Software Clusterings with Vector Semantics and the Call Graph*

Marios Papachristou†
papachristoumarios@gmail.com
National Technical University of Athens
Athens, Attica, Greece

ABSTRACT

In this paper, we propose a novel method to determine a software's modules without knowledge of its architectural structure, and empirically validate the method's performance. We cluster files by combining document embeddings, generated with the *Doc2Vec* algorithm, and the *call graph*, provided by Static Graph Analyzers to an augmented graph. We use the *Louvain Algorithm* to determine its community structure and propose a module-level clustering. Our method performs better in terms of stability, authoritativeness , and extremity over other state-of-the-art clustering methods proposed in the literature and is able to decently recover the ground truth clustering of the Linux Kernel. Finally, we conclude that semantic information from vector semantics as well as the call graph can produce accurate results for software clusterings of large systems.

CCS CONCEPTS

• **Computing methodologies** → **Cluster analysis**; **Machine learning algorithms**; • **Software and its engineering** → **Software architectures**.

KEYWORDS

document embeddings, doc2vec, linux kernel, natural language processing, software architecture recovery, software clustering, static graph analysis, vector semantics

ACM Reference Format:
Marios Papachristou. 2019. Software Clusterings with Vector Semantics and the Call Graph. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3338906.3342483

1 INTRODUCTION

In modern software systems, the need for adoption of modular software architectural schemata becomes inevitable for robust maintainability of their codebases. Since the architecture of a software system is the most fundamental realization of the software system, when there is no description of the architecture of it, software architects make attempts to recover it. In the past, methods have been proposed to identify the main components of a software system using hierarchical clustering algorithms [1, 9] mainly by clustering modules into subsystems, given using the source code. Recently, the possibilities offered by machine learning techniques and practices in the Natural Language Processing (NLP) can also be applied on codebases as corpora of text, thus enabling the extraction of useful information for the semantics of the source code inaugurating new modes of reviewing the codebase.

Our motivation, through this paper, is to use new features, namely vector semantics, and structural information via the call graph in order to perform community detection and arrive at module-level clusterings. We will study the Linux codebase and we will compare it with ground truth using the MoJo measure [15] as well as compare it with other state-of-the-art and baseline clustering algorithms in terms of *stability, authoritativeness* and *extremity* [9, 19].

2 RELATED WORK

Onaiza Maqbool and Haroon A. Babri [9] present an overview of the various approaches taken towards hierarchical clustering algorithms for software systems. In their study, they compare many clustering algorithms such as ACDC, LIMBO and other Traditional Clustering Algorithms such as Single Linkage, Complete Linkage and Weighted Linkage presenting their efficiency regarding multiple feature sets. Added to this, a *semantics-based architectural view* of the system, as discussed in reference [2] reveals significant aspects of a software system and its change over time, which suggests that semantic-based approaches should be followed for gaining better understandings of a software system.

Work in Software Clustering Algorithms (SCA) has been extensively done by many authors in references [15], [17], [16] and [1] with the ACDC and LIMBO clustering algorithms. In particular, the ACDC algorithm leans toward to software components comprehension based on subsystem patterns. Their approach considers an initial structure of the system, without taking into account semantics, and tries to build comprehensive clusterings of the given ground structure. The authors also conducted a study on an older version of Linux. Moreover, Andritsos and Tzerpos in reference [1] present an Information-Theoretic approach of SCA by developing LIMBO which clusters modules upon inserting their Distributional Cluster Features to a B+-tree variant and then applying the *Agglomerative Information Bottleneck* algorithm.

3 METHOD

The codebase is initially preprocessed and the resulting data is fed into a Skip-Gram model for the initial components, which correspond to nodes in the call graph. Next, we add weights between the

*The research described has been carried out as part of the CROSSMINER Project, which has received funding from the European Union's Horizon 2020 Research and Innovation Programme under grant agreement No. 732223
†Also with Business Analytics Lab at Athens University of Economics and Business.

Table 1: Experimental Results for Linux 4.21

Algorithm	Feature Dimension	# Clusters	Size Range	$\bar{x}$	σ	Median	MoJo Distance
ACDC [16]	–	9055	1 – 4245	5	46	2	33694
Average Linkage [13]	*300*	*21*	1–3406	163	725	1	2092
Complete Linkage [4]	*300*	*21*	1–1529	163	412	19	1710
LIMBO ($B = 100,\ S = \infty$) [1]	12317	*21*	50–1810	163	375	50	1482
Ward Linkage [18]	*300*	*21*	21–948	163	223	70	1138
SADE	*300*	10 (±2)	2 (±0) – 132(±13)	64 (±4)	40 (±4)	65 (±10)	243 (±1)
SADE (Directed)	*300*	5 (±2)	1 (±1) – 614(±1)	141 (±39)	253 (±25)	2 (±0.3)	237 (±2)
Ground Truth	–	21	1–1348	163	341	11.0	–

nodes with the normalized similarity $w(i, j) = \left(1 + \cos(\mathbf{d}_i, \mathbf{d}_j)\right)/2$ between the embeddings. Finally, we run the Louvain algorithm to obtain a clustering. The overall process is described below.

First of all, we tokenize the source code and remove the stop-words. After that, for each token, we split it into its constituent parts using dynamic programming [5] and lemmatize each individual sub-token via using the NLP package spaCy [7]. For example, the method `__zone_seqlock_init` corresponds to `zone`, `seqlock` and `init`, `inprogress` is split into `in` and `progress` and `literals` becomes `literal`. We, then, train a Doc2Vec model using Gensim [12]. The codebase is also processed by a Static Graph Analyzer. In our approach, focusing on C projects, we use *CScout* [14] in order to generate the directed call graph through function calls between the files. Each source code file is assigned to a module. Modules can be user-defined or automatically generated by their respective directories at a desired directory tree depth. The input to the clustering procedure is the call graph with normalized cosine similarities of the modules as edge weights. Then we run the Louvain Clustering Algorithm [3] in order to obtain the software clustering via maximizing the modularity function with a greedy approach. In case we want to consider edge directionality, we do a bipartite transformation, detect communities and merge the results with a union-find data structure as stated in reference [8].

Our method is implemented in the Python Language using spaCy[7], Gensim [12] and NetworkX [6] and is named SADE. The source code and data are available in [11] and [10].

We have chosen to use Linux as a codebase to evaluate our method on since it is a large and complex system with 27 years of continuous development and spanning 20.3 millions source lines of code (SLOC) at the time of writing. Moreover, it is easy to establish a ground truth due to its clear structure and construct test-cases for evaluations. Finally, it was also used in related studies [16, 19] as a reference system for evaluation.

Our method was tested on Linux 4.21, consisting of 20.3 million SLOC against Average-Linkage [13], Complete-Linkage [4] and Ward-Linkage [18] using the same document embeddings as well as ACDC with structural information [16] and LIMBO [1] with Bag-of-Words features. The results are presented in Table 1 and user-defined parameters are formatted in italics. As ground truth, we have used the first level directories as a target clustering and as input, we have considered the modules of the one-top directories. For example, the source code file `drivers/net/ieee802154/mcr20a.c` has a ground truth value of `drivers` and it is considered under the same module as every `.c` and `.h` file under `drivers/net/ieee802154`. Since Louvain Community Detection produces different, but very similar, results from every time it runs, we ran the simulations multiple times and averaged the results. The experimentation with a large and complex system may constitute a threat to our findings' validity since the results are not general.

4 RESULTS, EVALUATION & DISCUSSION

In Table 1, we present the results of the clusterings ordered in increasing MoJo distance with respect to the ground truth. We have included the number of clusters produced, the cluster size range, the average cluster size the standard deviation in the cluster sizes, the median cluster and the MoJo distances with respect to the ground truth. The ground truth clustering is also present in the table. Our quality metrics for the clusterings are *extremity*, *authoritativeness* and *stability* as proposed in reference [19].

Our approach produces balanced clusterings thus demonstrating low *extremity* compared to the other clusterings, producing a number of clusters that is close to the ground truth with smaller standard deviation in the cluster sizes than the other clustering methods, without knowing the number of clusters *a priori*. Besides this, it gives balanced clusters with respect to *Median Cluster* criterion. Note that ACDC, which takes into account only the structural properties of the files, produced a high granularity clustering that failed to catch the structure of Linux. In the *authoritativeness* criterion, our approach outperforms the other clusterings in terms of MoJo distance being very close (smaller is better) to the ground truth, with small standard deviation, posing great *stability* in the resulting clusterings. Closing, the other clustering methods gave disappointing MoJo distances even with a fixed number of clusters.

5 CONCLUSIONS

This paper is set out to show how we can use a combination of vector semantics and information from the call graph in order to produce meaningful clusterings of a software system. We have outperformed state-of-the-art software clusterings and conventional agglomerative clustering algorithms in terms of *extremity*, *authoritativeness* and *stability* without even knowing the number of clusters of the ground truth. This evidence supports the further usage of vector semantics and the call graph for architecture recovery. Finally, the further integration with static analyzers and the development of evaluation policies with users should be taken into account, especially when dealing with old codebases lacking technical documentation.

REFERENCES

[1] Periklis Andritsos and Vassilios Tzerpos. 2005. Information-theoretic software clustering. *IEEE Transactions on Software Engineering* 2 (2005), 150–165.

[2] Pooyan Behnamghader, Duc Minh Le, Joshua Garcia, Daniel Link, Arman Shahbazian, and Nenad Medvidovic. 2017. A large-scale study of architectural evolution in open-source software systems. *Empirical Software Engineering* 22, 3 (2017), 1146–1193.

[3] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment* 2008, 10 (2008), P10008.

[4] Daniel Defays. 1977. An efficient algorithm for a complete link method. *Comput. J.* 20, 4 (1977), 364–366.

[5] Anderson Derek. 2018. wordninja. https://github.com/keredson/wordninja.

[6] Aric Hagberg, Pieter Swart, and Daniel S Chult. 2008. *Exploring network structure, dynamics, and function using NetworkX.* Technical Report. Los Alamos National Lab.(LANL), Los Alamos, NM (United States).

[7] Matthew Honnibal and Ines Montani. 2017. spacy 2: Natural language understanding with bloom embeddings, convolutional neural networks and incremental parsing. *To appear* (2017).

[8] Fragkiskos D Malliaros and Michalis Vazirgiannis. 2013. Clustering and community detection in directed networks: A survey. *Physics Reports* 533, 4 (2013), 95–142.

[9] Onaiza Maqbool and Haroon Babri. 2007. Hierarchical clustering for software architecture recovery. *IEEE Transactions on Software Engineering* 33, 11 (2007).

[10] Marios Papachristou. 2019. Linux Kernel 4.21 Call Graph. https://doi.org/10.5281/zenodo.2652487

[11] Marios Papachristou. 2019. Software Architecture with Document Embeddings and the Call Graph Source Code. https://doi.org/10.5281/zenodo.2673033

[12] Radim Rehurek and Petr Sojka. 2011. Gensim – Statistical Semantics in Python. (2011).

[13] Robert R Sokal. 1958. A statistical method for evaluating systematic relationship. *University of Kansas science bulletin* 28 (1958), 1409–1438.

[14] Diomidis Spinellis. 2010. CScout: A refactoring browser for C. *Science of Computer Programming* 75, 4 (2010), 216.

[15] Vassilios Tzerpos and Richard C Holt. 1999. MoJo: A distance metric for software clusterings. In *Reverse Engineering, 1999. Proceedings. Sixth Working Conference on.* IEEE, 187–193.

[16] Vassilios Tzerpos and Richard C Holt. 2000. ACDC: an algorithm for comprehension-driven clustering. In *Reverse Engineering, 2000. Proceedings. Seventh Working Conference on.* IEEE, 258–267.

[17] Vassilios Tzerpos and Richard C Holt. 2000. On the stability of software clustering algorithms. In *Program Comprehension, 2000. Proceedings. IWPC 2000. 8th International Workshop on.* IEEE, 211–218.

[18] Joe H Ward Jr. 1963. Hierarchical grouping to optimize an objective function. *Journal of the American statistical association* 58, 301 (1963), 236–244.

[19] Jingwei Wu, Ahmed E Hassan, and Richard C Holt. 2005. Comparison of clustering algorithms in the context of software evolution. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on.* IEEE, 525–535.