Detecting Model View Controller Architectural Layers using Clustering in Mobile Codebases

Dragoş Dobrean[a] and Laura Dioşan[b]
Faculty of Mathematics and Computer Science, Babeş Bolyai University, Cluj-Napoca, Romania

Keywords: Mobile Applications Software Architecture Analyser, Automatic Analysis of Software Architectures, Structural and Lexical Information, Software Clustering.

Abstract: Mobile applications are one of the most common software projects written nowadays. The software architectures used for building those type of products heavily impacts their lifecycle as the architectural issues affect the internal quality of a software system hindering its maintainability and extensibility. We are presenting a novel approach, Clustering ARchitecture Layers (*CARL*), for detecting architectural layers using an automatic method that could represent the first step in the identification and elimination of various architectural smells. Unlike supervised Machine Learning approaches, the involved clustering method does not require any initial training data or modelling phase to set up the detecting system. As a further key of novelty, the method works by considering as codebase's hybrid features the information inferred from both module dependency graph and the mobile SDKs. Our approach considers and fuses various types of structural as well as lexical dependencies extracted from the codebase, it analyses the types of the components, their methods signatures as well as their properties. Our method is a generic one and can be applied to any presentational applications that use SDKs for building their user interfaces. We assess the effectiveness of our proposed layer detection approach over three public and private codebases of various dimensions and complexities. External and internal clustering metrics were used to evaluate the detection quality, obtaining an Average Accuracy of 77,95%. Moreover, the Precision measure was computed for each layer of the investigated codebase architectures and the average of this metric (over all layers and codebases) is 79,32% while the average Recall on all layers obtained is 75,93%.

1 INTRODUCTION AND CONTEXT

Mobile applications and presentational software products represent an important part of the realm of software development. We are furthering our work towards creating a system for improving the architectural correctness of those types of projects. By respecting an architectural pattern the codebase becomes more testable and more extensible in the areas which are important for the business. In addition to this, having a well-defined architecture in place helps the new or inexperienced developers write new code more easily by having clear architectural guidelines in place.

In this paper we present Clustering ARchitecture Layers (*CARL*), an unsupervised approach for solving the same problem using clustering. In order to pave the way for analyzing more specialized architectures for which a heuristic approach might not yield best results as the components are hard to be split into layers by using various rules and sometimes those rules cannot be easily inferred as all the codebases have their own particularities we are focusing on automating this process by using AI algorithms, most specifically clustering.

Clustering (Russell and Norvig, 2016) is one of the most popular unsupervised Machine Learning approaches for finding groups (clusters) in a set of data that exhibit similarities. When talking about software architectures, the elements composing the architectural layers should have similarities since they fulfill the same role. For instance in the case of Model View Controller (MVC), data manipulation for the Model layer, control of the data flow for the Controller layer and handling the input and output in the View layer.

In this study, we search for the right information

[a] https://orcid.org/0000-0001-7521-7552
[b] https://orcid.org/0000-0002-6339-1622

Dobrean, D. and Dioşan, L.
Detecting Model View Controller Architectural Layers using Clustering in Mobile Codebases.
DOI: 10.5220/0009884601960203
In *Proceedings of the 15th International Conference on Software Technologies (ICSOFT 2020)*, pages 196-203
ISBN: 978-989-758-443-5

to be passed to the clustering algorithms in order to be able to separate the elements of the codebase in architectural layers. Some of the practical applications of such approach include: education – where developers and students could learn easily to develop better code, professional software development – where systems like mobile *CARL* can be used for providing insightful information to the management and to the developers regarding the architectural health of a codebase.

The following section presents previous work in the domain of architecture reconstruction using Machine Learning (ML) and Artificial Intelligence (AI) methods as well as details regarding the chosen architectural pattern (MVC). Section 3 talks about our approach *CARL*. The evaluation process and the conducted experiments together with the results of the conducted experiments are presented in 4. Section 5 shows the threats to validity of our proposal, while the last part (Section 6) states our conclusions and some ideas for further work.

2 BACKGROUND

In this paper, we are continuing our work in the area of mobile software architectures examination and we are proposing a novel, AI-based, automatic way of detecting the components of each architectural layer in a mobile codebase. The MVC architecture is used as the base architectural pattern for our study as it's one of the most used and well known presentational architectural patterns (Daoudi et al., 2019).

2.1 Model View Controller

The MVC is one of the most widespread presentational architectural patterns. It is used extensively in all sorts of client applications, web, desktop and mobile. It provides a simple separation of concerns between the components of a codebase in 3 layers: **Model** – responsible for business logic, **View** – responsible for the user's input and the output of the application and **Controller** – keeps the state of the application, acts as a mediator between the Model and the View layer. There are many flavors of MVC in which the data-flow is different but they all share the same model of separation. MVC is also the precursor of more specialized presentational architectural patterns such as Model View View Model (MVVM) or Model View Presenter (MVP) (Daoudi et al., 2019).

2.2 Related Work

Architecture reconstruction methods are two-folded: identification of architectural modules (by clustering) and identification of architectural rules among modules. Several approaches were proposed to support the architecture reconstruction process based on static analysis. A part of these approaches exploit the structural information extracted from the codebase ((Mancoridis et al., 1999), (Mitchell and Mancoridis, 2008)), another part exploit the lexical information ((Anquetil and Lethbridge, 1999), (Kuhn et al., 2007)), while some recent approaches exploit both of them ((Garcia et al., 2013), (Tzerpos and Holt, 2000), (Le et al., 2015), (Saeidi et al., 2015)). Furthermore, few of them consider the architectural style of the system under analysis. Because our empirical validation is performed in the context of three iOS applications and Apple's flavour of MVC which can be viewed as a linear layered architectural pattern, we describe in what follows several approaches that take into account the particularities of the layered-based architecture.

Similar to the general systems case, the approaches developed for analysing the layer-style projects take into account either the structural features of the codebase ((Müller et al., 1993), (Sangal et al., 2005), (Sarkar et al., 2009), (Constantinou et al., 2011)) or the hybrid (structural and lexical) features ((Scanniello et al., 2010), (Belle et al., 2013), (Belle et al., 2014), (**?**)).

Muller (Müller et al., 1993) identified various building blocks (e.g., variables, procedures, modules, and subsystems) by using composition operations that respect two principles: low coupling and high cohesion. Lattix tool (Sangal et al., 2005) extracted the inter-module dependencies among the code modules by conventional static analysis. Sarkar (Sarkar et al., 2009) proposed a human-assisted approach (in fact a semi-automatic approach) to identify the intended organization of the modules into layers by analyzing the source code. The approach combines the structural as well as non-structural domain-specific information to assign a module to a layer. The k-means clustering is involved in their approach; there are also proposed three layering principles that are related to the common violations of the layered architectures as well as a set of metrics that measure the violation of these principles. Constantinou (Constantinou et al., 2011) investigated how to design metrics that can reveal architectural information about a software system and more specifically, how architectural layers are correlated to design metrics. In such an approach, the design metrics do not reflect the structural information about a codebase.

Scanniello (Scanniello et al., 2010) used the link analysis algorithm for identifying the layers, and for each layer, they run a k-means clustering for grouping the modules from that layer; a lexical similarity of two modules is computed by considering the comments and the text associated to the module's code. No constraints regarding the layered-style architecture are actually involved in their approach. Belle (Belle et al., 2013) started by an approach based only on structural dependencies. Belle improved this approach by taking into account some lexical information (the namespace of a package) (Belle et al., 2014).

3 CLUSTERING ARCHITECTURE LAYERS

Clients – presentational software applications — are usually self-contained and they commonly use monolithic architectures. Generally, their codebase is split into layers that contain components that serve a common well-defined purpose. Each of those layers achieves a macro purpose.

An architectural layer represents the macro purpose of a set of components in the codebase; individual components within the layer fulfill other micro purposes in order to make the macro one possible. For instance, the macro purpose would be the separation between the UI interaction logic from the database handling operations. As micro purposes, we could have for example a component from the Model layer which notifies other components regarding a state change – its micro purpose being to notify.

In this study, we are interested in identifying the architectural layers of the codebase by detecting the purpose they serve based on the information available in the codebase. Our analysis is a static one and uses only the information in the interfaces (API contracts, public and private method and properties definition, inheritance information) of the investigated components, without looking at their implementation (the body of the functions). In addition to this, we do not analyze the logic of the components or the way the dependencies are being used.

We start this study with the idea that elements within the same architectural layer should have similarities like: the same purpose, a similar code or interface, a similar contribution to the system's flow and in order to detect those we plan to use an automatic approach which is characterized by two important properties: unsupervised and autonomous.

Unsupervised belongs to the Machine Learning's perspective and means that the algorithm needs no prior knowledge of the data it is about to process, not requiring any ground truth. In the case of software identification, by ground truth, we refer to the labels manually associated with the components by one or more human experts in the field. This ground truth information is not involved in the learning process of identification; it is considered only in the validation phase of the proposed approach, for computing the performance metrics. To the best of our knowledge, there is no benchmark dataset of labeled applications in the mobile domain. The same validation procedure was followed in a recent study about the identification of architectural patterns in Android applications by using particular and language-dependent heuristics (Daoudi et al., 2019).

Autonomous implies that there is no developer involvement in the process; by contrast, there are techniques in which the algorithm is directed to the solution by means of developer intervention (those approaches based on UML (Sangal et al., 2005), for instance). The autonomy of the proposed solution can be evaluated also in terms of dependencies to particular libraries, APIs, languages. Even we will exemplify and validate our solution in the iOS ecosystem, the proposed approach is a generic one; it can easily be implemented on other platforms that deal with presentational architectures and a framework for building the user interfaces.

In terms of Machine Learning, the performed analysis is considered a clusterization problem: grouping components into clusters without knowledge of the category they belong to. The clustering process follows three important steps: extract relevant information (features) from the raw data (known as instances); use all the features or just some of them to analyze the similarities among components and to build a clustering model; the output generated in this step is, in fact, a division of the data; validate the obtained clusters by using evaluation standards.

We name our approach Clustering ARchitecture Layers (*CARL*) and in what follows we will detail all these steps, emphasizing the particularities of mobile software clustering.

For **evaluating the performance** of our approach, several clustering performance metrics can be used (Pfitzner et al., 2009). Those are based on a predefined classification of the items to be grouped that reflects prior information on the data, which is used as standard and called ground truth (Mutual information, Rand index, Precision, Recall, F-measure, etc.).

4 NUMERICAL EXPERIMENTS

MVC was studied and analyzed by practitioners and academia (Daoudi et al., 2019). Moreover, by using MVC for validating our approach we open the way for other studies which might analyze more sophisticated architectural patterns, especially those descending from MVC.

Our analysis was focused on the iOS platform; however, the same process can be applied to any other platform which uses MVC and SDKs for building UI interfaces. In order to validate our idea we answered the following research questions.
RQ1: What features from the codebase can be used in a clustering algorithm?
RQ2: How effective is the proposed categorization method compared to manual inspections?
RQ3: What are the downsides of clustering a mobile codebase for detecting architectural layers?

4.1 Analysed Codebases

In order to validate our approach, we have conducted experiments on three different codebases, both open-source and private, of different sizes: Wikipedia - public information application (Wikimedia, 2018), Trust - public cryptocurrency wallet (Trust, 2018), E-Commerce - a private application.

Table 1: Short description of investigated applications.

Application	Blank	Comment	Code	Components
Wikipedia	6933	1473	35640	253
Trust	4772	3809	23919	403
E-Commerce	7861	3169	20525	433

Table 1 presents the characteristics of the codebases: blank – refers to empty lines, comment – represents comments in the code, code states the number of code lines, while components represent the total number of components in the codebase.

4.2 Evaluation Metrics

For evaluating our approach we asked 2 senior iOS developers (with over 5 years experience) to manually label all the components from all the codebases and we have used the manual inspection as ground-truth. Having these annotations, accuracy, precision and recall metrics can be computed (Fawcett, 2006):

- accuracy: $Acc = \frac{N^{AllLayers}_{DetectedCorrectly}}{N_{allComponents}}$
- precision for the layer X: $P_X = \frac{N^X_{DetectedCorrectly}}{N^X_{TotalDetected}}$
- recall for the layer X: $R_X = \frac{N^X_{DetectedCorrectly}}{N^X_{GroundTruth}}$,

where: $N^X_{DetectedCorrectly}$ is the number of components detected by the system which belong to the X layer and are found in the ground truth for that layer; $N^X_{TotalDetected}$ is the number of components detected by the system as belonging to the X layer; $N^X_{GroundTruth}$ is the number of the components which belong to the X layer in the ground truth.

In (Garcia et al., 2013) the authors defined the *cluster-to-cluster* (c2c) metric which is equal to the Precision metric P_X. Precision is the term used in machine learning, while c2c is commonly used in software engineering approaches; however for the current study they mean the same thing.

4.3 Methodology

We design two stages for our study: a preliminary stage and a complex stage. In the preliminary phase, we have considered only a mobile application and we tried to cluster its components in the corresponding layers by an incremental process in terms of selected components' features as follows:

- **Number of Dependencies** (F_1)**:** how many dependencies a component has with each of the other codebase's components; we use $F_1(c_i)$, for all components c_i ($i \in \{1,2,\ldots,n\}$) of a codebase;
- **Presence of Dependencies** (F_2)**:** the type of dependencies that it has with each of the other codebase's components;
- **Name Distance** (F_3)**:** how many dependencies it has with each of the other codebase's components and the distances between the name of the current component and the names of the other codebase's components;
- **Keywords Presence** (F_4)**:** the features F_3 are enriched by the keyword-based features;

For all these feature subsets, the same clustering algorithm and validation step are applied (see the details in Section 4.4).

The best subset of features identified in the preliminary study is used in a second more complex one, where three applications are investigated (see the details in Section 4.5). Finally, the findings are analyzed and possible improvements are suggested.

For the rest of the study, we are focusing only on the iOS platform and we are analyzing Swift codebases. However, our proposed approach can be easily extrapolated to other platforms that use SDKs for building user interfaces and are presentational software products.

Since we have analyzed Swift codebases all the methods defined in extensions of classes defined in the codebase were added to the extended class and the extensions were not taken into consideration when constructed the features since they represent the same component. Moreover, all the protocols were also ignored because they only provided blueprints for the behavior of a component. If the component implements a protocol then the methods and the behavior will also be present in the implementing component.

4.4 Preliminary Evaluation

In order to test different approaches and to see which information from the codebase or what combination yields the best results, we have chosen a medium-sized iOS application (20525 lines of code) that uses the MVC architecture and which was used as a benchmark when applying different clustering approaches. The application used as a benchmark in the approach section is the E-Commerce one. We have asked an expert to manually label all the components in the benchmark application. These labels were used to calculate the precision and the recall for all the layers as well as the accuracy in order to compare different approaches. Based on the information used by the clustering algorithm we describe all three different attempts at classifying the components of a codebase. In all our approaches the features are encoded into $n \times |F(c)|$ matrices. The number of lines (n) corresponds to the number of the components in the analyzed codebase and is constant in all scenarios, while the number of columns ($|F(c)|$) varies and is equal to the number of features considered.

Features based on the Number of Dependencies. Our first attempt was to group the components by taking into account the dependencies between them. The features associated with a component of the codebase correspond to the number of dependencies between that component and all the other components of the investigated application. In fact, these features of a component A are stored as vectors of length equal to the total number of components from the codebase (n). The i^{th} element in this vector can be either 0 (when no dependency between A and the i^{th} component was found) or a positive integer value that represents the number of dependencies between A and the i^{th} component of the codebase. A dependency represents a link between two components and takes the form of associations or inheritance relations. The dependencies were extracted by analyzing the method signatures and properties of the components. For instance, if component A has a property of type B, then we have a dependency between A and B. It is important to mention that the matrix of elements is not symmetric: if component A has a link to a component B, that does not mean that the component B also has a dependency on the A component. We call this version of our approach *CARL-F_1*. The accuracy obtained was **24.82%**, the best recall was for the View layer, while the precision was maxim for the Controller (see Table 2). The approach did not yield good results as we had many components which have the same number of dependencies (similar features), but they were not similar in term of concerns and roles that they implement. E.g. we have components from the View layer which had only one other dependency and, in the same time, we have items in the Model layer with only one dependency.

Features based on the Type of Dependencies. In order to improve the performance of the first clustering attempt we have restricted the dependency-based features to only two values: 0 if no dependency between the elements was found or 1 otherwise. This approach yields better results than the previous one, as in this approach the direction of the dependencies matters more than their number. However, we have chosen not to pursue this path as when we enhanced the data provided to the clustering algorithms, the results were not different than using the first approach – *features based on the number of dependencies* in combination with the new data. In addition to this, to provide more data to the clustering algorithm while using this approach, we would have doubled the width of the matrices and then we would run into a feature selection issue. We call this version of our approach *CARL-F_2*. The accuracy obtained in this scenario was almost doubled compared to the first approach – Accuracy is **46.60%** in Table 2.

Hybrid Features based on Dependencies and on the Component's Name. To further improve the clustering of the components more information had to be provided regarding the application's elements. After manually analyzing several mobile codebases, we have seen that there is a correlation between the names of the components and the layers in which they would reside. Usually components from the same layer have similar names (*SellerViewController*, *BuyerViewController* – for the Controller layer or *SellerItemView*, *BuyerItemView* for the View layer) In order to score the similarity between the names of two components, the Levenshtein distance (Levenshtein, 1966) was computed. For every component A in the codebase, we have fused, by addition, the number of dependencies between A and another component B and the string distance between the A's and B's names. In this manner, for each component a set of hybrid (structural and lexical) n features is obtained. This version is called *CARL-F_3*. With the new information regarding the distance between the components, the accuracy was improved to **52.46%**– see Table 2.

Hybrid Features based on Dependencies, on the Component's Name and on Keywords. Another important aspect we have discovered while manually analyzing mobile codebases was the fact that the components had certain keywords in their names based on the layer in which they reside. We have noticed that for the Controller layer, the *controller* keyword was present in a large majority of its components, this is also true for the View layer with the *view* keyword. We have enhanced the feature set from the *CARL-*F_3 approach by considering for each component two new elements: one for the *controller* keyword and one for the *view*. A component is actually represented by $n+2$ features. If the name of the component contained the *view* keyword, a value of 500 was considered, otherwise the feature was set to 0 ($W(c_i,'view')$ is 500 or 0). In the case of the *controller*, if the name of the component contained the keyword, a value of 10.000 was used, otherwise, the feature entry was 0 ($(Kw(c_i,'controller')$ is 10.000 or 0). The large numbers were chosen in order for this information to be viewed as stronger (in opposition to the distance between the names) by the clustering algorithm. The values of 500 and 10.000 were also chosen because the elements from the Controller layer, most specifically ViewController elements that account for a large majority of all the components in this layer, contained both keywords. Thus, their chances of being correctly clustered increased. We call this version of our approach *CARL-*F_4. The accuracy was improved by using the above-mentioned information in the clustering process with over 20% reaching **78.45%** (see Table 2) on the benchmark application.

Hybrid Features based on Dependencies, on the Component's Name, on Keywords and on SDK Inheritance. In the last scenario, we have enhanced the *CARL-*F_4 method by taking into account information regarding the type of the components, also. In the mobile SDKs, we have components from which the developers inherit in order to implement certain parts of the application. In a typical application, all the elements presented on the screen inherit from an SDK defined View element. This is also true for a large number of Controller elements – which handle both the input received from the user and the state of the application. We have constructed lists of all the View and Controller SDK defined elements and we have provided the clustering algorithm information about whether or not a component inherits from one of those. In addition to the features involves in *CARL-*F_4 approach, we have added two new values: one for indicating whether or not the component inherits from an SDK defined View element and one for the Controller case. For the View element, value 750 was used if the component inherited from the View component defined in the SDK and 0 otherwise ($Inh_{SDK}(c_i, View)$ was 750 or 0). The second value, for the Controller element, was 2.000 if there was an inheritance and 0 otherwise ($Inh_{SDK}(c_i, Controller)$ was 2.000 or 0). We call this version of our approach *CARL-*F_5. The new information further improved the performance of our clustering algorithm by over 12%, reaching an accuracy **85.25%** (see Table 2) on the benchmark. The View layer seems to be entirely and correctly detected, but some components that should belong to the Controller layer are associated with the Model layer. This result could be a consequence of the usage of keywords and SDK's inheritance relations dedicated to View and Controller clusters, only. A clusterization improvement is possible to be obtained by extending the set of keywords (by looking at the coding conventions and standards used in the code – manually or through an automated process).

Table 2 presents our findings on the benchmark application in the five feature selection scenarios.

4.5 Empirical Evaluation

After the experiments were run we analyzed the data and answered the research questions based on the results obtained. This subsection presents our findings.

RQ1 - What features from the codebase can be used in a clustering algorithm? In section 4.4 we have seen five different *CARL* approaches that gradually used more features extracted from the codebase.

We were interested in detecting the architectural layers of the codebase. Therefore, firstly we looked at the dependencies between the components as structural features that can be extracted from the codebase. Based on the number of dependencies we had two approaches, one where we used the exact number of dependencies between the components and one where we applied a binary encoding, highlighting the dependency's presence only.

Another information, a lexical one this time, we used was the name of the files and we have discovered similar components have similar names. We measured the similarity using the Levenshtein distance. In addition, some keywords were also used for the View and Controller layer where we have observed that developers follow a naming convention. Furthermore, the SDK inheritance information provided to the clustering algorithm further improved the accuracy. These last characteristics enrich the dependency-based structural features extracted from the codebase by semantic information and emphasize the relationship of a component with the used SDK.

To sum it up, we have used the next information:

Table 2: Analysis of all the five versions of *CARL* on the benchmark application.

	Model		View		Controller		Accu-
	Precision	Recall	Precision	Recall	Precision	Recall	racy
CARL-F$_1$	0.50	0.01	0.22	1,00	1,00	0.10	0.24
CARL-F$_2$	0.49	0.93	0.17	0.09	1,00	0.08	0.46
CARL-F$_3$	0.62	0.75	0.33	0.53	0.65	0.22	0.52
CARL-F$_4$	0.70	0.93	0.84	0.83	0.99	0.56	0.78
CARL-F$_5$	0.76	0.99	1,00	1,00	0.99	0.57	0.85

- dependencies between components – the actual number and a binary version;
- distance between the names of the files – for identifying groups of elements that share similarities between names;
- keywords – whether or not the name of the component contained certain keywords;
- SDK inheritance – whether or not the component inherits from an SDK defined one.

RQ2 - How effective is the proposed categorization method compared to manual inspections? Using *CARL-F*$_5$ approach we have obtained an average accuracy of **77,95%**, an average precision of **79,32%** and an average recall of **75,93%** on all the analyzed codebases (see Table 3 that presents the results obtained by applying *CARL-F*$_5$ on all the analysed codebases). We have observed that on one of the most complex and largest projects — Wikipedia — we have obtained an accuracy of **82,40%**. In the case of the worst-performing codebases analyzed with our method we have found out that the elements did not have a consistent naming convention. They did not contain many elements that had similarities between names or contained one of the used keywords. Our method works better in cases where the codebase is consistent with respect to naming conventions for each architectural layer and is greatly impacted by the coding standards and the consistencies of the project.

RQ3 - What are the downsides of applying clustering to a mobile codebase for detecting architectural layers? One of the most important downsides is the fact that by using the proposed approach the elements are clustered based on their dependencies which means that a component can be wrongly detected if it was not implemented properly and has forbidden or wrong dependencies.

Moreover our method also used the naming conventions and assumes the presence of certain keywords. While the keywords can be adapted based on the specification of the project, this process will become a more complex and less automated one. In addition to this, if the codebase does not respect certain naming conventions the proposed approach will also not yield good results.

Another downside of the proposed approach is the fact that the clusters have to be manually inspected for deciding what type of layer do they represent at least in the case of more specialised architectures with multiple layers. In the case of our study we have labelled the clusters based on the type of the majority of the elements, but in case of large projects with more architectural layers this process might not be automated and manual intervention might be needed.

5 THREATS TO VALIDITY

After the analysis we have found out that *CARL* presents the following threats of validity:

- **Internal –** we used the Levenshtein distance to compute the string distance metrics. We have discovered from trial and error experiments the predefined scores for the *Keywords presence* and *SDK inheritance* methods. In addition, the choice of clustering algorithm was made purely based on trial and error experiments; there might be other approaches that use different mechanisms that work better.
- **External –** the experiments were run on the iOS platform and on the Swift language, there might be other SDKs and languages which have particularities which we have not addressed in this paper. Moreover, we have focused this preliminary research only on the MVC pattern without taking much into consideration more complex architectural patterns and their particularities.
- **Conclusion –** from a results point of view, the ground truth was constructed by a human experts which might introduce biased based on their experience. Furthermore, the analyzed codebases might also be responsible for some bias and more experiments should be run.

6 CONCLUSION & FURTHER WORK

With our study, we have proven that there is potential in using AI techniques in the domain of software architectures on mobile devices. The proposed *CARL*

Table 3: *CARL-F_5* results in term of detection quality.

Codebase	Model		View		Controller		Accu-
	Precision	Recall	Precision	Recall	Precision	Recall	racy
Wikipedia	0.78	0.83	1.00	0.54	0.83	0.98	0.82
Trust	0.79	0.69	0.38	0.66	0.62	0.57	0.66
E-comm	0.76	0.99	1.00	1.00	0.99	0.57	0.85

method works well given the fact that is an unsupervised method that needs to cluster components for which is hard to define numerical metrics that feed the clustering algorithm.

This method can not be used as a standalone technique for identifying architectural issues as it does not take into consideration the constraints of the codebase, as it is able to split the codebase based on the various information obtained from its components but it cannot decide whether a component should actually be in a certain layer or not.

With the previous idea in mind, we are planning to develop a hybrid system in which we'll use a heuristic approach combined with the clustering for trying to further improve the system and to be able to detect more precisely the components of the codebase.

The proposed approach has a large variety of practical applications, it can be used for improving the architectural qualities of codebases or for educational purposes where beginners could learn to better structure their code by enforcing them to follow a certain architectural pattern.

REFERENCES

Anquetil, N. and Lethbridge, T. C. (1999). Recovering soft. architecture from the names of source files. *Journal of Soft. Maintenance: Research and Practice*, 11(3):201–221.

Belle, A. B., El-Boussaidi, G., Desrosiers, C., and Mili, H. (2013). The layered architecture revisited: Is it an optimization problem? *SEKE*, pages 344–349.

Belle, A. B., El Boussaidi, G., and Mili, H. (2014). Recovering soft. layers from object oriented systems. *2014 9th Int. Conf. on Evaluation of Novel Approaches to Soft. Engineering (ENASE)*, pages 1–12.

Constantinou, E., Kakarontzas, G., and Stamelos, I. (2011). Towards open source soft. system architecture recovery using design metrics. *2011 15th Panhellenic Conf. on Informatics*, pages 166–170.

Daoudi, A., ElBoussaidi, G., Moha, N., and Kpodjedo, S. (2019). An exploratory study of MVC-based architectural patterns in android apps. *Proc. of the 34th ACM/SIGAPP Symposium on Applied Computing*, pages 1711–1720.

Fawcett, T. (2006). An introduction to roc analysis. *Pattern recognition letters*, 27(8):861–874.

Garcia, J., Ivkovic, I., and Medvidovic, N. (2013). A comparative analysis of soft. architecture recovery techniques. *Proc. of the 28th IEEE/ACM Int. Conf. on Automated Soft. Engineering*, pages 486–496.

Kuhn, A., Ducasse, S., and Gírba, T. (2007). Semantic clustering: Identifying topics in source code. *Information and Soft. Technology*, 49(3):230–243.

Le, D. M., Behnamghader, P., Garcia, J., Link, D., Shahbazian, A., and Medvidovic, N. (2015). An empirical study of architectural change in open-source soft. systems. *MSR, 2015 IEEE/ACM 12th Working Conf. on*, pages 235–245.

Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics doklady*, 10(8):707–710.

Mancoridis, S., Mitchell, B. S., Chen, Y., and Gansner, E. R. (1999). Bunch: A clustering tool for the recovery and maintenance of soft. system structures. *Proc. IEEE Int. Conf. on Soft. Maintenance-1999 (ICSM'99).'Soft. Maintenance for Business Change'(Cat. No. 99CB36360)*, pages 50–59.

Mitchell, B. S. and Mancoridis, S. (2008). On the evaluation of the bunch search-based soft. modularization algorithm. *Soft Computing*, 12(1):77–93.

Müller, H. A., Orgun, M. A., Tilley, S. R., and Uhl, J. S. (1993). A reverse-engineering approach to subsystem structure identification. *Journal of Soft. Maintenance: Research and Practice*, 5(4):181–204.

Pfitzner, D., Leibbrandt, R., and Powers, D. (2009). Characterization and evaluation of similarity measures for pairs of clusterings. *Knowledge and Information Systems*, 19(3):361.

Russell, S. J. and Norvig, P. (2016). *Artificial intelligence: a modern approach.* Malaysia; Pearson Education Limited,.

Saeidi, A. M., Hage, J., Khadka, R., and Jansen, S. (2015). A search-based approach to multi-view clustering of soft. systems. *2015 IEEE 22nd Int. Conf. on Soft. Analysis, Evolution, and Reengineering (SANER)*, pages 429–438.

Sangal, N., Jordan, E., Sinha, V., and Jackson, D. (2005). Using dependency models to manage complex soft. architecture. *ACM Sigplan Notices*, 40(10):167–176.

Sarkar, S., Maskeri, G., and Ramachandran, S. (2009). Discovery of architectural layers and measurement of layering violations in source code. *Journal of Systems and Soft.*, 82(11):1891–1905.

Scanniello, G., D'Amico, A., D'Amico, C., and D'Amico, T. (2010). Using the kleinberg algorithm and vector space model for soft. system clustering. *2010 IEEE 18th Int. Conf. on Program Comprehension*, pages 180–189.

Trust (2018). Trust wallet iOS application. link.

Tzerpos, V. and Holt, R. C. (2000). ACCD: an algorithm for comprehension-driven clustering. *Proc. 7th Working Conf. on Reverse Engineering*, pages 258–267.

Wikimedia (2018). Wikipedia ios application. link.