A Comparison of Machine Learning-Based Text Classifiers for Mapping Source Code to Architectural Modules

Alexander Florean[1], Laoa Jalal[1], Zipani Tom Sinkala[1] and Sebastian Herold[1]

[1]Department of Mathematics and Computer Science, Karlstad University, Sweden

Abstract

A mapping between a system's implementation and its software architecture is mandatory in many architecture consistency checking techniques. Creating such a mapping manually is a non-trivial task for most complex software systems. Machine learning-based text classification may be an highly effective tool for automating this task. How to make use of this tool most effectively has not been thoroughly investigated yet.

This article presents a comparative analysis of three classifiers applied to map the implementations of five open-source systems to their architectures. The performance of the classifiers is evaluated for different extraction and preprocessing settings as well as different training set sizes.

The results suggest that Logical Regression and Support Vector Machines both outperform Naive Bayes unless information about coarse-grained implementation structures cannot be exploited. Moreover, initial manual mappings of more than 15% of all source code files, or 10 files per module, do not seem to lead to a significantly better classification.

Keywords

software architecture consistency, code-to-architecture mapping, text classification, machine learning

1. Motivation

Software architecture degradation is the phenomenon of the implementation of a software system diverging from the intended software architecture [1]. The potential consequences of this divergence include a decay of maintainability as well as the decreased ability of the system to meet other desired quality properties. Expensive system re-engineering or discontinuations of software products can be the consequences [2, 3, 4, 5].

One approach to combat software architecture degradation is software architecture consistency checking [6]. The core idea of these techniques is to implement frequent checks for inconsistencies between the intended software architecture and the current implementation of a system to detect degradation early. The individual techniques differ in the variety of consistency constraints, or types of divergence that they can detect. They range from dependency-focused and source code analysis-based techniques [7] to logical query-based techniques for checking architecturally induced constraints far beyond dependencies [8, 9, 10, 11].

Most approaches have in common that some kind of mapping between architectural units, e.g. modules, and implementation units, such as source code files, is required. Reflexion modelling, for example, exploits this mapping to detect source code dependencies that are not covered by dependencies in an architectural model and which are hence discouraged [7].

In some cases, architectural documentation describing the relationship between architecture and implementation can help create this mapping. More often than not though, architectural documentation is missing or outdated such that the architecture and the mapping towards code need to be recovered from a system's implementation [12]. Performed manually, this constitutes a challenging and labour-intensive task even for system experts. As expressed by professional software architects and designers in a study by Ali et al., creating the mapping is one of the major obstacles to adopting architectural consistency techniques in industrial practice [13].

Researchers have thus put some attention into developing techniques that support software engineers in this task by creating mappings partially automatically or by recommending mappings [14, 15, 16, 17, 18, 19]. Most recently, text classification based on machine learning has been applied to automatically categorize units of source code according to the architectural concern or module they implement and should be mapped to [17, 20].

These approaches show promising results. The question arises though whether the full potential of machine learning for text classification in this context has already been tapped. Several text classification algorithms that perform well in different contexts have not yet been investigated. The question of how to optimally extract and preprocess source code for classification has not yet been exhaustively explored either.

The goal of this article is to shed some light on the performance, i.e. the predictive capability, and other properties of several machine learning-based text classifiers for

ECSA 2021 Companion Volume

florean.alexander@gmail.com (A. Florean); laoa99@outlook.com (L. Jalal); zipani.sinkala@kau.se (Z. T. Sinkala); sebastian.herold@kau.se (S. Herold)

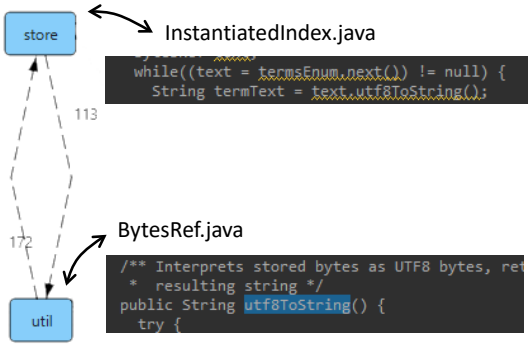

Figure 1: Exemplary cutout of a reflexion modelling and source code dependency contributing to undesired architectural dependency.

the described task. We present a comparative analysis of three classifiers that were applied to map the code of five different systems to their specified architectures.

The contribution of the paper is a set of findings that may guide further research and use of these classifiers for the task of interest. These guidelines, on the one hand, give advice for the selection of an appropriate classifier based on assumptions regarding the alignment between architecture and modular implementation elements like packages. On the other hand, they provide rules of thumb for the recommended size of an initial, manual mapping required to train the classifiers.

The remaining article is organized as follows. The following section describes relevant technical background as well as related work. Section 3 explains the experimental setup of the comparative analysis. In Sec. 4, we summarize the results, which are discussed in Sec. 5. The article is concluded in Sec. 6.

2. Background

2.1. Architecture Consistency Checking

Techniques for checking the consistency between the software architecture of a system and its implementation come in various forms. Several authors provide exhaustive overviews of available approaches and tools [6, 9]. The approaches differ in the way how architectures are represented and the formalism on which the actual checking mechanism relies and, hence, the type of architectural constraints that can be expressed and checked.

Many techniques have in common that they require an association between elements of the architecture and elements of the implementation for many typical consistency constraints. The most fundamental consistency constraints are related to dependencies. The intended architecture of a software system often defines allowed / prohibited, or expected / discouraged dependencies between architectural modules. In order to check whether the dependencies present in source code conform with those, i.e. to compare code with architectural dependencies, the architectural modules to which sources and targets of code dependencies are mapped, need to be known. Fig. 1 shows a cut-out of a so-called reflexion model of one of the systems (Lucene) used in the experiments presented in this paper. It depicts two architecture modules as boxes. Dashed lines (as opposed to solid lines) indicate that dependencies between these modules are architecturally discouraged in either direction; they are, however, present in source code: for example, there is a call, located in file `InstantiatedIndex.java` that is being mapped to store, to a method called `utf8ToString()`, located in file `BytesRef.java`, which is being mapped to util. Through simple code analysis and tracing the mapping, this can be identified as architecturally discouraged dependency between the modules store and util.

Consequently, creating and maintaining such a mapping is crucial for applying architectural consistency checking effectively. As pointed out by Ali et al., the effort needed for manual mapping constitutes a serious concern in industrial practice [13]. Several approaches exist to support software engineers in this time-consuming task. The most relevant ones will be discussed in Sec. 2.3.

2.2. Text Classification with Machine Learning

Text classification is one of the fundamental activities in Natural Language Processing [21]. The goal of text classification is to assign a text written in natural language to one or more predefined categories. Applications of this activity include, for example, sentiment analysis or spam detection.

The central idea of applying machine learning to the task of text classification is to train a classification model based upon text samples for which the assigned categories are already known. Fig. 2 depicts the typical steps in training a text classification model and using it to predict its label/category, i.e. to classify new text. For learning, a set of text documents for which their labels, i.e. categories, are known, is required. These documents are often preprocessed, e.g., to remove stop words or to stem words. A feature extractor transforms the preprocessed documents into a numerical vector. Finally, the classification model is trained according to the machine learning algorithm that is applied. It can then be used to predict the label of a new text that was preprocessed and brought into its numerical representation.

The overall, general procedure can be transferred to the specific context of mapping code to architecture quite easily. The documents to be classified are the aforemen-

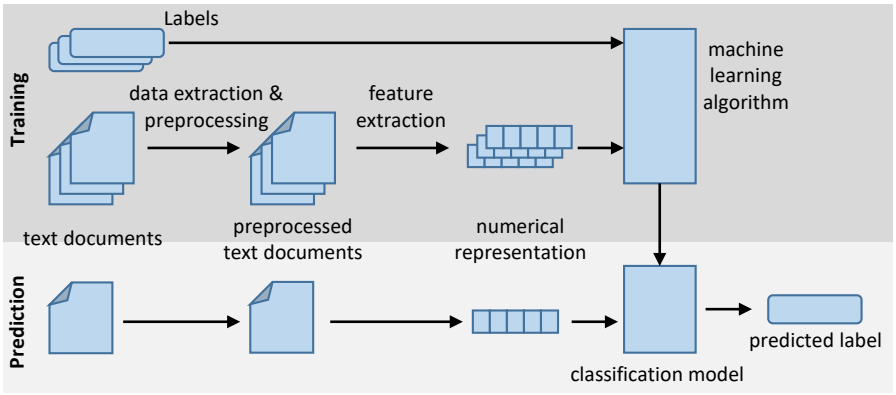

Figure 2: Schematic process of training and using a text classification model through machine learning.

tioned source code entities, like source code files. Architectural modules are represented by labels—for yet unlabelled source code entities, a classification model should propose the correct module. For training, we require a sufficiently large set of source code entities for which their labels—the modules they are mapped to—is known.

2.3. Related Work

The studies by Christl et al. were among the first to investigate techniques for automating the mapping step needed in architecture consistency checking [14, 15]. They developed a technique, called HuGMe, for interactive, human-guided mapping and compared two different attraction functions, CountAttract and MQAttract, measuring how well a code entity will map to an architectural module based on structural properties.

Bittencourt et al. presented a technique based on information retrieval, thus addressing the mapping problem from an textual analysis angle instead [16]. They developed an attraction function based on Latent Semantic Indexing and evaluated it separately as well as a hybrid approach with both CountAttract and MQAttract. The best results were achieved by integrating their novel attraction function and CountAttract.

Both approaches require a set of manually mapped source code entities as a foothold for the applied techniques. Sinkala and Herold instead exploit textual descriptions of the modules of intended architectures to provide their information retrieval-based technique called InMap with initial information for recommending mappings [18, 19].

Olsson et al. developed and analysed a technique based on machine learning [17]. Taking an initial, manually created mapping of a portion of the source code, a Naive Bayes classifier is trained and then used to predict the mapping for the remaining source code entities. The information used for classification is extracted from the compiled source code and consists of package names, file/class names, and attribute and variable identifiers. Compound words, like indicated through camel-casing, are split and the resulting texts are stemmed. The resulting documents are complemented by terms reflecting dependencies. This way structural information can be considered in the classifier without the need to integrate a separate dependency analysis approach. The authors show that this approach outperforms HuGMe significantly; if module descriptions are available, though, InMap performs slightly better [19].

The focus of the work by Link et al. is slightly different yet related [20]. In their approach called RELAX, code entities are not mapped to architectural modules but concerns which are potentially reusable across systems. Any document of a system considered being part of an architectural concern can be fed into the training process of a Naive Bayes classifier to categorize new documents according to their textual content. The approach is compared with two other clustering approaches for architecture recovery as this is the main scenario that the authors target. For five out of eight case study systems, RELAX is shown to perform best in comparison. The study does neither include details of the preprocessing of documents nor a replication package such that technical details of how information is extracted from source code remain unclear.

3. Experimental Design

3.1. Research Questions

The overarching motivating question for this study is how well do different machine-learning based classification models perform in mapping code entities to architec-

Table 1
Descriptive statistics of the subject systems.

System	#files	lines of code	lines of comments	#modules	#files/ module (sd)
Ant	713	86,685	76,987	15	47.5 (65.1)
JabRef	845	88,562	17,187	6	140.7 (161.2)
Lucene	508	60,345	33,342	7	83.0 (64.7)
ProM	867	69,492	22,763	15	123.3 (55.4)
TeamMates	812	102,072	12,514	6	135.3 (119.2)

tural modules. As Sec. 2.3 shows, the focus of related approaches so far has been a single classification algorithm and less a comparison of classifiers or an investigation of their performance properties when applied in the context of interest.

The envisaged scenario for the usage of machine learning technique in this context is that a classifier is first trained with an initial set of manually created mappings based on the textual content of source code files. After that, the trained classification model is used to predict the mappings for the remaining source code files.

We therefore break the main motivating question down into two research questions:

- RQ1: How does the selection of source code elements during preprocessing affect the performance of these classifiers?
- RQ2: How is the performance of different classifiers affected by the size of the training set size, i.e. the number of code entities that need to be mapped manually initially?

For each of the questions, we define a separate experiment based on the same set of systems and classifiers.

3.2. Subject Systems and Classifiers

For training and evaluating text classifiers for the task at hand, a set of systems is required for each of which a) the source code is accessible and b) the mapping between an intended architecture and the source code is known. We explored two data sources for identifying systems that fulfil these prerequisites: the SAEroCon repository[1] and the repository of the s4rdm3x tool [22]. Five open source software systems from these two repositories as listed in Table 1 were selected for this study. They are all written in Java.

Three commonly used machine learning-based classifiers were selected for the study. Naive Bayes for text classification was selected as the most relevant related work is built on it (see Sec. 2.3) and because of its good performance with even little training data [23]. Support Vector Machines (SVM) were selected for the same reason and their good accuracy outperforming Naive Bayes in comparative studies [24]. Logistic Regression as the final classifier has been shown to perform at similar performance levels as SVM and was hence selected for comparison, too [25].

[1] https://github.com/sebastianherold/SAEroConRepo/wiki

3.3. Experiment 1: Comparing Extraction and Preprocessing Variants

The goal of the first experiment is to address RQ1. In a first step, we developed a list of elements in (Java) source code that we believed to potentially carry architecturally relevant information w.r.t. the required mapping. We judged the following elements to be potentially relevant:

- Package declarations: Indicate containment relationships that might match course-grained architectural structures.
- Import declarations: Elements of the same architectural module often share the same dependencies.
- Class declarations: Types defined in the same module might share the same (part of the) domain vocabulary as expressed in their names.
- Public methods: Same rationale as for class declarations.
- Comments: May refer to architectural aspects and decisions, parts of the domain vocabulary, etc., beyond what is being expressed in code

We furthermore identified seven different preprocessing steps that could be activated or deactivated for each of the above elements in a source code file:

1. Splitting of compound words: split, e.g. camelcase notation, `getCustomerId` becomes `get Customer Id`.
2. Stemming: reduce inflected word to their stem, e.g. `notification` or `notify` become `notif`.
3. Transform to lower case
4. Removing single characters
5. Removing stop words, such as `the`, `and` or `of`
6. Removing Java keywords, such as `class` or `public`

7. Tokenization of words: chopping the stream of characters that the document consists of into actual tokens based on separators such as spaces, colons, etc.

A complete investigation of all combinations of preprocessing steps in a fixed order would lead to 2^7 options per extracted source code element. For extracting all of the above elements alone, this would lead to 2^{35} combination which we considered infeasible. Instead, we experimented with several settings in an exploratory pre-study from which we concluded to activate the preprocessing steps 3 to 7 per default for all code elements as deactivating them lead to decreased performance in the explored alternatives.

In the same pre-study two different feature representation techniques were compared, bag-of-words and tf-idf [26, 27]. We noted that bag-of-words outperformed tf-idf on average and hence chose the former for the experiments.

For each combination of subject systems, classifier, and combination of code extraction and active preprocessing steps, we trained and validated ten models following a Monte-Carlo cross-validation scheme [28]. The training set ratio was kept constant at 0.2 and stratified sampling was applied. The latter ensures that the proportion of the classes (i.e., modules) in the overall dataset is kept in both training and testing sets during cross-validation. This ensures that both sets are representative for the overall dataset.

The performance of the models were evaluated in terms of accuracy, i.e. the relative frequency of correct classifications, and averaged over all subject systems.

3.4. Experiment 2: Measuring the Effect of Training Set Sizes

The goal of the second experiment is to address RQ2. Based on the results of the first experiment, one of the best performing combinations of extraction and preprocessing settings was selected for each classification algorithm. The code files for each system were extracted and preprocessed accordingly and represented as bags-of-words.

We then trained models for each of the three classification algorithms at different training set sizes expressed as fractions of the overall datasets, i.e. relative number of available mappings between source code files and architectural modules. Per combination of system, training set size of interest, and classifier, we trained and evaluated according to a Monte Carlo cross-validation with 100 splits and stratified sampling.

In order to evaluate the resulting models, we computed several precision and recall averages per system, training set size, and classifier. The average precision/recall is defined as the precision/recall per class (module) divided by the number of classes. The weighted average precision/recall takes the proportions of classes into account and weights the individual precision/recall scores accordingly. The weighted average recall is equal to the accuracy of a classification model[2].

Practically speaking, this experiment corresponds roughly to a situation in which a software architect/designer can estimate the number of code entities that should be mapped to each of the architectural modules. The experiment could offer advice regarding the relative number of entities she should map per module in order to get a sufficiently accurate automated mapping for the rest of the system.

This scenario, however, is not always realistic as module sizes may be unknown or estimations may be wrong. For that reason, we repeated the experiment described above with different absolute training set sizes, expressed as absolute number of files per modules that should enter the training set. Obviously, this way of sampling is not stratified; the number of splits and metrics for evaluation remain the same as for comparing based upon relative training set sizes.

3.5. Replication Package

The replication package, including the scripts for preprocessing the data, training and evaluating the classifiers is available at https://github.com/sebastianherold/ml-for-architecture-mapping.

4. Results

As described in Sec. 3.3, we explored the accuracy of all classification algorithms for different data extraction and preprocessing settings in the first experiment. Fig. 3 summarises the findings per combination of extracted source code elements. All three classification algorithms scored best when the data extracted from the code files was limited to package declarations and class declarations. Logistic regression and SVM achieved accuracies of 0.93 each, outperforming Naive Bayes by 0.07. SVM's and Logistic regression's accuracy drop significantly to 0.68 and 0.73 at maximum, respectively, when package declarations are not included in the data. Naive Bayes drops to 0.75 at extracting everything else but package declarations, performing more accurately than SVM and Logistic Regression in this scenario.

[2]The recall of each individual class c_i is weighted by $\frac{|c_i|}{n}$ with n being the total number of data points. Since $|c_i| = TP_{c_i} + FN_{c_i}$, each term for the weighted average recall turns into TP_{c_i}/n which summed up over all classes is equivalent to the definition of accuracy.

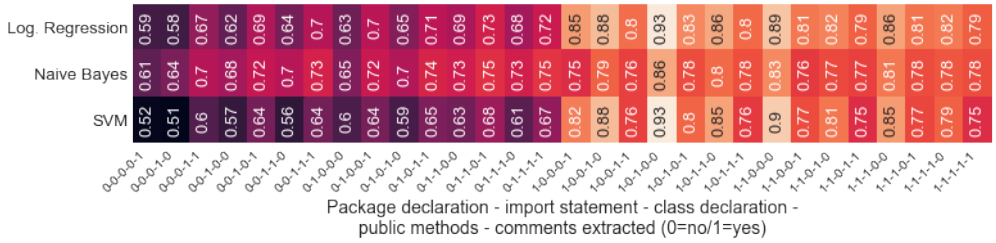

Figure 3: Accuracy for extraction of different parts of source code. Values are averages over all investigated combinations of preprocessing steps for the extracted parts.

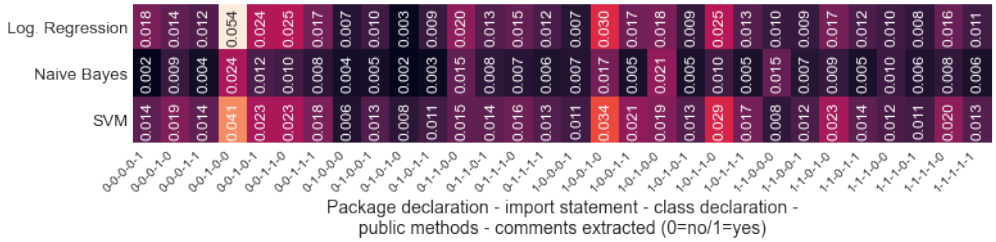

Figure 4: Standard deviation of accuracy averages for each combination of extracted source code parts as depicted in Fig. 3.

The role of comments also changed with the inclusion of package declarations. With package declarations included, adding comments, while keeping inclusion/exclusion of the other code elements unchanged, seem to rather decrease the accuracy of the classifiers. Without package declarations, including comments lead to accuracy improvements of up to 0.07.

It should be noted that the mapping onto modules aligned quite well with the package structure in all five systems, which might explain the impact of including the package declaration. We excluded the results for only extracting package declarations as we believe that the very good scores (beyond 0.98) of those models were overfitting and heavily biased towards the selected systems.

Fig. 4 illustrates the standard deviation for each extraction setting and classifier. The standard deviation is below 0.02 in 80% of the cases, exceeding 0.05 slightly in only one case. These results show that the variable preprocessing settings, stemming of words and splitting of compound words, affect accuracy only slightly.

The results related to classification performance over training set size as relative fraction of the overall number of source code files are visualised in Fig. 5. They confirm that SVM and Logistic Regression perform better than Naive Bayes in accuracy, precision and recall in almost all settings[3]. The improvement in accuracy decreased for all systems and classifiers beyond a relative training size of 0.15, in particular for JabRef, ProM, and Teammates. The curves show similar behaviour for the weighted average precision. Unweighted averages keep a steeper slope in comparison even beyond training set sizes of 0.15 which shows that the performance for smaller modules benefits from increasing the training set size.

In Fig. 6, the results of evaluating classification performance for different numbers of files per module in the training set are shown. Most curves across all metrics show a sharp increase in performance that slows down at 10 files per module. This is less pronounced, sometimes hardly visible, for Naive Bayes as compared to SVM and Logistic Regression. The results for the weighted averages seem more similar to their unweighted counterparts in this experiment. SVM and Logistic Regression outperform Naive Bayes in almost all settings and systems in this experiment, too.

5. Discussion

5.1. Findings regarding RQ1

In the following we discuss and summarize the finding related to the question how different ways of data extraction and preprocessing affect the performance of the tested classifiers.

The experiment results show that package declarations constitute a significant piece of information for the tested classifiers. A decrease of 0.3 in accuracy for settings

[3]These experiments were performed extracting package declarations and class declarations.

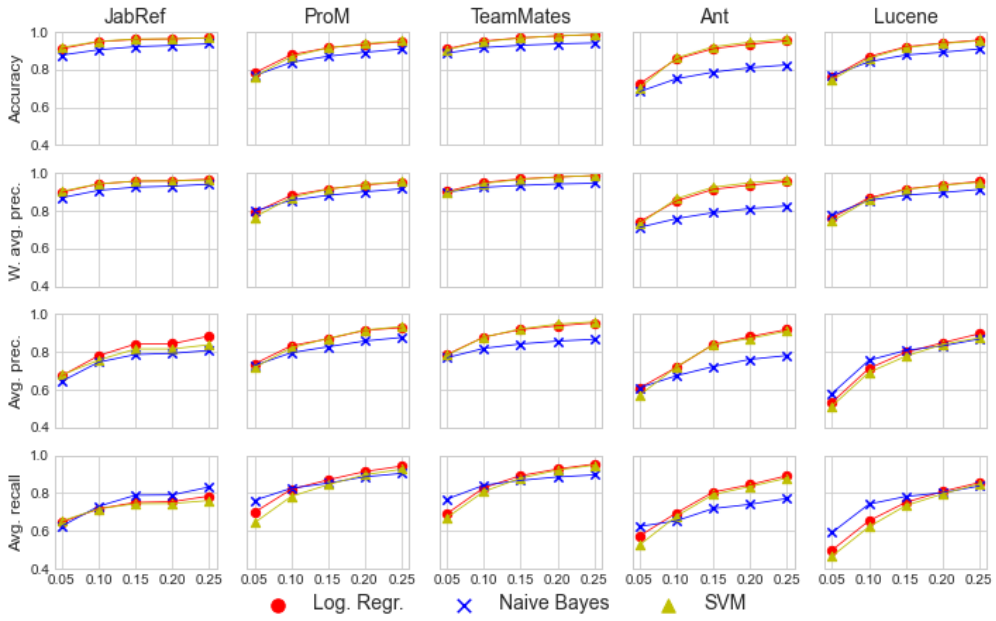

Figure 5: Performance metrics of classifiers over relative training set size.

in which the only difference is to not consider package declarations is common across the results.

This seems quite natural as the mappings for the system used for training are largely aggregating source code elements along several subtrees of the package hierarchy instead of individual classes from unrelated packages. Only in the mapping of JabRef exist cases of packages whose contained classes/interfaces, i.e. and corresponding files, are mapped to different modules, and which these different mappings do not align with the subpackage/subdirectory structure. It does hence not surprise that settings including package declarations and only few other pieces of information score best. In our experiments, class declarations seem to complement package declarations best. Since Naive Bayes does not perform as well as the other classification algorithms, we formulate our first finding as

Finding 1. *In settings, in which the architectural module structure can be assumed to align well with macro-structures declared in the system's implementation, these declarations and type information should be extracted. SVM and Logistic Regression provide more accurate results than Naive Bayes.*

Note that a straight-forward alignment does not necessarily imply that a mapping can easily be constructed manually without the need for automation in the first place. In large-scale systems, structures of hundreds of packages are not uncommon. If architectural modules are well-aligned but mapped to more than one package in such systems, identifying the relevant packages for a module can still be tedious.

An interesting question in the light of the first finding is whether the approaches by Olsson et al. and Link et al. could benefit from using a different classifier than Naive Bayes [17, 20]. While the alignment with source code structures is largely unclear for Link et al., Olsson et al. applied their approach to the same, well-aligned systems used in this study. This finding also suggests that their approach could be further tuned to only use package and type information as compared to including also variable identifiers. In use cases, in which the slightly slower training of SVM and Logistic Regression is an issue, Naive Bayes might be the better alternative.

It is common that the mappings are not that well-aligned and straight-forward [29]. Furthermore, some programming languages do not declare any containment relationships equivalent to packages declarations. The tested systems do not represent this scenario properly. We therefore looked at the performance of the classifiers without considering package declarations as approximation of their behaviour if we did not have that information or considered it useless. In this setting, Naive Bayes exploiting import declarations, class declarations, and comments, showed the best accuracy (on a par with additionally including declarations of public methods).

Finding 2. *If alignment with any macro-structures de-*

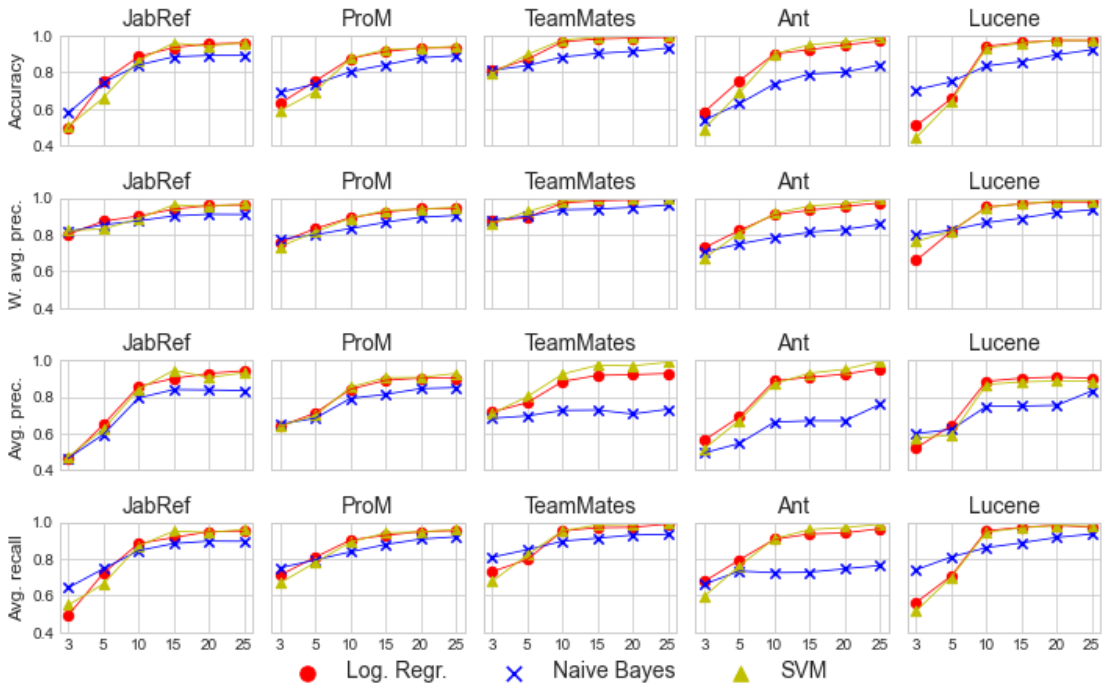

Figure 6: Performance metrics of classifiers over absolute training set size.

clared or derived from source cannot or should not be assumed, Naive Bayes trained based on declarations of types, imports, and comments should be used.

The standard deviation within groups of identical extractions regarding different preprocessing settings is very low. This indicates that the impact of stemming and splitting of compound words does not have a significant impact on the resulting accuracy of any of the tested classification algorithms.

Finding 3. *The selection of parts to be extracted for classifier training and mapping prediction appears to be more important than the selection of the preprocessing steps considered optional in this study.*

Further investigation may be necessary to investigate the potentially larger impact of other preprocessing steps in the individual scenarios described above.

5.2. Findings regarding RQ2

In this subsection, we summarize the finding related to the question of how the training set size, corresponding to the number of initially, manually mapped files, affects classifier performance.

The results suggest that in many cases the additional gain in accuracy, precision, and recall slackens at around 15% of the overall dataset (equal to the total number of source code files) and above. Enhancing the initial mapping beyond this point may therefore turn out infeasible. Even in the relatively small sample systems of this study like JabRef, increasing this mapping by 5% of the overall number of code files means to map more than 40 additional files. This may possibly not pay off, in particular for larger systems, if the gain in classification performance is minimal. We therefore state:

Finding 4. *If the number of files supposed to be mapped to each module can be estimated, mapping around 15% of that number in the initial mapping may be a good rule-of-thumb for training an efficient classifier.*

The results of experimenting with absolute training set size complement these result for scenarios in which it is not possible or desirable to estimate the number of files mapped to each module. For the tested systems, the gain in accuracy, precision, and recall flattens out at 10 files per module in the initial mapping which leads us to our final finding:

Finding 5. *An initial mapping of at least 10 files per module may lead to a satisfactorily performing classifier.*

Again these findings can only properly compared to Olsson et al. as Link et al. do no report details about training

set sizes [17, 20]. The results reported by Olsson et al., suggest that an initial mapping of ca. 20% lead to satisfying performance. Those results, however, are measured as average over even imbalanced initial mappings that do not take proportions of modules into account. We therefore think that our findings recommending a slightly smaller relative size of the, however, stratified mapping is in line with those results.

5.3. Validity

Several factors limit the external validity of this study. Firstly, although the subject systems are anything but trivial, they certainly do not represent large-scale software systems. Further research in particular to confirm or refine the findings regarding training set sizes is required. Moreover, the results may not accurately reflect the behaviour of classifiers if package or equivalent declarations are considered but the architecture does not align with them. A further threat to external validity is the scoping to systems written in Java. This is due to the limited availability of systems for which the architecture as well as the source code is available. The systems identified appeared all to be Java-based. Moreover, we did not tune the hyperparameters but only touched upon this non-exhaustively in the before-mentioned exploratory pre-study. This might be considered a limitation as well as a threat to external validity as the results might differ for classification models with different hyperparameters.

The experiments aim at identifying causal relationships between independent variables (extraction/preprocessing settings and training set sizes, respectively) and dependent variables (performance measures). We are pretty confident that the internal validity is high as all other identified parameters were kept constant throughout the experiments. A potential threat are of course bugs in the scripts and software used to extract and preprocess data as well as for training and evaluating the classifiers. We consider this risk to be low though as established software libraries were used for this purpose and any self-written code (largely produced by the first and second author) was carefully reviewed by the third and fourth author.

We consider the selected method for cross-validation as main threat to construct validity. Inappropriate train/test splits may lead to biased classification models that might not reflect a classifier's performance properly. We believe though that the chosen repetitions for the cross-validation in the experiments was sufficient to mitigate this risk.

6. Conclusion

The results of the presented study indicate that there is no silver bullet classifier. The choice of an optimal classifier and elements to be extracted from source code is influenced by system characteristics like the alignment of macro-structural elements with the assumed architecture. To identify more of such distinguishing characteristics or scenarios seems to be an interesting objective of future research. It will be particularly relevant to investigate whether the recommendations regarding the size of initial mappings hold in practice and if they apply for larger systems, too.

Last but not least more classifiers wait to be tested for their ability to automate code-to-architecture mapping. For these as well as for those tested in this study, different preprocessing techniques should be investigated more deeply and the improvement that hyperparameter tuning might achieve should be explored. Such a more exhaustive comparative study might also need to take the performance and the resource demands of the training process into account.

References

[1] D. E. Perry, A. L. Wolf, Foundations for the study of software architecture, SIGSOFT Softw. Eng. Notes 17 (1992) 40–52. doi:10.1145/141874.141884.

[2] M. W. Godfrey, E. H. S. Lee, Secrets from the monster: Extracting mozilla's software architecture, in: In Proc. of 2000 Intl. Symposium on Constructing software engineering tools (CoSET 2000, 2000, pp. 15–23.

[3] C. Deiters, P. Dohrmann, S. Herold, A. Rausch, Rule-based architectural compliance checks for enterprise architecture management, in: 2009 IEEE International Enterprise Distributed Object Computing Conference, 2009, pp. 183–192. doi:10.1109/EDOC.2009.15.

[4] J. van Gurp, J. Bosch, Design erosion: problems and causes, Journal of Systems and Software 61 (2002) 105–119. doi:10.1016/S0164-1212(01)00152-2.

[5] S. Sarkar, S. Ramachandran, G. S. Kumar, M. K. Iyengar, K. Rangarajan, S. Sivagnanam, Modularization of a large-scale business application: A case study, IEEE Software 26 (2009) 28–35. doi:10.1109/MS.2009.42.

[6] L. Passos, R. Terra, M. T. Valente, R. Diniz, N. Mendonça, Static architecture-conformance checking: An illustrative overview, IEEE Software 27 (2010) 82–89. doi:10.1109/MS.2009.117.

[7] G. Murphy, D. Notkin, K. Sullivan, Software reflexion models: bridging the gap between design and

implementation, IEEE Transactions on Software Engineering 27 (2001) 364–380. doi:10.1109/32.917525.

[8] O. de Moor, D. Sereni, M. Verbaere, E. Hajiyev, P. Avgustinov, T. Ekman, N. Ongkingco, J. Tibble, .QL: Object-Oriented Queries Made Easy, Springer Berlin Heidelberg, 2008, pp. 78–133. doi:10.1007/978-3-540-88643-3_3.

[9] S. Herold, Architectural compliance in component-based systems, Ph.D. thesis, Clausthal University of Technology, 2011.

[10] S. Herold, A. Rausch, Complementing model-driven development for the detection of software architecture erosion, in: Proceedings of the 5th International Workshop on Modeling in Software Engineering, MiSE '13, IEEE Press, 2013, p. 24–30.

[11] S. Schröder, G. Buchgeher, Formalizing architectural rules with ontologies - an industrial evaluation, in: 2019 26th Asia-Pacific Software Engineering Conference (APSEC), 2019, pp. 55–62. doi:10.1109/APSEC48747.2019.00017.

[12] W. Ding, P. Liang, A. Tang, H. Van Vliet, M. Shahin, How do open source communities document software architecture: An exploratory survey, in: 2014 19th International Conference on Engineering of Complex Computer Systems, 2014, pp. 136–145. doi:10.1109/ICECCS.2014.26.

[13] N. Ali, S. Baker, R. O'Crowley, S. Herold, J. Buckley, Architecture consistency: State of the practice, challenges and requirements, Emp. Softw. Eng. 23 (2018) 224–258. doi:10.1007/s10664-017-9515-3.

[14] A. Christl, R. Koschke, M.-A. Storey, Equipping the reflexion method with automated clustering, in: 12th Working Conference on Reverse Engineering (WCRE'05), 2005, pp. 10 pp.–98. doi:10.1109/WCRE.2005.17.

[15] A. Christl, R. Koschke, M.-A. Storey, Automated clustering to support the reflexion method, Information and Software Technology 49 (2007) 255–274. URL: https://www.sciencedirect.com/science/article/pii/S095058490600187X. doi:https://doi.org/10.1016/j.infsof.2006.10.015, 12th Working Conference on Reverse Engineering.

[16] R. A. Bittencourt, G. J. d. Santos, D. D. S. Guerrero, G. C. Murphy, Improving automated mapping in reflexion models using information retrieval techniques, in: 2010 17th Working Conference on Reverse Engineering, 2010, pp. 163–172. doi:10.1109/WCRE.2010.26.

[17] T. Olsson, M. Ericsson, A. Wingkvist, Semi-automatic mapping of source code using naive bayes, in: Proceedings of the 13th European Conference on Software Architecture - Volume 2, ECSA '19, Association for Computing Machinery, New York, NY, USA, 2019, p. 209–216. doi:10.1145/3344948.3344984.

[18] Z. T. Sinkala, S. Herold, InMap: Automated interactive code-to-architecture mapping, in: Proceedings of the 36th Annual ACM Symposium on Applied Computing, SAC '21, Association for Computing Machinery, New York, NY, USA, 2021, p. 1439–1442. doi:10.1145/3412841.3442124.

[19] Z. T. Sinkala, S. Herold, InMap: Automated interactive code-to-architecture mapping recommendations, in: 2021 IEEE 18th International Conference on Software Architecture (ICSA), 2021. doi:10.1109/ICSA51549.2021.00024.

[20] D. Link, P. Behnamghader, R. Moazeni, B. Boehm, Recover and relax: Concern-oriented software architecture recovery for systems development and maintenance, in: Proceedings of the International Conference on Software and System Processes, ICSSP '19, IEEE Press, 2019, p. 64–73. doi:10.1109/ICSSP.2019.00018.

[21] C. C. Aggarwal, C. Zhai, A survey of text classification algorithms, in: Mining text data, Springer, 2012, pp. 163–222.

[22] T. Olsson, M. Ericsson, A. Wingkvist, s4rdm3x: A tool suite to explore code to architecture mapping techniques, Journal of Open Source Software 6 (2021) 2791. doi:10.21105/joss.02791.

[23] I. H. Witten, E. Frank, M. A. Hall, Data Mining: Practical Machine Learning Tools and Techniques, 3 ed., Morgan Kaufmann, Amsterdam, 2011.

[24] A. Sheshasaayee, G. Thailambal, Comparison of classification algorithms in text mining, International Journal of Pure and Applied Mathematics 116 (2017) 425–433.

[25] K. Shah, H. Patel, D. Sanghvi, M. Shah, A comparative analysis of logistic regression, random forest and knn models for the text classification, Augmented Human Research 5 (2020) 1–16.

[26] Z. S. Harris, Distributional structure, WORD 10 (1954) 146–162. doi:10.1080/00437956.1954.11659520.

[27] K. Sparck Jones, A statistical interpretation of term specificity and its application in retrieval, Journal of documentation 28 (1972). doi:10.1108/eb026526.

[28] R. R. Picard, R. D. Cook, Cross-validation of regression models, Journal of the American Statistical Association 79 (1984) 575–583. doi:10.1080/01621459.1984.10478083.

[29] J. Buckley, N. Ali, M. English, J. Rosik, S. Herold, Real-time reflexion modelling in architecture reconciliation: A multi case study, Information and Software Technology 61 (2015) 107–123. URL: https://www.sciencedirect.com/science/article/pii/S0950584915000270. doi:https://doi.org/10.1016/j.infsof.2015.01.011.