Software Clusterings with Vector Semantics and the Call Graph

Marios Papachristou
papachristoumarios@gmail.com
el15101@mail.ntua.gr

BALab, Athens University of Economics and Bussiness
National Technical University of Athens
Advisor: Prof. Diomidis Spinellis

Student Research Competition
ESEC/FSE 2019, Tallin
ACAC 2019, Athens

Problem Statement

- The architecture of a software system is the most fundamental realization of it

Problem Statement

- The architecture of a software system is the most fundamental realization of it
- When there is no specific definition of it, we can attempt to recover it

Problem Statement

- The architecture of a software system is the most fundamental realization of it
- When there is no specific definition of it, we can attempt to recover it
- One particular problem is the **clustering of its components into modules**

Problem Statement

- The architecture of a software system is the most fundamental realization of it
- When there is no specific definition of it, we can attempt to recover it
- One particular problem is the **clustering of its components into modules**
- Many methods exist in literature

Our motivation

Our motivation, through this work is to

1. Provide a method for software clusterings through **vector semantics** and the **call graph**

Our motivation

Our motivation, through this work is to

1. Provide a method for software clusterings through **vector semantics** and the **call graph**
2. Evaluate our method on the **Linux Kernel Codebase**

Our motivation

Our motivation, through this work is to

1. Provide a method for software clusterings through **vector semantics** and the **call graph**
2. Evaluate our method on the **Linux Kernel Codebase**
3. Compare it against state-of-the-art methods (ACDC [12], LIMBO [1]) and agglomerative clustering methods (agglomerative clustering [9, 4, 13])

Our approach I

We took a simple approach to the problem

1. Define the initial "grains" of the system. With the term "grains" we can refer e.g. to source files (.c), source (.c) and header (.h) files (combined) as well as one-top directory modules.
2. Preprocess the files attributed to the "grains"
3. Train a Skip-Gram model (Doc2Vec [6]) on them and obtain vector representations of the "grains" $\mathbf{x}_1, \dots, \mathbf{x}_n$
4. Generate the call graphs of the system using a static code analyzer (e.g. CScout [10])

Our approach II

5. Put weights on the graph minor $H(V, E)$ induced by the "grains" as the normalized cosine similarities between them

$$w(i,j) = \frac{1 + \cos(\mathbf{x}_i, \mathbf{x}_j)}{2} \qquad \forall (i,j) \in E(H)$$

6. Run Louvain Community Detection on H and obtain software clusterings

Preprocessing

1. First the code is tokenized and identifiers are split into their constituent parts using dynamic programming and n-grams. Stop-words are removed.

Preprocessing

1. First the code is tokenized and identifiers are split into their constituent parts using dynamic programming and n-grams. Stop-words are removed.
2. For example `zone_seqlock_init` becomes `zone`, `seqlock`, `init` and `inprogress` becomes `in` and `progress`

Preprocessing

1. First the code is tokenized and identifiers are split into their constituent parts using dynamic programming and n-grams. Stop-words are removed.
2. For example `zone_seqlock_init` becomes `zone`, `seqlock`, `init` and `inprogress` becomes `in` and `progress`
3. The resulting tokens are lemmatized using the English Lemmatizer provided by the spaCy [5] package

Embeddings

A Skip-Gram model is trained. The objective of such a model is to maximize the probability that a word appears in a window (context) of size $2k + 1$

$$\frac{1}{N} \sum_{t=k}^{N-k} \log \Pr[w_t \mid w_{t-k}, \dots, w_{t+k}]$$

where

$$\Pr[w_c \mid w_t] = \frac{\exp(s(w_c, w_t))}{\sum_{j=1}^{V} \exp(s(w_t, j))} \qquad s(w_c, w_t) = \langle \mathbf{d}_c, \mathbf{d}_t \rangle$$

We have used Doc2Vec for our training which extends the aforementioned idea to extract document embeddings.

The Linux Kernel Codebase

- A **HUGE** codebase consisting of ∼20.3 million lines of source code

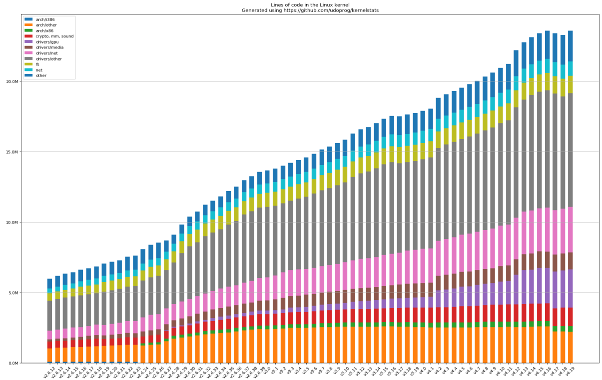

Figure: Linux Kernel Codebase Size over time. Source: Reddit

- Constantly growing
- Easy-to-find ground truth for evaluation

Call Graphs I

The call graphs were extracted with CScout [10] and are of the following forms

1. Macro and Function Call Graph
2. Control Dependency Graph
3. File include Graph
4. Compile-time Dependency Graph
5. Data dependency Graph (through globals)

The extraction of the call graphs took $\sim$ 10h and required $\sim$ 32GB of RAM on a Debian server.

Call Graphs II

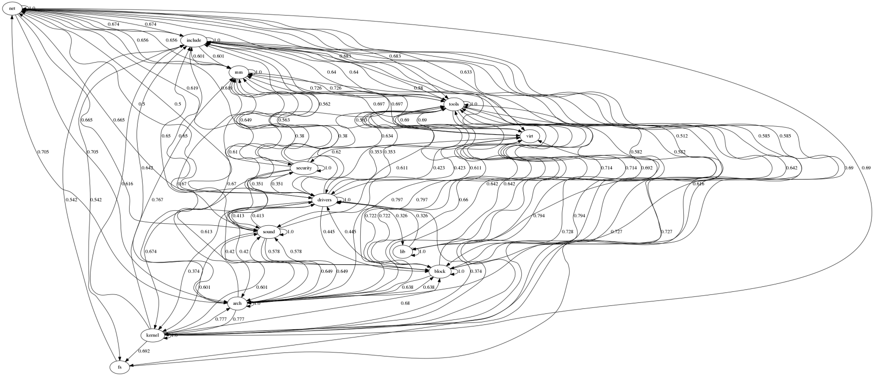

Figure: Call Graph Example between Kernel one-level directories

Preparing the graph for clustering

- The weights assigned to every edge are the normalized cosine similarities

$$\cos(\mathbf{x}_i, \mathbf{x}_j) = \frac{\langle \mathbf{x}_i, \mathbf{x}_j \rangle}{\|\mathbf{x}_i\| \|\mathbf{x}_j\|}$$

$$w(i, j) = \frac{1 + \cos(\mathbf{x}_i, \mathbf{x}_j)}{2} \qquad \forall (i, j) \in E(H)$$

Preparing the graph for clustering

- The weights assigned to every edge are the normalized cosine similarities

$$\cos(\mathbf{x}_i, \mathbf{x}_j) = \frac{\langle \mathbf{x}_i, \mathbf{x}_j \rangle}{\|\mathbf{x}_i\| \|\mathbf{x}_j\|}$$

$$w(i, j) = \frac{1 + \cos(\mathbf{x}_i, \mathbf{x}_j)}{2} \qquad \forall (i, j) \in E(H)$$

- Experiments were run using both the directed and the undirected version of the graph. The directed version of the graph required doing a bipartite transformation [7] where every edge (i, j) was mapped to $\{i, j'\}$ where j' was a copy of $j \in V$. After community detection, the communities which j and j' belonged to were merged using a union-find data structure.

Louvain Community Detection I

- The Louvain method for community detection aims to produce communities which maximize the modularity function

$$Q(H) = \frac{1}{2m} \sum_{(i,j) \in E(H)} \left(w(i,j) - \frac{k(i)k(j)}{2m} \right)$$

where $m = \sum_{(i,j) \in E} w(i,j)$ and $k(i) = \sum_{j \in \mathrm{in}(i)} w(i,j)$.

Louvain Community Detection II

- The change $\Delta Q(i,j)$ in modularity is derived as

$$\Delta Q = \left[\frac{\Sigma_{in} + 2k_{in}(i)}{2m} - \left(\frac{\Sigma_{tot} + k(i)}{2m}\right)^2\right] - \left[\frac{\Sigma_{in}}{2m} - \left(\frac{\Sigma_{tot}}{2m}\right)^2 - \left(\frac{k(i)}{2m}\right)^2\right]$$

  where Σ_{in} is sum of all the weights of the links inside the community i is moving into, Σ_{tot} is the sum of all the weights of the links to nodes in the community i is moving into, $k_{in}(i)$ is the sum of the weights of the links between i and other nodes in the community that i is moving into.
- This measure determines **how much the modularity changes via moving (joining) community** i **with community** j. At each round, these values are calculated for the community i and its neighboring communities and i is merged with $j^* \in \mathrm{argmax}_{j \in N(i)}\{\Delta Q(i,j)\}$
- Runs in time $O(n \log n)$

The MoJo Clustering Distance

The MoJo [11] metric is a clustering distance metric used for comparing software clusterings. The MoJo distance between two clusterings $\mathcal{C}_1, \mathcal{C}_2$ is defined as the **minimum number of moves and joins** to transform one clustering to another where

- **Move:** Moving a resource from one cluster to another (that includes moving a resource into a previously nonexistent cluster or creating a new cluster.

The MoJo Clustering Distance

The MoJo [11] metric is a clustering distance metric used for comparing software clusterings. The MoJo distance between two clusterings $\mathcal{C}_1, \mathcal{C}_2$ is defined as the **minimum number of moves and joins** to transform one clustering to another where

- **Move:** Moving a resource from one cluster to another (that includes moving a resource into a previously nonexistent cluster or creating a new cluster.
- **Join:** Join two clusters into one.

The MoJo Clustering Distance

The MoJo [11] metric is a clustering distance metric used for comparing software clusterings. The MoJo distance between two clusterings $\mathcal{C}_1, \mathcal{C}_2$ is defined as the **minimum number of moves and joins** to transform one clustering to another where

- **Move:** Moving a resource from one cluster to another (that includes moving a resource into a previously nonexistent cluster or creating a new cluster.
- **Join:** Join two clusters into one.
- The MoJo distance is therefore defined as

$$\mathrm{MoJo}(\mathcal{C}_1, \mathcal{C}_2) = \mathsf{min}\{\mathrm{mno}(\mathcal{C}_1, \mathcal{C}_2), \mathrm{mno}(\mathcal{C}_2, \mathcal{C}_1)\}$$

The MoJo Clustering Distance

The MoJo [11] metric is a clustering distance metric used for comparing software clusterings. The MoJo distance between two clusterings $\mathcal{C}_1, \mathcal{C}_2$ is defined as the **minimum number of moves and joins** to transform one clustering to another where

- **Move:** Moving a resource from one cluster to another (that includes moving a resource into a previously nonexistent cluster or creating a new cluster.
- **Join:** Join two clusters into one.
- The MoJo distance is therefore defined as

$$\mathrm{MoJo}(\mathcal{C}_1, \mathcal{C}_2) = \mathsf{min}\{\mathrm{mno}(\mathcal{C}_1, \mathcal{C}_2), \mathrm{mno}(\mathcal{C}_2, \mathcal{C}_1)\}$$

- Exact computation is not **efficient** so a **heuristic** is proposed.

Agglomerative Clustering I

Idea

In every iteration pick two points/vertices u and v that maximize a linkage function and merge them together.

Algorithm

function $\textsc{AgglomerativeClustering}(w, L, m, \mathbf{x}_1, \ldots, \mathbf{x}_n)$

$\quad \mathcal{C}_0 \leftarrow \{\{\mathbf{x}_1,\}, \ldots, \{\mathbf{x}_n\}\}$

$\quad$ **for** $1 \leq t \leq m$ **do**

$\quad\quad (\hat{A}, \hat{B}) \leftarrow \operatorname{argmax}_{A,B \in \mathcal{C}_{t-1}} L(|A|, |B|, w(A, B))$

$\quad\quad \mathcal{C}_t \leftarrow \mathcal{C}_{t-1} \setminus \{\{\hat{A}\}, \{\hat{B}\}\} \cup \{\{\hat{A} \cup \hat{B}\}\}$

$\quad$ **end for**

$\quad$ **return** $\mathcal{C}_m$

end function

Agglomerative Clustering II

Linkage functions vary

- Average Linkage[1] $\mathrm{argmax}_{A,B} \frac{w(A,B)}{|A||B|}$
- Complete Linkage $\mathrm{argmax}_{a \in A, b \in B} w(a, b)$
- Single Linkage $\mathrm{argmin}_{a \in A, b \in B} w(a, b)$
- Ward Linkage. Ward minimizes $J(C) = \sum_{\mathbf{x},\mathbf{y} \in C} \|\mathbf{x} - \mathbf{y}\|^2$ for each cluster $C \in \mathcal{C}$.
- Information Loss (Agglomerative Information Bottleneck Algorithm)

The affinity function w can be any distance measure. In our comparison, we have used the cosine distance affinity measure between the document embeddings.

[1] $w(A, B) = \sum_{a \in A, b \in B} w(a, b)$

Main Software Clustering Algorithms

The two main algorithms appearing in literature [8, 2] are

- LIMBO: Clusters modules upon inserting their Distributional Cluster Features to a B+-tree variant and then applying the Agglomerative Information Bottleneck algorithm.[2]
- ACDC: Leans toward to software components comprehension based on subsystem patterns.

[2] aka Agglomerative Clustering with Mutual Information Loss as Measure

Evaluation

- Our method was tested on Linux 4.21, consisting of 20.3 million SLOC against Average-Linkage [9], Complete-Linkage [4] and Ward-Linkage [13] using the same document embeddings as well as ACDC with structural information [12] and LIMBO [1] with Bag-of-Words features.
- 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`.
- Results are averaged over runs

Results I

Method	Dim.	n_c	Range	$\bar{x}$	σ	Median
ACDC	–	9055	1 – 4245	5	46	2
Average Linkage	*300*	*21*	1–3406	163	725	1
Complete Linkage	*300*	*21*	1–1529	163	412	19
LIMBO	12317	*21*	50–1810	163	375	50
Ward Linkage	*300*	*21*	21–948	163	223	70
SADE	*300*	10 (± 2)	2 (± 0) -132 (± 13)	64 (± 4)	40 (± 4)	65 (± 10)
SADE (Directed)	*300*	5 (± 2)	1 (± 1) - 614 (± 1)	141 (± 39)	253 (± 25)	2 (± 0.3)
Ground Truth	–	21	1–1348	163	341	11.0

Table: Experimental Results for Linux 4.21. Italics denote manually defined parameters

Results II

Method	MoJo Distance
ACDC	33694
Average Linkage	2092
Complete Linkage	1710
LIMBO [3]	1482
Ward Linkage[4]	1138
SADE	243 ($\pm$ 1)
SADE (Directed)	237 ($\pm$ 2)

Table: MoJo Distances

[3] $(B = 100,\ S = \infty)$ and Bag-of-Words Features

[4] Eucledian Affinity

Results & Discussion

- Surpass all clustering algorithms in terms of MoJo Distance Metric

Results & Discussion

- Surpass all clustering algorithms in terms of MoJo Distance Metric
- Production of balanced clusterings

Results & Discussion

- Surpass all clustering algorithms in terms of MoJo Distance Metric
- Production of balanced clusterings
- Production of stable clusterings

Results & Discussion

- Surpass all clustering algorithms in terms of MoJo Distance Metric
- Production of balanced clusterings
- Production of stable clusterings
- Results were produced without knowing the number of clusters of the ground truth a priori

Results & Discussion

- Surpass all clustering algorithms in terms of MoJo Distance Metric
- Production of balanced clusterings
- Production of stable clusterings
- Results were produced without knowing the number of clusters of the ground truth a priori
- Provide a simplistic approach to software clustering combining vector semantics and the call graph

Conclusions

1. Use **vector semantics** and the **call graph** to produce meaningful clusterings

Conclusions

1. Use **vector semantics** and the **call graph** to produce meaningful clusterings
2. Performing our study on a very large system (Linux) gives us further insight on the nature of software itself

Conclusions

1. Use **vector semantics** and the **call graph** to produce meaningful clusterings
2. Performing our study on a very large system (Linux) gives us further insight on the nature of software itself
3. Outperform state-of-the-art and baseline methods in terms of **authoritativeness** and **extremity**

Conclusions

1. Use **vector semantics** and the **call graph** to produce meaningful clusterings
2. Performing our study on a very large system (Linux) gives us further insight on the nature of software itself
3. Outperform state-of-the-art and baseline methods in terms of **authoritativeness** and **extremity**
4. Produce **stable** and **balanced** clusterings

Future Work

- Testing our system with various codebases (e.g. gcc, Postgres etc.)
- Integration with more static analyzers
- Propose layered software architectures via finding a Directed Spanning Tree of Maximum (or Minimum) weight.

Code and Data

Refererences

SourceCode

Paper

https://github.com/papachristoumarios/sade

Dataset

Thank you!

https://github.com/papachristoumarios/sade

https://zenodo.org/record/2652487

References I

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

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

Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Görke, Martin Hoefer, Zoran Nikoloski, and Dorothea Wagner.
Maximizing modularity is hard.
arXiv preprint physics/0608255, 2006.

References II

Daniel Defays.
An efficient algorithm for a complete link method.
The Computer Journal, 20(4):364–366, 1977.

Matthew Honnibal and Ines Montani.
spacy 2: Natural language understanding with bloom embeddings, convolutional neural networks and incremental parsing.
Convolutional Neural Networks and Incremental Parsing, 2017.

Quoc Le and Tomas Mikolov.
Distributed representations of sentences and documents.
In *International Conference on Machine Learning*, pages 1188–1196, 2014.

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

References III

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

Robert R Sokal.
A statistical method for evaluating systematic relationship.
University of Kansas science bulletin, 28:1409–1438, 1958.

Diomidis Spinellis.
Cscout: A refactoring browser for c.
Science of Computer Programming, 75(4):216, 2010.

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

References IV

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

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