Automatic Detection of Architectural Bad Smells through Semantic Representation of Code

Ilaria Pigazzini
i.pigazzini@campus.unimib.it
Department of Informatics, Systems and Communication, University of Milano - Bicocca
Milano, Italy

ABSTRACT

Bad design decisions in software development can progressively affect the internal quality of a software system, causing architecture erosion. Such bad decisions are called *Architectural Smells (AS)* and should be detected as soon as possible, because their presence heavily hinders the maintainability and evolvability of the software. Many detection approaches rely on software analysis techniques which inspect the structure of the system under analysis and check with rules the presence of AS. However, some recent approaches leverage natural language processing techniques to recover *semantic* information from the system. This kind of information is useful to detect AS which violate "conceptual" design principles, such as the *separation of concerns* one. In this research study, I propose two detection strategies for AS detection based on *code2vec*, a neural model which is able to predict semantic properties of given snippets of code.

KEYWORDS

Architectural (bad) smells detection, code embeddings, architecture erosion, software concerns

ACM Reference Format:
Ilaria Pigazzini. 2019. Automatic Detection of Architectural Bad Smells through Semantic Representation of Code. In *European Conference on Software Architecture (ECSA), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3344948.3344951

1 INTRODUCTION

A challenging problem in software architecture maintainance is the recovery of the architecture and the identification of architectural anomalies. In particular, architectural smells [9] (also known as *architectural bad smells*) have been proven to be a source of architecture erosion, that is the progressive loss of structural integrity in software systems due to the violation of design principles [5]. The presence of smells in the architecture undermines software qualities such as evolvability and maintanability of the system, hence it is crucial to identify and remove them [8]. During my PhD I aim to study and develop innovative techniques for the automatic detection of AS, in addition to analyze the evolution of AS, their prediction and their impact on several quality issues. During the last year I worked on the the detection of architectural smells which can support the migration from monolithic to microservices architecture [14]. Current approaches for architectural smell detection usually exploit metrics evaluation, graph algorithms and Design Structure Matrix (DSM) algorithms[16][19][7]. Such approaches, combined with the definition of rules and thresholds, are used to obtain information about the structure of the software system under analysis. On the other hand, they do not give information about the semantic role of the different software components of the system, making it hard to detect some architectural smells which, for example, violate the separation of concerns principle [13]. In this research context, a *concern* is a software system's role, responsibility, concept, or purpose [18]. An approach able to extract semantic properties from software is the exploitation of Natural Language Processing (NLP) techniques and models applied to code. An example is *code2vec*, a neural model for representing snippets of code as countinuos distributed vectors (*code embeddings*) [1]. The code embeddings approach allows to associate a continuous distributed vector to a piece of code and compute the semantic similarity between different pieces of code. This allows to have semantic information bounded to specific parts of code and to perform different tasks. For instance, to understand how a specific concern of the system spreads through code and consequently identify anomalies such as architectural smells.

The aim of this study is to exploit the *code2vec* neural model, able to generate code embeddings, in order to produce a semantic representation of the source code and enhance the detection of architectural smells. The code2vec model has not been exploited for software analysis yet. The novelty of the proposed approach relies in the experimentation of a neural model to perform architectural analysis and the proposal of new detection techniques for architectural smells. As first step in this direction, the following research questions must be posed:

- *RQ1: "Is it possible to exploit the code2vec model to identify architectural smell in a project?"* In particular I am interested in understanding if code vectors can be used to detect architectural smells which regard how concerns are distributed in the software architecture.
- *RQ2: "Is it possible to represent software concerns with the code2vec model?"* In order to answer RQ2, I have to investigate whether the distributed code vector space is able to represent the semantic properties of the software architecture. Code2vec takes as input generic snippets of code. However, in order to simplify the problem and exploit the original formulation of the model and its provided implementation, I start from the investigation of how Code2vec represents *methods* in Object Oriented (OO) code. In the future, I aim to

ECSA, September 9–13, 2019, Paris, France

ACM ISBN 978-1-4503-7142-1/19/09…$15.00
https://doi.org/10.1145/3344948.3344951

extend the approach by considering snippets of class code. Hence, the investigation is driven by two further research questions:

- *RQ2.1 "Is there a relationship between the value of code vector similarity and method dependencies?"* The aim of this question is to study whether similarity values follow a specific probability distribution when considering couples of methods linked by a dependency.
- *RQ2.2 "Can vector similarity measure the cohesion among methods?"* The answer to this question aims to understand if vector similarity helps in measuring how much two methods are involved in the same concern.

The research study will include the implementation of the approach and its validation. The AS detection will be integrated in our existing tool Arcan[1] [7], previously developed for the detection of three AS on Java projects.

2 RELATED WORK

Exploiting Natural Language Processing (NLP) and Machine Learning (ML) techniques to extract semantic information from software gained popularity in the last 10 years. In this Section I first introduce some examples of models leveraging software semantic for software recovery and then I focus on the approaches aiming to detect AS. Jalali et al. [15] propose a multi-objective fitness function, named MOF, which exploits both structural and semantic features (such as semantic contained in the code comments and identifier names), to automatically guide optimization algorithms to find a good decomposition of software systems. Boaye et al. [2] propose a search-based approach that uses structural and lexical information to recover the layered architecture, at the package level, of an Object Oriented (OO) system. They assume that two packages are conceptually related if the packages' lexical information is similar. The authors model software concerns as topics extracted by using LDA: the conceptual relationship between two packages is computed as the cosine similarity between their corresponding topic proportion vectors. Corazza et al. [3] consider the source code of OO software divided in six "zones" depending on the granularity level (class name, method name, attribute name, parameter name, comment, source code statement) and try to assign to those zones an estimation of their relevance based on the contained lexical information to improve the quality of software recovery. They define a probabilistic model of the lexemes distribution and then exploit it to compute similarities among source code classes, which are then grouped by a k-Medoid clustering algorithm.

The presented models and methods aim to recover software architecture by considering semantic features of software itself. The following works go further by investigating the usage of semantic information to detect AS in OO software. Diaz-Pace et al. [6] explore whether social network analysis is useful to extract information from a software architecture in order to predict new dependencies and possible future appearance of AS in Java projects. They model software architecture as a dependency graph where edges are enriched with topological and content-based information modeled as *bag-of-words*. Then, they compute edge information as the Cosine Similarity scores for the different bag-of-words representations. Garcia et al [10] propose to recover software systems thanks to the detection of software concerns. To obtain concerns, they exploit a statistical language model named Latent Dirichlet Allocation (LDA). Their approach is implemented in a tool named Arcade, which is also able to detect Scattered Functionality and Feature Concentration in Java projects [12]. Their detection approach is based on the number of topics assigned to each software component.

All the above mentioned approaches do not consider context information of the analyzed code, moreover they were not designed for source code analysis. Regarding this aspect, a study from Hellendoorn et al. [11] suggested that existing neural networks are not the best solution to represent code semantic, but in my work I exploit a neural model specifically developed for code analysis. Hence, I aim to explore this model to enhance the current methodologies and investigate whether the embeddings approach outperforms the existing ones.

3 DESCRIPTION OF THE APPROACH

The aim of this project is to investigate how concerns spread through the architecture and to detect architectural smells by leveraging code embeddings representation. In this section, I propose a study to understand whether the vector representation of methods in Object Oriented (OO) projects provides useful semantic information. Then, I introduce the algorithms to identify Scattered Functionality and Feature Concentration, two examples of AS which violate the separation of concerns principle (see Section 3.2).

Both the two detection strategies leverage the graph representation of the structural dependencies in the project under analysis (*dependency graph*) combined with the semantic information extracted by the code2vec model. In particular, I exploit the ability of the model to generate vectors representing the semantics of the input methods. Hence, given an OO project, the architecture of the project is represented as $G(V, E)$, where V is the set of nodes comprehending methods, classes and packages of the project and E is the set of dependencies among them. Moreover, given the set of methods $M \subset V$, the code2vec model creates a vector space where each point corresponds to a method.

3.1 Analysis of the vector representation

The first part of the research focuses on understanding whether code embeddings are suitable for the representation of concerns inside software architecture. In particular, I am interested in investigating if vector similarity is a proper metric to quantify the semantic dependency among different methods in OO projects. The vector similarity considered in this study is the *cosine similarity*, which measures the cosine of the angle θ between two non-zero vectors $\vec{v}$ and $\vec{v}$.

$$similarity(\vec{v}, \vec{w}) = \cos\theta = \frac{\vec{v} \cdot \vec{w}}{\mid \vec{v} \mid\mid \vec{w} \mid} \quad (1)$$

This metric is non-negative and bounded between [0, 1]. I chose this metric since it is commonly used in NLP to measure document/text distance, but in the future I plan to experiment different distance metrics and compare their results.

The analysis consists of the following steps:

- Generation of the code vectors associated to the methods of an OO project with code2vec.

[1] http://essere.disco.unimib.it/wiki/

- Computation of the vector similarity of all methods of the OO project. The Cosine Similarity metric is computed for each pair of vectors without taking into account the order, hence the total number of combinations is

$$\sum_{i=0}^{n-1} n - i = \frac{n(n+1)}{2} \tag{2}$$

  where n is the number of vectors.
- Compare the similarity distribution of *virtual edges* and *concrete edges*. For virtual edge I mean the possible edge that can link two methods; since similarity is computed for every couple of methods, there is a similarity value for each virtual edge. Instead, a concrete edge is an actual dependency in the project.

The aim of the proposed analysis is to understand if similarity follows a particular distribution and to investigate if the distribution changes when considering couples of methods which actually are linked one to another. If proved on a meaningful number of OO projects, a non-random similarity distribution could be the first sign of significance of the similarity metric.

3.2 Architectural Smells detection

The second part of the research aims to exploit the generated vector space to detect architectural smells in the project under analysis. Currently, I propose the detection strategies for Scattered Functionality and Feature Concentration smells. Both smells regards how software concerns are implemented in the software architecture: a good design should follow the separation of concern principle, where each architectural component addresses a separate concern. When concerns are well-separated, individual parts of architecture can be reused, as well as developed and updated independently [13]. In particular Scattered Functionality affects systems where multiple components address the same concern and Feature Concentration is the contrary: affects a single component which addresses multiple concerns. Following, the definitions of the two smells:

- *Scattered Functionality (SF)* describes a system where multiple components are responsible for realizing the same high-level concern and, additionally, some of those components are responsible for orthogonal concerns[8]. In this context, I detect the smell on packages.
- *Feature Concentration (FC)* occurs when an architectural entity implements different functionalities in a single design construct [4]. Detected on packages.

In the following paragraphs, I propose the two detection strategies for the AS.

Scattered Functionality. The detection of Scattered Functionality smell proposed in this work exploits the similarity metric to build paths on the dependency graph which represent software concerns. The paths consist of a set of edges, e_t^*, which have the property of maximizing the similarity value between code vector m_z and the sum of the two antecedent code vectors m_x and m_j. The formal definition of *concern path* is the following:

Let $M \subset V$ be the set of methods m_i of the project. Let $E_m \subset E$ be the set of method dependencies. Let $G(M, E_m)$ the induced directed graph of $G(V, E)$ having for nodes M and for edges E_m. Let $M_{src} \subset M$ be the set of source vertices of the graph. Let $M_{snk} \subset M$ be the set of sink vertices of the graph.

- For each $m_i \in M$, compute the associated code vector.
- For each m_{src}, for each $y_{src} \in neigh(m_{src})$

$$\begin{aligned} e_0^* &= m_{src} \rightarrow y_{src} \\ p^* = \{e_t^* \mid\ & e_t^* = y \rightarrow z, \\ & \arg\max_{m_z} similarity(m_x + m_y, m_z), \\ & m_x = out(e_{t-1}^*), \\ & m_y = in(e_{t-1}^*), \\ & t \in \{1..n\}\} \end{aligned} \tag{3}$$

where n is the depth of the first sink node encountered along the concern path. Hence, the strategy to detect Scattered Functionality on a OO project consists in the following steps:

- Compute all the concern paths of the dependency graph of project
- If a computed path crosses more than a package, the involved packages are affected by the smell.

Feature Concentration. The detection strategy proposed for this smell exploits the semantic vector similarity to run a clustering algorithm. The aim of the clustering analysis is to group similar methods inside a given package: I hypothesize that the detected groups correspond to the concerns of the package. Then, if a package shows too many concerns, it may be affected by Feature Concentration smell. The proposed detection strategy consists in:

- For each package, for each couple of methods $m_i \in c_i, m_j \in m_j, c_i \neq c_j$, compute $similarity(m_i, m_j)$.
- Run a clustering algorithm to identify the groups of similar methods
- If the number of detected groups and the similarity distance among them is high, the package is affected by Feature Concentration.

The choice of a suitable clustering algorithm and the definition of thresholds will be a matter of future research during my PhD.

4 EVALUATION PLAN AND FIRST RESULTS

To assess the applicability of my approach, I plan to conduct a set of validations, both in Open Source and industrial setting.

In order to answer RQ1:*"Is it possible to exploit the code2vec model to identify architectural smell in the software architecture?"* I plan to implement the proposed approaches in our tool Arcan for architectural smell detection [7] and subsequently:

- Analyze a set of Open Source Java projects, for instance from the Qualitas Corpus [17], and validate the results by inspecting the publicly available information about the projects. In particular, by collecting issues from issue repositories such as Jira platform[2], which allows to perform queries on the issues of a given project and understand whether a software component (class/package) is involved in an issue of type *bug*, *improvement* or *new feature*. This validation method has already been used [12] and has the advantage to be replicable and objective with respect to manual validation.

[2]https://www.atlassian.com/software/jira

- Analyze a set of industrial projects and ask developers for expert validation.
- Compare the results with the detection offered by other approaches/tools, for instance by comparing with the concerns extracted with LDA model [12].

In order to answer RQ2: *"Is it possible to represent software concerns with the code2vec model?"*

- Generate the code vector space for a set of Open Source Java projects of different application domains;
- Analyze the similarity distribution (to answer RQ2.1).
- Run the clustering algorithm and manually check the quality of the detected clusters (to answer RQ2.2).

As the first step in my study, I started researching the answer to RQ2. Figure 1 and 2 show the similarity distributions of project Checkstyle v5.6 [17], computed on the 1306 methods belonging to package *checks*. The similarity values shown in Figure 1 were computed on all the possible combinations of different methods (*virtual edges*), while the ones in Figure 1 were computed on the couples of methods actually linked (*concrete edges*). The former are bounded in $[-0.3620, 1]$, while the latter in $[-0.1994, 0.9873]$: this means that, for Checkstyle, the set of possible values which model concrete edges among methods is smaller with respect to the one which model all possible edges. Such finding encourages further analysis in order to understand the significance of similarity. Moreover, the virtual edges distribution shows anomalous spikes for certain similarity values, which may be caused by possible approximations of the algorithm used during the similarity computation. In the future, I plan to investigate in particular the data with similarity equal to 1. Concerning the concrete edges distribution, the plot shows a concentration of values around 0.1 and 0.25. I tried to fit it as a mixture of two normal density distributions using the Expectation Maximization algorithm and it resulted that the data could belong to two different distributions. I plan to further investigate these results and validate the statistical methods in the near future.

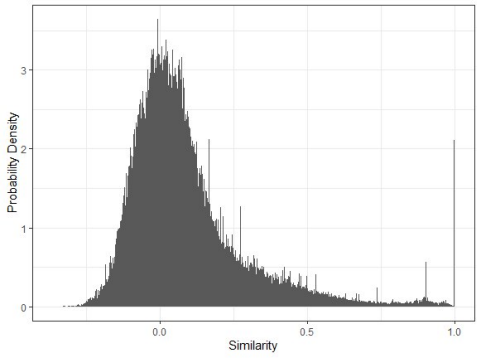

Figure 1: Checkstyle similarity - virtual edges

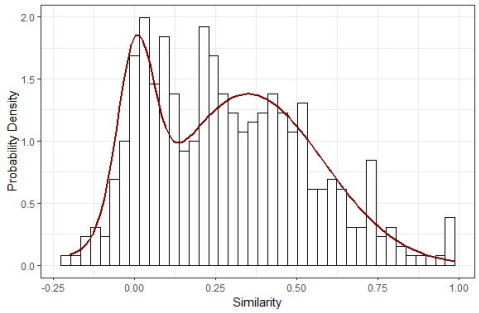

Figure 2: Checkstyle similarity - concrete edges

5 SUMMARY

With this study I propose two new strategies to detect concern-based architectural smells which rely on the semantic representation of code provided by the code2vec model. As part of my research work, I plan to implement such strategies in Arcan, our tool for AS detection and to validate my approach on Open Source and industrial projects.

ACKNOWLEDGMENTS

Prof. Francesca Arcelli Fontana supervises my PhD work.

REFERENCES

[1] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. Code2Vec: Learning Distributed Representations of Code. *Proc. ACM Program. Lang.* (2019).
[2] Alvine Boaye Belle, Ghizlane El Boussaidi, and Sègla Kpodjedo. 2016. Combining Lexical and Structural Information to Reconstruct Software Layers. *Inf. Softw. Technol.* 74, C (June 2016), 1–16. https://doi.org/10.1016/j.infsof.2016.01.008
[3] Anna Corazza, Sergio Martino, Valerio Maggio, and Giuseppe Scanniello. 2016. Weighing Lexical Information for Software Clustering in the Context of Architecture Recovery. *Empirical Softw. Engg.* 21, 1 (Feb. 2016), 72–103.
[4] Hugo Sica de Andrade, Eduardo Almeida, and Ivica Crnkovic. 2014. Architectural Bad Smells in Software Product Lines: An Exploratory Study. In *Proc. of the WICSA 2014 Companion Volume*. ACM.
[5] Lakshitha de Silva and Dharini Balasubramaniam. 2012. Controlling software architecture erosion: A survey. *Journal of Systems and Software* 85, 1 (2012).
[6] J.A. Díaz-Pace, A. Tommasel, and D. Godoy. [n. d.]. Towards Anticipation of Architectural Smells Using Link Prediction Techniques. In *18th IEEE SCAM, 2018.*
[7] Francesca Arcelli Fontana, Ilaria Pigazzini, Riccardo Roveda, Damian Andrew Tamburri, Marco Zanoni, and Elisabetta Di Nitto. 2017. Arcan: A Tool for Architectural Smells Detection. In *ICSA Workshops 2017.*
[8] Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. 2009. Identifying Architectural Bad Smells. In *CSMR 2009*. IEEE, 255–258.
[9] Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. 2009. Toward a Catalogue of Architectural Bad Smells. In *Proc. 5th Int'l Conf. the Quality of Software Architectures (QoSA 2009)*. Springer Berlin Heidelberg.
[10] Joshua Garcia, Daniel Popescu, Chris Mattmann, Nenad Medvidovic, and Yuanfang Cai. 2011. Enhancing Architectural Recovery Using Concerns. In *Proc. of ASE 2011*. IEEE Computer Society, Washington, DC, USA, 552–555.
[11] Vincent J. Hellendoorn and Premkumar Devanbu. 2017. Are Deep Neural Networks the Best Choice for Modeling Source Code?. In *Proc. of ESEC/FSE 2017.*
[12] Duc Minh Le, Daniel Link, Arman Shahbazian, and Nenad Medvidovic. 2018. An Empirical Study of Architectural Decay in Open-Source Software. In *ICSA 2018.*
[13] Martin Lippert and Stephen Roock. 2006. *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully.* Wiley. 286 pages.
[14] Ilaria Pigazzini, Francesca Arcelli Fontana, and Andrea Maggioni. 2019. Tool support for the migration to microservice architecture: an industrial case study. In *To appear in Proc. of the European Conf. on Software Architecture (ECSA 2019).*
[15] Nafiseh Sadat Jalali, Habib Izadkhah, and Shahriar Lotfi. 2018. Multi-objective search-based software modularization: structural and non-structural features. *Soft Computing* (11 2018). https://doi.org/10.1007/s00500-018-3666-z
[16] Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. 2014. *Refactoring for Software Design Smells: Managing Technical Debt* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
[17] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. 2010. The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. In *Proc. 17th Asia Pacific Software Eng. Conference (APSEC 2010)*. IEEE, Sydney, Australia, 336–345.
[18] Edsger W. Dijkstra. 1974. On the Role Of Scientific Thought. (01 1974).
[19] Lu Xiao, Yuanfang Cai, and Rick Kazman. 2014. Titan: A Toolset That Connects Software Architecture with Quality Analysis. In *Proc. of FSE 2014.* ACM.