Using Automatically Recommended Seed Mappings for Machine Learning-Based Code-to-Architecture Mappers

Sebastian Herold
Department of Mathematics and Computer Science
Karlstad University
Karlstad, Sweden
sebastian.herold@kau.se

Zipani Tom Sinkala
Department of Mathematics and Computer Science
Karlstad University
Karlstad, Sweden
tom.sinkala@kau.se

ABSTRACT

Software architecture consistency checking (SACC) is a popular method to detect architecture degradation. Most SACC techniques require software engineers to manually map a subset of entities of a system's implementation onto elements of its intended software architecture. Manually creating such a "seed mapping" for complex systems is a time-consuming activity.

The objective of this paper is to investigate if creating seed mappings semi-automatically based on mapping recommendations for training automatic, machine learning-based mappers can reduce the effort for this task.

To this end, we applied InMap, a highly accurate, interactive code-to-architecture mapping approach, to create seed mappings for five open source system with known architectures and mappings. Three different machine learning-based mappers were trained with these seed mappings and analysed regarding their predictive performance. We then compared the manual effort involved in using the combination of InMap and the most accurate automatic mapper and the manual effort of mapping the systems solely with InMap.

The results suggest that InMap, with a minor adaption, can be used to seed an accurate mapper based on Naive Bayes. A full mapping with only InMap though turns out to involve slightly less manual effort on average; this is, however, not consistent across all systems. These results give reason to assume that more advanced ways of combining automatic mappers with InMap may further reduce that effort.

CCS CONCEPTS

• **Software and its engineering** → *Software reverse engineering*; *Software architectures*; *Software evolution*;

KEYWORDS

software architecture consistency, code-to-architecture mapping, software architecture degradation, machine learning

ACM Reference Format:
Sebastian Herold and Zipani Tom Sinkala. 2023. Using Automatically Recommended Seed Mappings for Machine Learning-Based Code-to-Architecture Mappers. In *The 38th ACM/SIGAPP Symposium on Applied Computing (SAC '23), March 27-March 31, 2023, Tallinn, Estonia.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3555776.3577628

SAC '23, March 27-March 31, 2023, Tallinn, Estonia

ACM ISBN 978-1-4503-9517-5/23/03.
https://doi.org/10.1145/3555776.3577628

1 INTRODUCTION

The implementations of software systems tend to diverge from the intended architecture over time. This effect is known as software architecture degradation [18]. It can have dire consequences such as a continuous decay of the affected system's maintainability and a decreased ability of the system to meet other desired quality properties. This, in turn, may lead to extensive and costly re-engineering efforts or expensive re-developments of systems [6, 9, 19, 25].

Software architecture consistency checking (SACC) is one approach to mitigate software architecture degradation [17]. SACC techniques promote frequent checks for inconsistencies between the intended software architecture of a system and its current implementation. Architecture degradation can be detected early this way and software engineers can take actions against it in time.

Most popular and commercially successful SACC techniques, such as reflexion modelling or related techniques, require some sort of mapping between entities of the source code, such as code files or classes, and elements of the intended architectures, such as modules [10, 17]. In reflexion modelling, for example, code entities are mapped to architectural modules in order to be able to automatically analyse source code dependencies and compare them with dependency constraints stated by the architecture [12]. Creating and maintaining this mapping are complex and time-consuming tasks because they require a deep understanding of the system at hand. This is particularly true in scenarios in which a complex system has to be initially and fully mapped to enable SACC in the first place. Practitioners indeed expressed in an empirical study by Ali et al. that the efforts associated with code-to-architecture mapping are an obstacle to adoption of SACC in practice [1].

Several approaches exist to (semi-)automate this mapping activity and to hence reduce the manual effort required to perform it [3–5, 14]. Many of the recent and best performing approaches utilize machine learning (ML) [8, 14]. Identifying the "correct" module for a source code entity can be understood as a multiclass classification problem that those approaches treat as a supervised learning task. Based on an initial seed mapping of a fraction of source code entities that serve as the training set, a classifier is trained that attempts to predict the architectural module that the remaining source code entities should be mapped to.

However, even creating this initial seed mapping can be a time-consuming activity. Previous results suggest that this mapping should contain at least approximately 15% of the overall number of source code entities to result in a satisfactorily accurate mapper [8,

14]. For large systems with thousands of entities, this still requires a significant effort. Research also shows that there is a huge variation in precision and recall depending on which entities are used as a seed, and that the question of what constitutes a good seed mapping is not easy to answer universally [14].

In this paper, we investigate the use of InMap for the creation of seed mappings for ML-based mappers. InMap is an highly accurate, interactive approach that does not need an initial seed mapping - at the price that user interaction is required to refine the mapping incrementally [22, 23]. A hybrid approach, consisting of InMap creating the seed mapping and a ML-based mapper with high precision and recall could reduce the effort for mapping a complete system significantly.

To this end, we conducted an experiment using InMap in two different ways to create seed mappings of different sizes for five open-source systems. Three different types of classifiers were trained based on these seed mappings. The classifiers' predictive performance was compared to base line classifiers trained on random seed mappings. The efficiency, in terms of manual effort involved, of combining InMap with these classifiers was compared with the effort using InMap for mapping the systems completely.

The remaining paper is structured as follows. Section 2 covers background and related work. In Sec. 3, we will explain the experimental setup. The results of the experiments are presented in Sec. 4 and discussed in Sec. 5. The paper is concluded in Sec. 6.

2 BACKGROUND

In this section, we briefly introduce the foundations of InMap, which is the approach to create seed mappings in the conducted experiment, and provide an overview of related approaches.

2.1 InMap

InMap is an interactive approach to code-to-architecture mapping based on information retrieval. The targeted type of architectural models corresponds to the concept of "high-level" models in reflexion modelling [12]. These models consist of architectural modules and intended, or allowed, dependencies between the modules. Source code entities, such as source code files, are mapped to architectural modules. The main property of InMap that distinguishes it from other techniques to automate this mapping is that it does not need a manually created seed mapping. The central idea achieving this property is to exploit module descriptions, i.e., descriptions of a module's purpose, which can often be found in architectural documentation or retrieved during architecture recovery sessions. This description, the name of a module, and textual information extracted from the set of source code files already mapped to the module (which is empty in the beginning) are transformed into a search query, which is executed for the set of unmapped source code files (corresponding to a corpus of documents in information retrieval terminology). The query execution results in scores for each unmapped file indicating how relevant the file is w.r.t. the query. The assumption is that the higher the score for a file, the more likely a mapping to the module represented by the query might be. InMap computes this score for each combination of a file and architectural module, extracts the highest scoring combination for each file, and presents the best scoring combinations as recommendations in descending order. The user can then accept or reject the presented recommendations. This feedback is then used by InMap to refine the queries, re-compute the scores and provide new recommendations. In an evaluation with six open-source system, InMap showed a precision of 0.82 and a recall of 0.97 on average and outperformed other mapping techniques requiring seed mappings [23].

2.2 Related Work

Christl et al. proposed an interactive approach to code-to-architecture mapping based on two different attraction functions that aim at determining how likely it is that a source code unit "belongs" to an architectural module based on dependencies [4, 5]. Building on the same overall interactive approach, Bittencourt et al. presented new attraction functions based on textual information and information retrieval concepts instead. They noticed that a combination of dependency-oriented and textual attraction functions worked best in general [3].

Florean et al. compared three ML-based classifiers for the use in code-to-architecture mapping with different ways of preprocessing code and evaluated their performances for different sizes of seed mappings [8]. The results suggested that Naive Bayes is inferior to Logistic Regression and SVM for this task, and that seed mappings consisting of at least of ten files per module, or seed mappings of at least 15% of the total source code files, are required for satisfactorily performing classifiers.

Olsson et al. presented an approach based on textual classification with Naive Bayes [14]. In their technique, information about dependencies in source code and architecture are encoded in the textual representation of source code entities such that both type of information, dependencies-focused and textual, can be considered by a single technique. This eliminates the need for an aggregation or weighting of separate models that might be hard to generalize across systems and use cases. In a comparative evaluation, this approach showed significantly better performance than the previously described approaches.

While this approach was outperformed by InMap in its original form, Olsson et al. showed that using module key words, describing the purpose of an architectural module in a precise way, could be used to create a seed mapping that showed increase precision and recall, even slightly outperforming InMap [16]. The same authors also investigated the question which properties of source code files might indicate that files should be included in the seed mapping in order to make the subsequent automatic mapping as accurate as possible [16]. They found in a single case study that the inclusion of classes exhibiting a high dependency fan-out in the seed mapping led to better automatic mapping results in general [13].

The purpose of the RELAX approach presented by Link et al. is to support architectural recovery [11]. Any textual document contributing to, or documenting, a software system is used to train a classifier that can classify implementation entities according to the architectural concern they contribute to. In contrast to the previous listed approaches, in which training of classifiers is based on system-specific data, an essential part of the motivation of RELAX is to resuse classifiers for common architectural concerns across systems.

Table 1: The subject systems.

System	#files	LOC	#modules
Ant	738	86,685	15
ArgoUML	763	111,626	17
JabRef	852	88,562	6
Jittac	117	8,012	9
TeamMates	321	102,072	6

In a comparison with two other clustering approaches, RELAX performed best for the majority of the evaluated systems.

3 EXPERIMENTAL DESIGN

In this section, we describe the motivating research questions and the protocol of the experiment aiming to address them.

3.1 Research Questions

The research questions for this study are defined as follows:

- RQ 1: How well do ML-based mappers performed if they are trained with automatically recommended seed mappings?
- RQ 2: How efficient are ML-based mappers trained that way in comparison to mapping the full system based on automatically recommended mappings?

The focus of the first research question is whether or automatically recommended mappings can be used to train accurate classifiers for mapping the remaining entities of a systems's implementation. An important assumption we make in this context is that accepting or rejecting mapping recommendations is less demanding and less time-consuming than manually figuring out correct mappings. The second question focuses on investigating whether the manual effort in using those automatic mappers is lower than the manual effort needed to check and accept/reject automatic recommendations.

3.2 Subject Systems and Classifiers

For this study, we needed access to software systems with the following requirements

- The source code of the system is accessible.
- A model of the intended architecture, reflecting the system's modules and having been validated by expert of the systems, exists.
- Short descriptions of the modules' purposes exist.

We picked five open-source systems meeting these criteria from the replication package of a previous study by Florean et al. [8]. The original sources for the mappings for these systems can either be found in the repository of the SAEroCon workshop series[1] or the repository of the s4rdm3x tool [15]. Table 1 lists these systems.

Similar to the study mentioned above, we focused on three textual classifiers, namely Naive Bayes (NB), Support Vector Machines (SVM), and Logistic Regression (LR) due to their ability to work with small amount of data (in particular Naive Bayes) and their performance in comparative studies [20, 21].

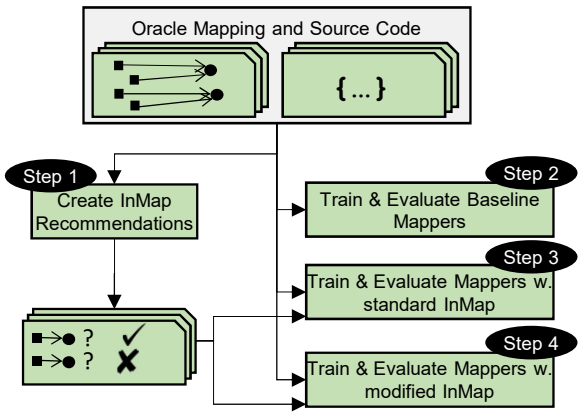

Figure 1: Visualisation of the experiment protocol.

3.3 Experiment Protocol

We chose InMap as the approach for recommending seed mappings as InMap itself does not need seed mappings to operate (see. Sec. 2). As **Step 1** in the experiment (see Fig. 1), we run InMap for all five systems. The tool as provided in the replication package of the corresponding paper operates in an experimental mode in which the recommendations computed by InMap are compared with an oracle mapping such that manual interactions are not required [23]. All mapping recommendations, complemented by information about their correctness w.r.t the oracle mapping, are exported as a list in the order they are provided.

In **Step 2**, we train and evaluate three classifiers following the methodology by Florean et al [8]. For all systems and classifiers, we train and validate 100 models following a Monte-Carlo cross validation scheme with stratification for different training set (i.e., seed mapping) sizes provided as fraction of the total number of source code entities. We measure precision and recall and take the average of these metrics across the 100 trained models to have a baseline value for the subsequent steps. In order to account for the varying sizes of classes in this classification problem, i.e., the number of source code units mapped to modules, we take the weighted average variant of both metrics. These variants weigh each class' precision and recall according to its relative size compared to the overall data points and average the resulting value across all classes. The weighted average recall is equivalent to the accuracy of a classifier [2]. For simplicity, we refer to the metrics simply as precision and recall in the remaining text.

In **Step 3**, for each relative seed mapping size r selected in step 2, we pick the first $n = \lceil e \cdot r \rceil$ correct recommendations from the InMap result list for each system (e being the total number of implementation entities, i.e., source code files, in the system). The precision of InMap in creating this seed mapping, i.e., n divided

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

[2] The accuracy of predictions in a multiclass classification problem is defined as $\sum_{c \in C} \frac{TP_c}{n}$ with C being the set of all classes, TP_i being the number of instances correctly classified in class c, and n being the total number of predictions. The weighted average recall sums up the recalls per class weighted by their relative size: $\sum_{c \in C} \frac{TP_c}{TP_c+FN_c} \cdot \frac{TP_c+FN_c}{n}$ with FN_c being the number of instances of class c incorrectly classified. A reduction of the fractions results in the expression for accuracy.

Table 2: InMap results for mapping all systems completely (#REC=number of recommendations made, #CREC=number of correct recommendations).

System	#Files	#REC	#CREC	Precision	Recall
Ant	738	1037	722	0.70	0.98
ArgoUML	763	1010	758	0.75	0.99
Jabref	852	882	837	0.95	0.98
Jittac	117	137	109	0.80	0.93
TeamMates	321	272	259	0.95	0.81

by the total number of recommendations needed to get those n correct recommendations, is computed as measurement of the effort for creating the seed mapping. For each of the systems, each classifier is trained on the set of mappings recommended in these n recommendations. The performance is again measured in terms of precision and recall.

Step 4 is a repetition of the previous step with a modification of the way the seed mapping is created. We noticed before the actual experiment execution that InMap shows tendencies to favour some of the modules in a system when making recommendations. This means that certain modules tend to be mapped much quicker than others. The seed mapping could therefore be very imbalanced and might only cover a certain subset of modules, which, in turn, could affect the predictive performance of the trained classifiers. Instead of picking the n best ranked recommendations, we therefore pick $\lceil \frac{e \cdot r}{m} \rceil$ correct mappings[3] for each module, m being the number of modules in the intended architecture of the system. From a user perspective, this corresponds to having a list of mapping recommendations per module to select from, instead of a single system-wide list as in the original InMap approach. The validation of the resulting classifiers trained with these seed mappings follows the same procedures as in Step 3.

3.4 Replication Package

The scripts to run the experiments described above and to visualise the results can be accessed at https://github.com/sebastianherold/inmap-seed-mapping.

4 RESULTS

The results of mapping the five systems through InMap alone (Step 1) are shown in Tab. 2. As (source code) files are the units, for which InMap generates recommendations, the number of files indicates the minimal number of recommendations that the user would have to inspect in case InMap finds a recommendation for each file. InMap achieves an average precision of 0.83 and and average recall of 0.94 whereby precision varies among systems more than recall. Translated to a hypothetical system with 1,000 files, for which InMap shows exactly this average performance, the results suggest that for 940 files the correct mappings would be identified after having presented and evaluated $940 * 1/0.83 = 1,133$ recommendations.

Table 3 shows the results for creating random seed mappings of different sizes between $5 - 25\%$ of the total number of files in the five software systems (Step 2, see Sec. 3. As would be expected, precision and recall generally increase with seed mapping size as the seed mapping corresponds to the training set; increasing the volume of data available for training a classifier can generally be expected to improve the classifier's predictive performance. LR and SVM perform on par in terms of precision and recall while NB show slightly lower precision and recall in four systems, and significantly lower values for Ant. The average precision values at a relative seed mapping size of 0.25 are 0.89 for LR, 0.84 for NB, and 0.88 for SVM; average recall is 0.88 for LR, 0.83 for NB, and 0.87 for SVM. NB performs slightly better than the other classifiers for the smallest relative seed set size of 0.05.

[3] or as many as possible if the module is smaller

Table 3: Baseline values for precision and recall for all classifiers at different relative seed mapping sizes (100 split Monte Carlo cross-validation).

		Weighted avg. precision					Weighted avg. recall				
System	Class.	0.05	0.10	0.15	0.20	0.25	0.05	0.10	0.15	0.20	0.25
Ant	LR.	0.62	0.74	0.79	0.83	0.85	0.57	0.70	0.77	0.81	0.83
	NB	0.60	0.65	0.67	0.69	0.70	0.58	0.62	0.65	0.67	0.69
	SVM	0.61	0.75	0.81	0.84	0.87	0.56	0.71	0.78	0.82	0.85
ArgoUML	LR	0.70	0.79	0.84	0.86	0.88	0.67	0.77	0.82	0.85	0.87
	NB	0.75	0.81	0.83	0.84	0.85	0.74	0.79	0.82	0.83	0.85
	SVM	0.66	0.76	0.82	0.84	0.87	0.64	0.74	0.80	0.83	0.86
Jabref	LR	0.86	0.91	0.92	0.94	0.94	0.84	0.90	0.92	0.93	0.94
	NB	0.84	0.87	0.89	0.90	0.90	0.83	0.87	0.88	0.89	0.89
	SVM	0.85	0.92	0.93	0.94	0.95	0.84	0.91	0.93	0.94	0.95
Jittac	LR	0.48	0.70	0.78	0.85	0.88	0.46	0.62	0.72	0.81	0.86
	NB	0.66	0.84	0.87	0.89	0.89	0.74	0.82	0.85	0.88	0.88
	SVM	0.47	0.66	0.75	0.83	0.87	0.45	0.60	0.70	0.79	0.85
TeamMates	LR	0.60	0.78	0.84	0.87	0.89	0.61	0.75	0.82	0.85	0.88
	NB	0.66	0.79	0.82	0.84	0.85	0.66	0.78	0.81	0.82	0.84
	SVM	0.58	0.75	0.81	0.85	0.87	0.59	0.74	0.80	0.84	0.86

Table 4: Precision of recommendations provided by standard InMap (single ranked list of recommendations) for creating seed mappings of different relative sizes.

System / Rel. seed size	0.05	0.10	0.15	0.20	0.25
Ant	0.82	0.87	0.91	0.91	0.93
ArgoUML	0.83	0.88	0.91	0.92	0.93
Jabref	0.83	0.89	0.91	0.92	0.94
Jittac	0.86	0.75	0.78	0.80	0.83
TeamMates	0.77	0.85	0.84	0.87	0.88
Avg.	0.82	0.85	0.87	0.88	0.90

Next, we created seed mapping of different sizes based on InMap's recommendations as described for Step 3 in Sec. 3. In order to judge the effort required to create that seed mapping, i.e., the numbers of recommendations to be manually checked, we computed the precision of InMap. The results are shown in Tab. 4.

Table 5 shows the precision and recall scores in this setting for the same systems, classifiers, and seed mapping sizes as used in the baseline mapping with InMap alone (see Tab. 2). We observe a sharp drop in both precision and recall. At a relative seed mapping size of 0.25, the average precision values are 0.61 (−0.28 compared to baseline) for LR, 0.63 (−0.21) for NB, and 0.57 (−0.31) for SVM. The average recalls are 0.59 (−0.29) for LR, 0.62 (−0.21) for NB, and 0.58 (−0.29) for SVM.

Table 5: Precision and recall of classifiers trained with seed mappings of different relative sizes, generated by standard InMap.

		Weighted avg. precision					Weighted avg. recall				
System	Class.	0.05	0.10	0.15	0.20	0.25	0.05	0.10	0.15	0.20	0.25
Ant	LR	0.41	0.42	0.46	0.57	0.56	0.15	0.36	0.50	0.48	0.57
	NB	0.49	0.45	0.39	0.51	0.49	0.52	0.52	0.52	0.52	0.54
	SVM	0.39	0.41	0.45	0.54	0.56	0.15	0.42	0.54	0.56	0.62
ArgoUML	LR	0.03	0.45	0.47	0.56	0.63	0.13	0.43	0.46	0.52	0.66
	NB	0.03	0.47	0.61	0.63	0.63	0.14	0.55	0.57	0.65	0.67
	SVM	0.03	0.43	0.43	0.53	0.61	0.13	0.47	0.46	0.52	0.65
Jabref	LR	0.53	0.79	0.82	0.79	0.85	0.39	0.68	0.64	0.58	0.66
	NB	0.61	0.66	0.78	0.77	0.83	0.57	0.59	0.63	0.62	0.72
	SVM	0.50	0.74	0.73	0.74	0.80	0.39	0.64	0.60	0.55	0.66
Jittac	LR	0.35	0.27	0.23	0.34	0.36	0.28	0.42	0.42	0.44	0.43
	NB	0.49	0.60	0.55	0.53	0.54	0.57	0.56	0.54	0.54	0.52
	SVM	0.33	0.25	0.22	0.32	0.32	0.28	0.41	0.40	0.42	0.39
TeamMates	LR	0.55	0.57	0.66	0.64	0.64	0.28	0.27	0.27	0.56	0.65
	NB	0.64	0.64	0.65	0.61	0.64	0.62	0.56	0.58	0.65	0.66
	SVM	0.54	0.45	0.58	0.60	0.57	0.27	0.26	0.17	0.42	0.56

Table 6: Precision of modified InMap (based on per-module rankings) for creating seed mappings of different relative sizes.

System / Rel. seed size	0.05	0.10	0.15	0.20	0.25
Ant	1.00	1.00	1.00	1.00	0.99
ArgoUML	0.71	0.75	0.78	0.80	0.82
Jabref	0.67	0.76	0.80	0.84	0.86
Jittac	0.75	0.81	0.81	0.83	0.86
TeamMates	0.80	0.81	0.84	0.83	0.82
Avg.	0.79	0.83	0.85	0.86	0.87

In the last step of the experiment, we assume that InMap provides recommendation on a per-module basis instead of a single, system-wide list of recommendations. This means that recommendations that would be further down that system-wide list can be picked for creating a seed mapping in case they are among the higher ranked ones for the module that they refer to. By doing so, the resulting seed mapping is more balanced and provides training data for possible classification outcomes (i.e., modules) as long as InMap presents a recommendation. Table 6 shows the precision scores achieved by this modified variant of InMap. The loss of precision that could be expected because lower scoring, and hence potentially more likely wrong, recommendations are picked as compared to InMap's default behaviour, is actually very low (see also Tab. 4) and does not exceed 0.03.

The results show a moderate drop in precision for LR and SVM in most systems and for most seed mapping sizes; the decrease in precision is smaller for NB (Table 7). The average precision scores at a relative seed mapping size of 0.25 are 0.79 (−0.10 compared to baseline) for LR, 0.82 (−0.02) for NB, and 0.78 (-0.10) for SVM. Average recalls are 0.52 (−0.36) for LR, 0.71 (−0.12) for NB, and 0.51 (−0.26) for SVM. As compared to using InMap stand-alone for mapping the systems at hand, the precision of combining the modified InMap with any of the classifiers is at a similar level. The recall though is considerably lower.

Table 7: Precision and recall of classifiers trained with seed mappings of different relative sizes, generated by modified InMap (using per-module recommendations).

		Weighted avg. precision					Weighted avg. recall				
System	Class.	0.05	0.10	0.15	0.20	0.25	0.05	0.10	0.15	0.20	0.25
Ant	LR	0.50	0.58	0.67	0.72	0.77	0.11	0.14	0.27	0.29	0.36
	NB	0.61	0.64	0.68	0.69	0.74	0.32	0.30	0.39	0.44	0.46
	SVM	0.49	0.59	0.66	0.72	0.74	0.10	0.15	0.25	0.26	0.31
ArgoUML	LR	0.58	0.66	0.80	0.78	0.78	0.24	0.29	0.45	0.48	0.52
	NB	0.74	0.76	0.82	0.82	0.85	0.59	0.60	0.70	0.73	0.78
	SVM	0.56	0.68	0.83	0.80	0.78	0.24	0.30	0.43	0.45	0.47
Jabref	LR	0.83	0.85	0.92	0.92	0.93	0.35	0.37	0.71	0.79	0.82
	NB	0.89	0.93	0.93	0.95	0.95	0.70	0.73	0.75	0.78	0.82
	SVM	0.82	0.84	0.92	0.94	0.93	0.33	0.37	0.78	0.81	0.83
Jittac	LR	0.38	0.56	0.56	0.73	0.77	0.23	0.33	0.33	0.48	0.45
	NB	0.86	0.83	0.83	0.87	0.79	0.60	0.74	0.74	0.78	0.72
	SVM	0.38	0.64	0.64	0.68	0.81	0.22	0.34	0.34	0.39	0.52
TeamMates	LR	0.62	0.58	0.66	0.67	0.69	0.33	0.36	0.44	0.47	0.45
	NB	0.71	0.74	0.76	0.75	0.79	0.70	0.71	0.73	0.71	0.78
	SVM	0.59	0.55	0.54	0.61	0.65	0.33	0.34	0.30	0.33	0.40

5 DISCUSSION

In the following subsections, we discuss the results in the light of the motivating research questions as well as potential threats to validity.

5.1 Findings regarding RQ 1

We phrased RQ 1 as "How well do ML-based mappers performed if they are trained with automatically recommended seed mappings?". The presented results suggest that a seed mapping generated with standard InMap decreases the decreases the predictive performance of all investigated classifiers significantly beyond the point of usefulness. Even with a seed size of 25% of the overall system, at which both metrics generally indicate better results than at smaller seed sizes, average precision stays below 0.61 and average recall peaks at 0.62 (both for NB). A notable exception is Jabref, for which all classifiers achieve a precision over 0.8 and recall between 0.66 and 0.72.

The reason for this drop in classification performance is most likely due to the imbalanced representation of architectural modules in the seed mappings. Figure 2 visualizes the distribution of modules as targets of mappings contained in seed mappings at different seed mapping sizes (0.1, 0.15, 0.2, 0.25) for Ant. For a relative size of 0.1 and 0.15, only five of the in total 15 modules of Ant are represented in the seed mapping. This affects the training and performance of the classifiers significantly as there is no training data for ten of the modules. The resulting classifiers will not be able to map entities belonging to these modules. In the example of Ant, all classifiers show an improvement in precision when moving from a relative seed mapping size of 0.15, representing only five modules to 0.2, representing nine modules (see Tab. 5). The impact of this imbalance, or rather the lack of it, might also explain the exceptional behaviour for Jabref since four out of five architectural modules are represented in the seed mapping at a relative size of 0.1 and higher (see the visualization contained in the replication package).

The results also indicate that picking the best mapping recommendations per module instead of selecting the best mapping from a single, system-wide list of recommendations creates seed mappings better suited for training code-to-architecture mappers. All

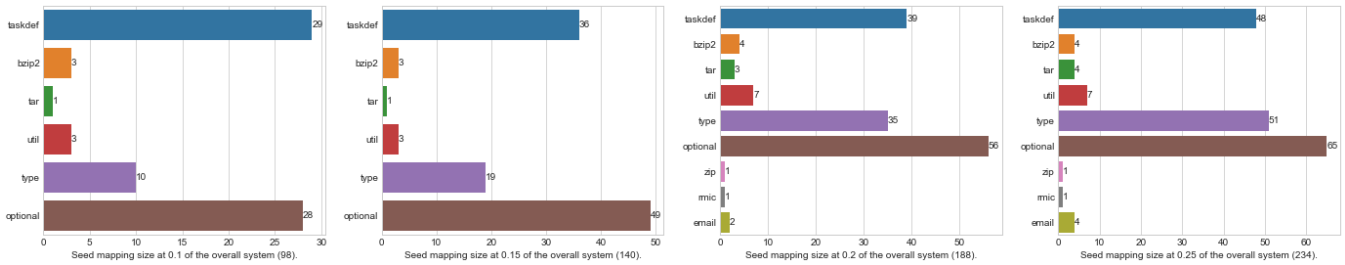

Figure 2: Example (Ant) of seed label distributions created by InMap.

three investigated mappers show improved precision, however, only Naive Bayes also improves in recall compared to feeding the seed mapping with the system-wide best recommendations. This is an interesting contrast to Florean et al.'s experiments with standard InMap and their conclusion to favour LR or SVM over NB.

In conclusion, the results suggest that only mappers based on NB, trained with a seed mapping on a best-per-module basis, show sufficiently accurate mapping performance.

5.2 Findings regarding RQ 2

In order to address the question of how efficient the combinations automatically recommended seed mappings and the presented classifiers are, we assume that the main aspect influencing the efficiency of the mapping process is the amount of manual labour involved. This means that compared to manual effort, time and resources required for training classifiers, classification, running InMap, etc., are neglectable.

The manual work involved in creating a code-to-architecture mapping with InMap is caused by two tasks: reviewing the provided mapping recommendations and mapping any remaining unmapped entities manually. A low precision causes a large effort for reviewing as the overhead caused by inspecting incorrect mapping recommendations is big. A low recall indicates that many entities remain unmapped and require a manual mapping.

Table 8 shows the number of reviewed recommendations and manually mapped entities for each of the five systems used in the experiments. These values can easily be derived from the values for the numbers of total files (entities to be mapped), numbers of recommendations, and numbers of correct recommendations shown in Table 2. Additionally, we look at a hypothetical system ("HS") with 1,000 entities to be mapped. The figures for HS assume that InMap shows precision and recall for this system as averaged over the other five systems (see Sec. 4). Given an average precision of 0.94, sixty entities in HS would remain unmapped and need manual mapping in such a scenario. In addition to the 940 correct mapping recommendations, the imperfect average precision of 0.83 would cause an overhead of incorrect recommendations. The total number of recommendations is calculated as the number of correct recommendations divided by the precision.

For computing the overall effort, we assume that each instance of of any of the two types of tasks requires the same constant amount of units of effort, e.g., one unit. The total effort as depicted in Table 8 is then linearly correlated to the sum of reviewed recommendations and manually mappings.

Table 8: Manual efforts for mapping with InMap only.

System	Recommendations reviewed	Manual Mappings	Total Effort
Ant	1,037	16	1,053
ArgoUML	1,010	5	1,015
Jabref	882	15	897
Jittac	137	8	145
TeamMates	272	62	334
HS	1,133	60	1,193

Combining seed mapping generation with InMap and automated mapping of the remaining entities requires three manual tasks: Reviewing InMap's recommendations for the seed mapping, reviewing the automated mapping, and fixing wrong, automatically made mappings. The effort for reviewing the seed mapping recommendations depends obviously on the size of the seed mapping and the precision of InMap. The size of the seed mapping is inversely proportional to the effort for reviewing the automated mapping as the more is mapped in the seed, the fewer entities are mapped automatically and require manual checks. The amount of effort caused by having to fix mapping manually is obviously proportional to a classifier's accuracy (equivalent to its weighted average recall). Provided high classification accuracy, one could argue against the proportionality related to checking the automated mapping. This would be the case if users started "blindly trusting" the automatic mapping because it was only wrong in exceptional cases and rare mistakes would be fixed later.

In Tab. 9, efforts are described for using the modified InMap variant for the different seed mapping sizes and NB as classifier for the subsequent automatic mapping. For the hypothetical system, the numbers of seed mappings reviewed are computed based on the precision of InMap as shown in Tab. 6. The number of reviewed automatic mappings is result of subtracting the number of entities covered in the seed mapping from the overall number of entities in the system. From Tab. 7, we can compute average values of (weighted average) recalls, or accuracy, for NB for each relative seed mapping size. Those values can then be used to compute the number of incorrect automatic mappings as the total number of automatic mappings minus the correct ones, determined by multiplying the total number by the accuracy of the automatic mapper.

As before, we assume that all instances of these tasks cause the same constant amount of effort, i.e., one unit. A comparison

Table 9: Manual efforts for combining InMap and NB.

System	Relative seed size	Seed rec. reviewed	Autom. Mapp. Reviewed	Mappings Fixed	Total Effort
Ant	0.05	45	693	471	1,209
Ant	0.10	73	665	466	1,204
Ant	0.15	106	632	386	1,124
Ant	0.20	124	614	344	1,082
Ant	0.25	148	591	319	1,058
ArgoUML	0.05	66	716	294	1,076
ArgoUML	0.10	99	689	276	1,064
ArgoUML	0.15	126	665	200	991
ArgoUML	0.20	148	645	175	968
ArgoUML	0.25	180	615	135	930
Jabref	0.05	61	811	243	1,115
Jabref	0.10	86	787	212	1,085
Jabref	0.15	108	764	191	1,063
Jabref	0.20	127	745	164	1,036
Jabref	0.25	149	724	130	1,003
Jittac	0.05	12	108	43	163
Jittac	0.10	21	100	26	147
Jittac	0.15	21	100	26	147
Jittac	0.20	29	93	20	142
Jittac	0.25	35	87	24	146
TeamMates	0.05	25	301	90	416
TeamMates	0.10	36	292	85	413
TeamMates	0.15	56	274	74	404
TeamMates	0.20	66	266	77	409
TeamMates	0.25	87	250	55	392
HS	0.05	63	950	399	1,412
HS	0.10	120	900	342	1,362
HS	0.15	176	850	289	1,315
HS	0.20	233	800	248	1,281
HS	0.25	287	750	218	1,255

between the efforts shown in Tab. 8 and Tab. 9 shows that combining modified InMap with automatic mapping with NB as described reduces the overall effort for ArgoUML (by ca. 8.4%) and Jittac (by ca. 2.1%). For the other three systems, the combined approach lies behind by 9.9% on average. For HS, the shortfall is about 5.2%.

We therefore state regarding RQ 2 that the investigated combination of InMap with the best-performing classifier might perform similarly efficient in terms of manual effort required, and even slightly better than InMap stand-alone. In other instances though, the combined approach might fall behind significantly. On average, the results suggest that using InMap stand-alone is more efficient.

5.3 Validity

The selection of subject systems poses a threat to external validity. The systems are small- to medium-sized desktop applications that most certainly do not represent the full population of software systems. The requirements of having a validated mapping to compare mapping performance against and having descriptions of architectural modules reduce the number of systems that could be used off-the-shelf for a study like the presented one. Due to lack of resources, creating mappings for a larger and more diverse set of software systems was deemed unfeasible for this study. Instead we opted for selecting the largest set of systems known to us that fulfil the above requirements.

This implies another potential threat to external validity as all systems are written in Java, which is another requirement imposed by the tooling applied (InMap). We believe, however, that the results are generalizable to systems of similar size and written in other statically typed and imperative/object-oriented languages. Missing type declarations, for example, might cause InMap to be less effective and textual classification to be less accurate.

We limited the study to three types of classifiers only, which is a potential threat to external validity. They were selected as previous studies showed they performed reasonably well for the task at hand and with small amount of data in general. With larger systems, the use and comparison of different neural network architectures could be one direction to extend the external validity in this field of research. The same is true for using more advanced code representation models, such as CodeBERT or code2vec [2, 7].

One implicit result of the conducted experiment is that selecting the seed mapping on a best-per-module basis improves the predictive performance of automatic mappers We are confident that the internal validity of the experiments is high as we repeated the experiments for the three seed mapping generation methods with identical values for other independent variables (such as InMap settings, training setting, seed mapping sizes, etc.). Moreover, we are very confident that no circumstantial or environmental changes between experiments could have caused the observed changes in predictive capabilities and efficiency.

A potential threat to construct validity is posed by the assumption made to quantify the manual effort involved in mapping code to architecture. In particular the assumption that fixing an incorrect mapping or manually mapping an entity causes as much work as reviewing mappings might be debatable. To the best of our knowledge there is no empirical evidence to our knowledge that would provide any quantitative data on this issue. We therefore decided to present the detailed figures on effort for the different tasks such that researchers can easily recalculate efforts based on their own assumptions. Only empirical studies investigating how practitioners work with mapping recommendations and automatic mappers could eventually provide evidence for more reliable assumptions.

6 CONCLUSION

The results obtained from the experiment described in this article suggest that, on average, using InMap for creating seed mappings for ML-based code-to-architecture mappers leads to mappers with below-average predictive capabilities. Picking the seed mapping from module-specific lists of recommendations instead of a system-wide lists generally improves the observed precision. Naive Bayes outperformed Logistic Regression and SVM when used in combination with InMap. This is contrasted by the results with random seed mappings in which any of the two seemed to be the better alternatives to Naive Bayes[8].

Furthermore, the manual effort in performing the mapping of a full system seeded this way appears to be slightly higher than with InMap alone. For individual systems, however, the trained automatic mapper performs on par or slightly better than InMap stand-alone. Further investigations are needed to understand the circumstances in which this way of hybrid mapping appears favourable.

In order to improve the generation of seed mappings, it may be worthwhile to consider further information from source code in future studies. Several studies by Olsson et al. move towards that direction and suggest to initially map entities with high dependency fan-out and use keywords as pointers for the mapping [13, 16]. Integrating this for manually mapping those potential "high-impact

entities" even before the full seed mapping is created through InMap might make the overall mapping even more efficient.

Another direction to increase efficiency could be to integrate InMap and ML-based mappers incrementally in the sense that the latter only returns a partial mapping of the "best" candidates, judged, for example, based on the actual score that Naive Bayes computes. This could be reviewed and fed back to InMap to start another cycle. In this scenario, ML-based mappers are rather used as a recommendation system in itself than as automatic mappers. Employing methods of ensemble learning that integrate prediction of several models (mappers), e.g., through voting, could be another mean to improve automated code-to-architecture mapping [26].

Recent works on InMap propose hierarchical extensions to the approach that recommend mapping for larger units of source code, such as packages, instead of only considering atomic entities [24]. This might reduce the effort for the mapping drastically as long as the structure in which the source code is organized hierarchically is reasonably well-aligned with the intended software architecture of a system. The combination of InMap recommending mappings for hierarchical groups of implementation entities as one and machine learning techniques has not been explored yet. More empirical studies are required to better understand the factors and steps driving the effort in tool-supported code-to-architecture mapping. Such studies are also necessary to better estimate the trade-off between having to review automatically suggested mappings, finding mapping manually, and fixing incorrect automatic mappings.

ACKNOWLEDGMENTS

This research was partly funded by Region Värmland via the DigitalWell project.

REFERENCES

[1] Nour Ali, Sean Baker, Ross O'Crowley, Sebastian Herold, and Jim Buckley. 2018. Architecture Consistency: State of the Practice, Challenges and Requirements. *Emp. Softw. Eng.* 23, 1 (Feb. 2018), 224–258. https://doi.org/10.1007/s10664-017-9515-3

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. Code2vec: Learning Distributed Representations of Code. *Proc. ACM Program. Lang.* 3, POPL, Article 40 (jan 2019), 29 pages. https://doi.org/10.1145/3290353

[3] Roberto Almeida Bittencourt, Gustavo Jansen de Souza Santos, Dalton Dario Serey Guerrero, and Gail C. Murphy. 2010. Improving Automated Mapping in Reflexion Models Using Information Retrieval Techniques. In *2010 17th Working Conference on Reverse Engineering*. 163–172. https://doi.org/10.1109/WCRE.2010.26

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

[5] Andreas Christl, Rainer Koschke, and Margaret-Anne Storey. 2007. Automated clustering to support the reflexion method. *Information and Software Technology* 49, 3 (2007), 255–274. https://doi.org/10.1016/j.infsof.2006.10.015 12th Working Conference on Reverse Engineering.

[6] Constanze Deiters, Patrick Dohrmann, Sebastian Herold, and Andreas Rausch. 2009. Rule-Based Architectural Compliance Checks for Enterprise Architecture Management. In *2009 IEEE International Enterprise Distributed Object Computing Conference*. 183–192. https://doi.org/10.1109/EDOC.2009.15

[7] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[8] Alexander Florean, Laoa Jalal, Zipani Tom Sinkala, and Sebastian Herold. 2021. A Comparison of Machine Learning-Based Text Classifiers for Mapping Source Code to Architectural Modules. In *Companion Proceedings of the 15th European Conference on Software Architecture : (CEUR Workshop Proceedings)*, Vol. 2978. CEUR-WS.

[9] Michael W. Godfrey and Eric H. S. Lee. 2000. Secrets from the Monster: Extracting Mozilla's Software Architecture. In *In Proc. of 2000 Intl. Symposium on Constructing software engineering tools (CoSET 2000*. 15–23.

[10] Sebastian Herold, Martin Blom, and Jim Buckley. 2016. Evidence in architecture degradation and consistency checking research: preliminary results from a literature review. In *Proccedings of the 10th European Conference on Software Architecture Workshops, Copenhagen, Denmark, November 28 - December 2, 2016*, Rami Bahsoon and Rainer Weinreich (Eds.). ACM, 20.

[11] Daniel Link, Pooyan Behnamghader, Ramin Moazeni, and Barry Boehm. 2019. 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, 64–73. https://doi.org/10.1109/ICSSP.2019.00018

[12] G.C. Murphy, D. Notkin, and K.J. Sullivan. 2001. Software reflexion models: bridging the gap between design and implementation. *IEEE Transactions on Software Engineering* 27, 4 (2001), 364–380. https://doi.org/10.1109/32.917525

[13] Tobias Olsson, Morgan Ericsson, and Anna Wingkvist. 2018. Towards improved initial mapping in semi automatic clustering. In *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings, ECSA 2018, Madrid, Spain, September 24-28, 2018*, Jennifer Pérez, Raffaela Mirandola, and Hong-Mei Chen (Eds.). ACM, 51:1–51:7. https://doi.org/10.1145/3241403.3241456

[14] Tobias Olsson, Morgan Ericsson, and Anna Wingkvist. 2019. 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, 209–216. https://doi.org/10.1145/3344948.3344984

[15] Tobias Olsson, Morgan Ericsson, and Anna Wingkvist. 2021. s4rdm3x: A Tool Suite to Explore Code to Architecture Mapping Techniques. *Journal of Open Source Software* 6, 58 (2021), 2791. https://doi.org/10.21105/joss.02791

[16] Tobias Olsson, Morgan Ericsson, and Anna Wingkvist. 2022. Mapping Source Code to Modular Architectures Using Keywords. In *Software Architecture*, Patrizia Scandurra, Matthias Galster, Raffaela Mirandola, and Danny Weyns (Eds.). Springer International Publishing, 65–85.

[17] Leonardo Passos, Ricardo Terra, Marco Tulio Valente, Renato Diniz, and Nabor Mendonça. 2010. Static Architecture-Conformance Checking: An Illustrative Overview. *IEEE Software* 27, 5 (2010), 82–89. https://doi.org/10.1109/MS.2009.117

[18] Dewayne E. Perry and Alexander L. Wolf. 1992. Foundations for the Study of Software Architecture. *SIGSOFT Softw. Eng. Notes* 17, 4 (Oct. 1992), 40–52. https://doi.org/10.1145/141874.141884

[19] Santonu Sarkar, Shubha Ramachandran, G. Sathish Kumar, Madhu K. Iyengar, K. Rangarajan, and Saravanan Sivagnanam. 2009. Modularization of a Large-Scale Business Application: A Case Study. *IEEE Software* 26, 2 (2009), 28–35. https://doi.org/10.1109/MS.2009.42

[20] Kanish Shah, Henil Patel, Devanshi Sanghvi, and Manan Shah. 2020. A comparative analysis of logistic regression, random forest and KNN models for the text classification. *Augmented Human Research* 5, 1 (2020), 1–16.

[21] A. Sheshasaayee and G. Thailambal. 2017. Comparison of Classification Algorithms in Text Mining. *International Journal of Pure and Applied Mathematics* 116, 22 (2017), 425–433.

[22] Zipani Tom Sinkala and Sebastian Herold. 2021. 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, 1439–1442. https://doi.org/10.1145/3412841.3442124

[23] Zipani Tom Sinkala and Sebastian Herold. 2021. InMap: Automated Interactive Code-to-Architecture Mapping Recommendations. In *2021 IEEE 18th International Conference on Software Architecture (ICSA)*. https://doi.org/10.1109/ICSA51549.2021.00024

[24] Zipani Tom Sinkala and Sebastian Herold. 2022. Hierarchical Code-to-Architecture Mapping. In *Software Architecture*, Patrizia Scandurra, Matthias Galster, Raffaela Mirandola, and Danny Weyns (Eds.). Springer International Publishing, Cham, 86–104.

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

[26] Zhi-Hua Zhou. 2012. *Ensemble Methods: Foundations and Algorithms* (1st ed.). Chapman & Hall/CRC.