vectors, we could filter out the submodules and leave only the top-level imports. Presently, local modules are also included in the feature set, however, these could be safely removed, being only noise in this representation.

Acknowledgements

This work was supported by the European Regional Development Fund and the Romanian Government through the Competitiveness Operational Programme 2014–2020, project ID P_37_679, MySMIS code 103319, contract no. 157/16.12.2016.

References

[1] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

[2] E. Blanzieri and A. Bryl. A survey of learning-based techniques of email spam filtering. *Artificial Intelligence Review*, 29(1):63–92, 2008.

[3] B. Boser, I. Guyon, and V. Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152. ACM, 1992.

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

[5] S. Cesare and Y. Xiang. *Software similarity and classification*. Springer Science & Business Media, 2012.

[6] T. M. Cover and P. E. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, IT-13(1):21–7, January 1967.

[7] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American society for information science*, 41(6):391, 1990.

[8] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A.-r. Mohamed, and P. Kohli. Robustfill: Neural program learning under noisy i/o. *arXiv preprint arXiv:1703.07469*, 2017.

[9] A. Graves, G. Wayne, and I. Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

[10] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *Journal of machine learning research*, 3(Mar):1157–1182, 2003.

[11] I. Guyon, J. Weston, S. Barnhill, and V. Vapnik. Gene selection for cancer classification using support vector machines. *Machine learning*, 46(1):389–422, 2002.

[12] A. E. Hassan and T. Xie. Software intelligence: the future of mining software engineering data. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 161–166. ACM, 2010.

[13] C.-W. Hsu and C.-J. Lin. A comparison of methods for multiclass support vector machines. *IEEE transactions on Neural Networks*, 13(2):415–425, 2002.

[14] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue. On automatic categorization of open source software. In *3rd Workshop on Open Source Software Engineering*, pages 79–83, 2003.

[15] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue. Mudablue: An automatic categorization system for open source repositories. *Journal of Systems and Software*, 79(7):939–953, 2006.

[16] M. Lam, R. Sethi, J. Ullman, and A. Aho. Compilers: Principles. *Techniques, and Tools*, 2006.

[17] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to information retrieval*. Cambridge University Press, 2008.

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

[19] C. Perţicaş, B. Indurkhya, R. V. Florian, and L. Csató. Finding patterns in visualizations of programs. To appear in the Proceedings of the 28th Annual Workshop of Psychology of Programming Interest Group (PPIG 2017), Delft (Netherlands), July 1–2, 2017.

[20] J. R. Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[21] A. Rakotomamonjy. Variable selection using svm-based criteria. *Journal of machine learning research*, 3(Mar):1357–1370, 2003.

[22] B. Schölkopf, A. Smola, and K.-R. Müller. Nonlinear component analysis as a kernel eigenvalue problem. *Neural computation*, 10(5):1299–1319, 1998.

[23] F. Sebastiani. Machine learning in automated text categorization. *ACM Computing Surveys*, 34(1):1–47, 2002.

[24] K. Tian, M. Revelle, and D. Poshyvanyk. Using latent dirichlet allocation for automatic categorization of software. In *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*, pages 163–166. IEEE, 2009.

[25] J. Weston and C. Watkins. Support vector machines for multi-class pattern recognition. In *ESANN*, volume 99, pages 219–224, 1999.

[26] T. Xie, S. Thummalapenta, D. Lo, and C. Liu. Data mining for software engineering. *Computer*, 42(8), 2009.