Deploying Smart Program Understanding on a Large Code Base

Carlo Ieva*, Arnaud Gotlieb*, Souhila Kaci† and Nadjib Lazaar†
*Simula Research Laboratory, Oslo, Norway
Email: {carlo, arnaud}@simula.no
†LIRMM, University of Montpellier, Montpellier, France
Email: {kaci,lazaar}@lirmm.fr

***Abstract*—Program understanding aims at discovering human-readable properties of a software project from the analysis of its source code. Recently, we proposed a smart approach based on hierarchical agglomerative clustering that extracts so-called *program topoi* from source code. These topoi are high-level observable properties of the project. Based on textual and structural representations of the source code, our multi-steps approach clusters program topoi in an effective and efficient way. In this paper, we depict novel exploitation tasks of this program understanding approach and report on its application to Software Heritage. Software Heritage is an ambitious project which aims at collecting and archiving the biggest corpus of publicly available software source code. One of the project goals is to provide a new scientific instrument for computer scientists to evaluate advanced machine learning and software engineering methods on a very large source code repository. Our in-depth experiments reveal that unsupervised learning is the appropriate tool to mine and understand the biggest corpus of software source code ever produced.**

I. Introduction

Program understanding aims at discovering human-readable properties of a software project when only its source code is available. Addressing this problem is crucial whenever projects are developed by large communities of developers without any centralized specification process. The open source software community develops major software systems by regularly integrating experimental development branches into the main trunk and so, there is no central repository gathering all the specifications. For instance, the Linux kernel amounts to almost 20 MLOC (v4.12, July 2017) after more than 20 years of development, but there is no central repository gathering all its properties or specifications. Any new potential contributor faces the problem of understanding part of the kernel and has to imagine which property is best suited for validating it.

Recently, in [1], [2], we proposed a new program understanding approach called FEAT, which extracts so-called *program topoi* from the source code of a software project. These program topoi are high-level observable properties of the project, and they results from applying a tuned hierarchical agglomerative clustering process using social network analysis metrics [3]. Based on textual and structural representations of the source code, this multi-steps approach clusters program topoi in an effective and efficient way. Finding a specific program topoi in a large source code repository is hard because capturing the semantics of programs in the presence of tons and tons of implementation details is like finding a needle in a haystack. By leveraging an unsupervised machine learning technique, namely clustering, FEAT automatically extracts program topoi from source code by combining call graph elements and code and comments processing. More specifically, the approach is based on hierarchical agglomerative clustering because this technique allows its users to define well-tuned distance metric between code units. This was crucial to extract program topoi which capture by essence both structural and semantic aspects.

In the present paper, we depict novel exploitation tasks of the FEAT program understanding approach and report on its application to Software Heritage. Started in 2016, Software Heritage[1] (SH) is an ambitious non-profit project which aims at collecting and preserving for the very long term the biggest corpus of source code ever produced on earth [4]. By December 2018, the archive has collected 5,4 billion source code files extracted from 87,03 million software projects. Even if the primary goal of the project is to collect and preserve the source code, offering means to search and share the archive is crucial to foster its larger adoption. However, as said above, mining very large repositories of source code for program understanding is challenging. It requires not only to parse an enormous number of files, but also to construct artifacts which would allow final users to understand the structure and semantics of the source code. Searching a few keywords in an indexed list of keywords is easy but understanding how code units (e.g., C functions or Java methods) relate to each other and cluster them to see how high-level code feature are implemented is much more difficult. The paper reports on a large-scale in-depth experiment performed on 431 software projects extracted from SH, accounting for more than 25 MLOC. One goal was to study the impact of the various internal steps of the FEAT approach on the CPU time. Another goal was to find out correlations between the size of projects and the CPU time and memory usage taken by the approach. The overall results show that FEAT is an appropriate tool to mine large code base repositories such as SH.

This work is supported by the Certus SFI grant of the Research Council of Norway

[1] www.softwareheritage.org

There is more than a decade of intensive research to understand how to mine source code for program comprehension [5]–[11]. Our work inherits from results in feature location [12] and feature extraction [13]. FEAT differs from most of existing feature location/extraction works by its usage of unsupervised machine learning together with a dedicated hybrid distance which borrows ideas from social network analysis and natural language processing. By applying FEAT in Software Heritage, we provide a unique search capability which can retrieve program topoi from archived source code, but other more advanced use cases are forseen. Apart from feature location and extraction, semantic classification of software projects with summary generation, semantic clone detection or else version differentiating can be envisioned with FEAT. Through an in-depth experimental evaluation, this paper explores the usage of FEAT on the Software Heritage archive.

The rest of the paper includes Sec.II which gives crucial background on clustering and our hybrid distance combining structural and semantic analysis; Sec.III which presents novel exploitation tasks of FEAT opening the door to new applications of the tool in the Software Heritage archive; Sec.IV gives detailed experimental results obtained on 431 software projects. Finally, Sec.V concludes the paper.

II. The FEAT Program Understanding Approach

This section recalls the necessary background to understand the FEAT approach, which is an automatic approach devoted to understanding source code of large open-source software projects. An overview of FEAT is depicted in figure 1. The approach includes three distinct steps, namely *Preprocessing* (step 1 in the figure), *Hybrid Clustering* (step 2) and, *Entry point selection* (setp 3) which are detailed in the rest of the section.

A. Preprocessing - Step 1

FEAT takes as input the source code of a given software project, possibly augmented with source-level comments. From the source code and the comments, the corresponding call graph CG is extracted as well as a set of unit-documents $\mathcal{D}$. For a software project, the *call graph* captures the caller/callee relationship by considering one node per code unit (i.e., function or method) and each arc associated to the possible calls between these units. Note that a (single) arc $f \longrightarrow g$ is created in the graph as soon as there is a call orginating from f to function g. The call graph CG captures some abstraction of the structure of the project. A *unit-document* for a selected code unit is a bag of words extracted from the analysis of that code unit, augmented with its code-level comments. After a stemming and cleaning phase, only the relevant root-words are kept for a specific unit. This allows us to capture some semantical part of the code. More details on this preprocessing phase can be found in [2]. An alternative could have been to use CODE2VEC [14] for this task but we wanted to keep full control over the extracted elements and also to evaluate the impact of source elements and comments on the quality of extracted program topoi.

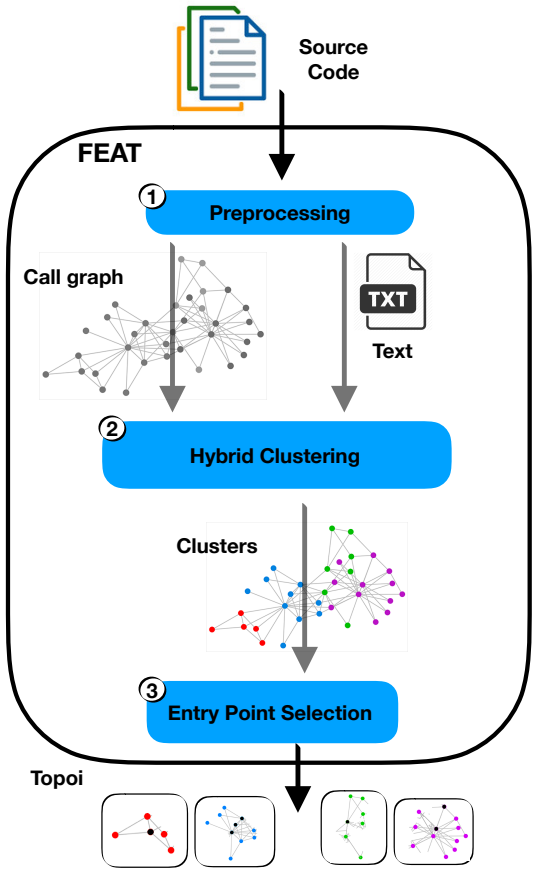

Fig. 1. FEAT process's overview

The result of this step is a pair $(\mathrm{CG}, \mathcal{D})$ which is used as input for the next step.

B. Hybrid Clustering - step 2

From the call graph and the document-units, FEAT runs a clustering step to compute a partition of the code units. The underlying technique is based on Hierarchical Agglomerative Clustering (HAC) [15], with dedicated notions of distances between units and clusters.

Distance Definition. Let C_i and C_j be two code unit clusters, the FEAT hybrid distance is a parametrized linear combination of two distances, namely $\mathrm{d_{CG}}$, a metric defined over the call graph and $\mathrm{d}_{\mathcal{D}}$, a distance computed over the extracted bag of words. Formally speaking,

$$\mathrm{d_{FEAT}}(C_i, C_j) = \alpha \mathrm{d_{CG}}(C_i, C_j) + (1-\alpha)\mathrm{d}_{\mathcal{D}}(C_i, C_j) \quad (1)$$

where α is a user-defined parameter which allows us to steer the process towards one or the other direction. Both $\mathrm{d_{CG}}$ and $\mathrm{d}_{\mathcal{D}}$ are normalized to take their real value in $[0, 1]$. On the call graph, the distance between two clusters C_i and C_j is reduced to the distance between their respective *medoids* m_i and m_j, which are the units lying at the most central position w.r.t. all units in a cluster [15]. $\mathrm{d_{CG}}$ is thus defined on the length of

the shortest path that exists between the two medoids (noted $|m_i \rightarrow m_j|$):

$$\mathrm{d_{CG}}(C_i, C_j) = \begin{cases} 0 & |m_i \rightarrow m_j| = 0 \\ \frac{1-\lambda}{1-\lambda^g} \sum_{i=0}^{k-1} \lambda^i & |m_i \rightarrow m_j| = k \\ 1 & |m_i \rightarrow m_j| = \infty \end{cases} \quad (2)$$

where g represents the call graph diameter and $\lambda > 1$ a constant which provides exponential growth of the distance.

Regarding the distance between document-units, we considered the angular distance between centroids of the clusters. Documents-units lie in an Euclidean space, then we considered the following formula for computing the centroid of a cluster C_i is $\mu_i = \frac{\mathbf{d}_1+\mathbf{d}_2+\cdots+\mathbf{d}_n}{|C_i|}$ and the $\mathrm{d}_{\mathcal{D}}$ distance:

$$\mathrm{d}_{\mathcal{D}}(C_i, C_j) = \frac{2}{\pi} \arccos\left(\frac{\mathbf{d}_{\mu_i} \cdot \mathbf{d}_{\mu_j}}{\|\mathbf{d}_{\mu_i}\| \|\mathbf{d}_{\mu_j}\|}\right) \quad (3)$$

Note that $\mathrm{d_{CG}}$ and $\mathrm{d}_{\mathcal{D}}$ are proper distances ensuring symmetry, positiveness and triangular inequality properties.

Cutting Criterion. By considering a bottom-up HAC process, we started with a partition where each cluster contains a single unit and then, step after step, the process agglomerates the various units in cluster. At each iteration, we merge clusters by computing the $\mathrm{d_{FEAT}}$ distance. The process can be interrupted at any time and that is why, we had to propose a so-called *cutting-criterion*, combining the structural and semantic parts of the problem:

(i) Modularity [3]: One of the most effective approaches for detecting clusters in graphs is based on the optimization of a measure known as *modularity*, which comes from social network analysis. Given a partition of vertices of a graph into disjoint clusters, modularity reflects the concentration of edges within clusters compared with random distribution of links between all nodes regardless of clusters.

Formally speaking, the modularity of a given partition $\mathcal{P}$ is:

$$\mathrm{Q}(\mathcal{P}) = \frac{1}{2m} \sum_{i,j} \left(A_{i,j} - \frac{k_i k_j}{2m}\right) \delta(C_i, C_j) \quad (4)$$

where A is the adjacency matrix of CG. That is, $A_{ij} = 1$ if there exists an edge between units u_i and u_j and $A_{ij} = 0$ otherwise. k_i (resp. k_j) is the degree of unit u_i (resp. u_j), C_i (resp. C_j) is the cluster containing u_i (resp. u_j), and m is the total number of edges. Function δ is the Kronecker delta: $\delta(C_i, C_j) = 1$ iff $C_i = C_j$ (units u_i,u_j are in the same cluster), 0 otherwise. High values of modularity correspond to interesting partitions of a call graph.

(ii) Coherence [16]: *Coherence* is a measure adopted in natural language processing (NLP), used for assessing how similar are the segments of a text. Coherence is based on the measure of words overlapping. We consider all unit-documents belonging to a cluster as sections of a whole text. Developers, within the context of the functions participating in the implementation of a system capability, use a consistent language revealed through the choice of names for variables, the text in comments, etc. So, while looking at clusters as textual documents, we want to find the partition $\mathcal{P}$ showing the highest coherence:

$$\mathrm{H}(\mathcal{P}) = \sum_{\forall \mathcal{C} \in \mathcal{P}} \left(1 - \frac{2}{|\mathcal{C}|(|\mathcal{C}|-1)} \sum_{k=1}^{|\mathcal{C}|} \sum_{j=k+1}^{|\mathcal{C}|} \mathrm{d}_{\mathcal{D}}(u_k, u_j)\right) \quad (5)$$

In order to return a high-quality partition, we considered a hybrid criterion to stop the clustering process based on a combination of coherence and modularity. Knowing that $\mathrm{Q} \in [-\frac{1}{2}, 1]$ and $\mathrm{H} \in [0, |\mathcal{P}|]$, we considered a linear combination of modularity and coherence using α as follows:

$$\mathrm{T_{FEAT}}(\mathcal{P}) = \alpha \frac{\mathrm{H}(\mathcal{P})}{|\mathcal{P}|} + (1-\alpha)\frac{2\mathrm{Q}(\mathcal{P})+1}{3} \quad (6)$$

FEAT uses the *priority queue* version of HAC having a time complexity of $\Theta(n^2 \log(n))$. The algorithm starts by considering each unit as a cluster. It computes the pairwise distances between clusters until either the entire partition is reduced to a single cluster or the cutting criterion is reached (i.e., $\mathrm{T_{FEAT}}$ value cannot be improved). At the end, the algorithm returns a partition $\mathcal{P}$ of m clusters. The overall hybrid clustering process is detailed in [2]. The output of this step is a set of clusters, each composed of several code units and dictionary extracted from the source code analysis.

C. Entry Point Selection - Step 3

In this third step, the FEAT approach selects the entry-points of each cluster C_i. An *entry point* is a code unit that gives access to the implementation of an observable system functionality. This notion is at the heart of program understanding in our approach. Examples of entry-point extracted from famous software projects include menu-click handlers in graphical user interfaces, public methods of APIs, etc.

By using Principal Component Analysis technique, a classical dimensionality reduction technique used in Machine Learning, it is possible to rank each unit present in clusters. The notion of entry-point is based on the two following assumptions: (i) Unlike other units, entry points are more likely to be the starting point of longer calling chains ; (ii) Entry-point are unlikely present at the end of calling chains ; Thus, given a code unit u, it is possible to characterize entry-points by using the following vector:
$\mathbf{v}_u = [\deg^-(u), \deg^+(u), \mathrm{RI}(u), \mathrm{RO}(u), \mathrm{SI}(u), \mathrm{SO}(u)]$:

- $\deg^-(u)$: number of incoming arcs of u;
- $\deg^+(u)$: number of outgoing arcs of u;
- $\mathrm{RI}(u)$: number of paths ending in u;
- $\mathrm{RO}(u)$: number of paths starting from u;
- $\mathrm{SI}(u)$: Sum of the lengths of all paths to u;
- $\mathrm{SO}(u)$: Sum of the lengths of all paths having u as source.

Using this vector, computed for every unit in a clustered set of units, it is then possible to rank all code units according to their respective $\mathbf{v}_u$ vectors. Ranking is performed using the two following criteria for optimality: (1) *Input*-attributes are subject of minimization; (2) *Output*-attributes are subject of maximization. Thus, for each cluster, the entry-point selection

step allows us to extract program topoi which are ordered subset of units (i.e., entry-point candidates) provided together with a dictonary of words.

D. Implementation of the FEAT Approach

The FEAT approach was implemented and made accessible through an interface shown in figure 2. The main application that has been proposed so far is feature location. *Feature location* aims at locating specific functions and their code implementations in software projects [12]. This application is usually considered as a difficult problem to handle as there is usually no semantics associated with syntactic search engines. Unlike these approaches, FEAT allows for searching using semantics information as shown in figure 2. This example illustrates the benefits of searching a large code repository to find meaningful infomations related to a specific topic. Let us propose to search implementations of how to print postscript documents. In the main window (noted 1 in the figure), FEAT is used as a search engine with keywords `+print, +postscript, -pdf`. From there, all software projects having program topoi related to print documents in postscript and not as pdf in their implementation dictionary (2 in the figure) are returned. Selecting one topoi allows us to visualize the subgraph of the call graph associated to that topoi (3).

III. FEAT in Software Heritage

The deployment of FEAT in Software Heritage (SH) has started with the initial application in mind, namely feature location. However, other exploitation tasks for FEAT were envisionned. This section presents these exploitation tasks and advocates for the larger adoption of FEAT in the context of SH.

A. Feature Location in SH

By querying the SH archive with feature-related keywords using a domain-specific language (e.g., SQL), FEAT can find program topoi which implement these features. Since the current granularity of access to SH is limited to files, the tool returns file names which contains units participating to the extracted program topoi. Once these files are found, visualizing the program topoi (both its index and the subgraph of the call graph) allows the final users to retrieve the implemented features corresponding to the submitted keywords. Figure 2 illustrates this use case of FEAT in the context of SH.

This use case of FEAT in SH is a first step in providing an advanced semantic search capability over this very large repository of source code.

B. Automatic Feature Extraction

In Software Engineering, software repository mining is considered mainstream in feature extraction [12], [17], [18]. Knowing that the source code of a given project has been archived in SH, FEAT could be used to extract automatically all the program topoi related to the project. Doing so, it could extract high-level user-observable capabilities of a software project. Unlike techniques which mine software documentations such as [8] or more recently [13], mining the source code possibly augmented with low-level comments present the advantage to extract actual features. The closest approaches to FEAT include [7] which exploits latent Dirichlet allocation to discover the most important code-units under the form of *topics*, [9] which is a source-code recommendation system based on variability models, [6] which uses clustering and Latent Semantic Indexing to assess the similarity between source-code artefacts and create clusters according to their similarity, [5] which exploits a sequential combination of information retrieval technologies and call graph analysis of the program. By defining its own distance notion which hybridizes structural and semantic elements, FEAT clusters source code units in a meaningful way. Unlike the above mentioned approaches, FEAT extracts program topoi which contain structural elements related to the call graph of the application and semantic elements coming from the extracted indexes from each source code units. Additionally, by using a parametrized linear combination of distances, FEAT can tune its usage of structure or semantics for each project. Note however that exploiting program topoi requires a deep understanding of the mechanisms which have been used to produce them. So, we believe that further experiments are needed to evaluate the ergonomy of the tool for feature extraction.

C. Project Versions Difference Analysis

By extracting program topoi from two successive versions of the same application, FEAT could be useful to automatically understand and document code changes. A newly discovered topoi would correspond to a new implemented feature (a.k.a., *evolutive maintenance*) while no difference between topoi would reveal that only corrections have been deployed in the new version (a.k.a., *corrective maintenance*). Providing a tool in SH which can help engineers to maintain the code from one version to another would be helpful especially when code changes span over all the source code. Traditional approaches for this problem tend to focus on syntactic changes only (i.e., paths) while FEAT can perform semantic differentiation between versions.

D. Semantic Clone Detection

Maintaining a repository of all source code produced on earth opens the door to the automatic control of plagiarism in source code production or source code patenting. In principle, any Company having performed an official deposit to SH would be in position to control that no other third-party has copied illegally the source code to include it elsewhere. Even this perspective of SH is appealing, it entails the ability to parse SH for *semantic clone detection* in addition to syntactic checking. By extracting program topoi, a tool like FEAT could check whether the same topoi are present twice or more in the archive and thus detect automatically illegal clones.

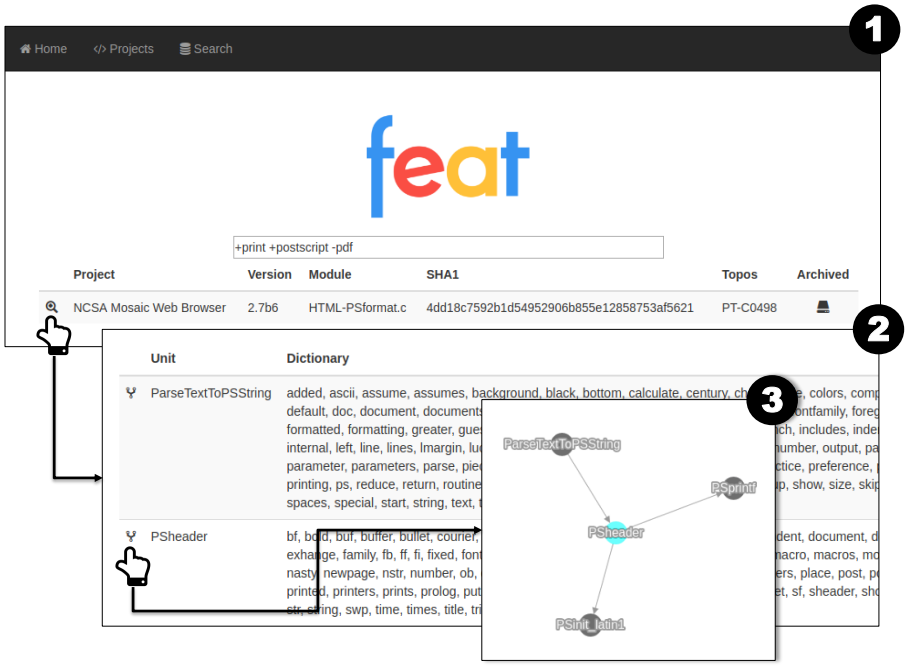

Fig. 2. Feature Location with FEAT in Software Heritage

E. Summary Generation and classification of Program Topoi

By generating natural language summaries of source-code artifacts (classes and code change sets), FEAT could be used in SH to generate automatically faithful summaries of archived projects. By classifying software projects through their implemented program topoi, summaries could be generated for each given class of project. Hence, the final users of SH would have access to a comprehensive text which describe each of the archived project. This summary generation feature of SH could help computer scientists to dig into SH in a meaningful and comprehensive way when they search for specific projects.

F. Mining Software Test Scripts

Software test scripts are formed of source code which calls the system-under-test in order to check it. These scripts contain hidden informations which cannot easily be revealed without a deep analysis of the testing process. For instance, dependencies between test scripts and individual test cases is usually not documented, as well as the inter-dependencies between test cases. We envision that FEAT could play a crucial role in automatically discovering these links by parsing the test script source code. In fact, extracted program topoi could gather test cases in a meaningful way and classify them by degree of importance. Playing with the parametrized approach of FEAT (with α), the role of semantics vs structure of test script could be explored in depth and could reveal the hidden structure of the test process. Additionaly, by clustering test cases, FEAT could optimize the test execution process by running only one test case per cluster. This would allow a better handling of redundancy among test cases, especially when similar user requirements are tested by distinct test cases.

IV. Experimental Evaluation on SH

This section presents our experimental evaluation on the performance of FEAT over a large selection of SH software projects.

The FEAT approach has been implemented on top of the software testing platform called CRYSTAL, which is based on OSGi (Open Services Gateway initiative) and BPMN (Business Process Modeling Notation). The experiments are carried out on an 8 CPU (Intel Xeon E5-2686 2.3 GHz), 61 GB RAM and 32 GB SSD hard disk.

We processed 431 projects downloaded from SH with FEAT. Projects were selected based on their diversity and difference. The goal of the experiments was to explore the scalability of the FEAT implementation. The selected projects are extracted from various application domains, written in the C programming language. The project size (compressed) ranges in between 150 Bytes and 50 MBytes. The SH repository is organized as a Merkle tree and the projects, which are nodes in this tree, are identified through their SHA1 code [4]. The 431 extracted projects amounts for $48,187$ files, $428,480$ units, a dictionary of $543,564$ words and more than 25 million lines of code (MLOC).

Table I reports statistics on the selected SH projects in terms of LOC, units, dictionary size and call graph density. For each characteristic, we report the min/max values, the average (AVG) value and the standard deviation (SD).

	MIN	MAX	AVR	SD
LOC	37	1,611,756	49,680	99,166
Units	3	3,964	844	886
Dictionary	4	5,980	1,261	1,053
CG Density	10^{-4}	0.7	0.02	0.06

TABLE I
MAIN CHARACTERISTICS OF A SELECTION OF PROJECTS EXTRACTED FROM SOFTWARE HERITAGE

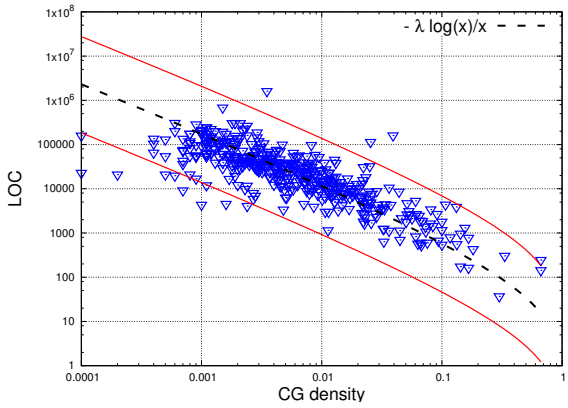

Fig. 3. Exploring the correlation between the density of call graphs w.r.t. the number of lines of code (LOC).

The first question we wanted to address is related to the potential correlation between project size and the complexity of project structure. For that, figure 3 depicts the density of the call graph of the selected SH projects according to their size in terms of LOC. Interestingly, over the 431 selected software projects, there is a negative correlation between LOC and call graph density as shown in the figure. The density follows a negative ($\frac{log(x)}{x}$) scale when LOC grows. For our experiments, we denote 17 projects with a call graph density exceeding 0.1 and reaching 0.7. 127 projects have a density ranged between 0.1 and 0.01. The remaining 287 projects are very sparse with LOC exceeding 10K. So, as a first finding, adding new code units increases mechanically the project size, but we can expect that it will also decrease the density of the call graph.

The purpose of our second experimental analysis was to examine the impact of LOC / #Units over the CPU time required to run FEAT.

Let us start with the relation between LOC and #Units. Figure 4.(a) reports a perfect positive correlation between the number of units and LOC. The number of units follows a linear scale when LOC increases. Therefore, the number of units and LOC of a project will always have the same impact on the performance of FEAT.

Now, let us take a close look at the impact of #Units on CPU time. From figure 4.(b), we observe that FEAT is able to deal with 4K units in less than 20 minutes. For projects of up to 1K units, FEAT extracts program topoi in less than 10 seconds and needs less than 2 minutes on projects of up to 2.5K units. Here, the correlation between #Units and CPU time is near-perfect positive. When the number of units grows, the CPU time follows a $n^2log(n)$ scale. This is especially true

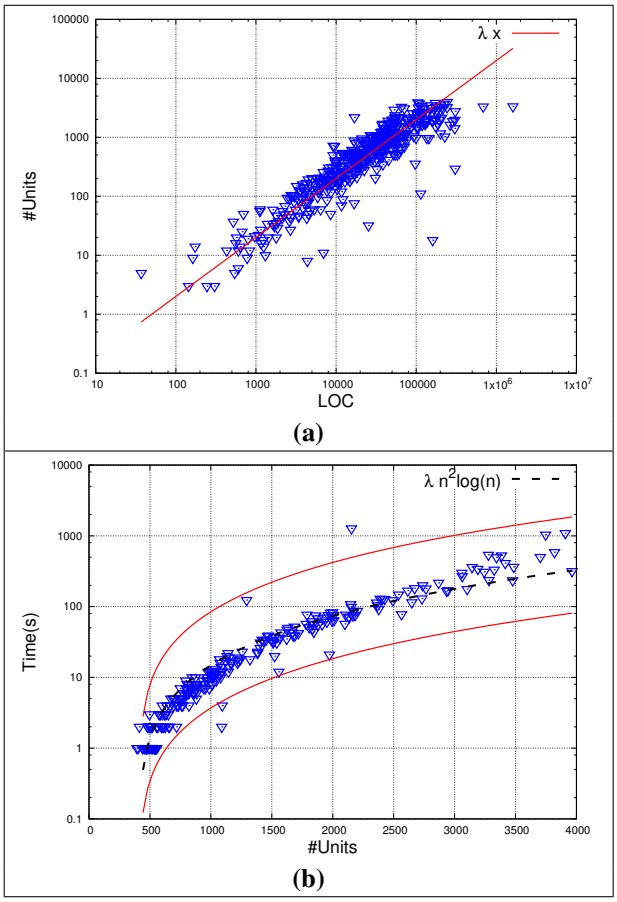

Fig. 4. Exploring the CPU time taken by FEAT w.r.t. the number of lines of code (a) and the number of units (b).

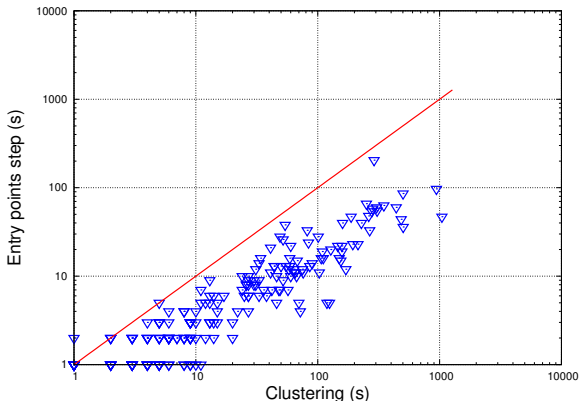

Fig. 5. Exploring the correlation between the clustering step of FEAT and the entry-point selection step in terms of CPU time.

when the hybrid clustering step of FEAT dominates the whole process of topoi extraction. That is, the hybrid clustering of FEAT has a complexity of $\Theta(n^2log(n))$ where n refers to the number of units. To strengthen our previous conclusion, we report in figure 5 the CPU time taken by steps 2 and 3 of the FEAT approach, namely hybrid clustering and entry-point selection. Here, the clustering step clearly dominates the entry point selection step. For instance, if we take P_i a project

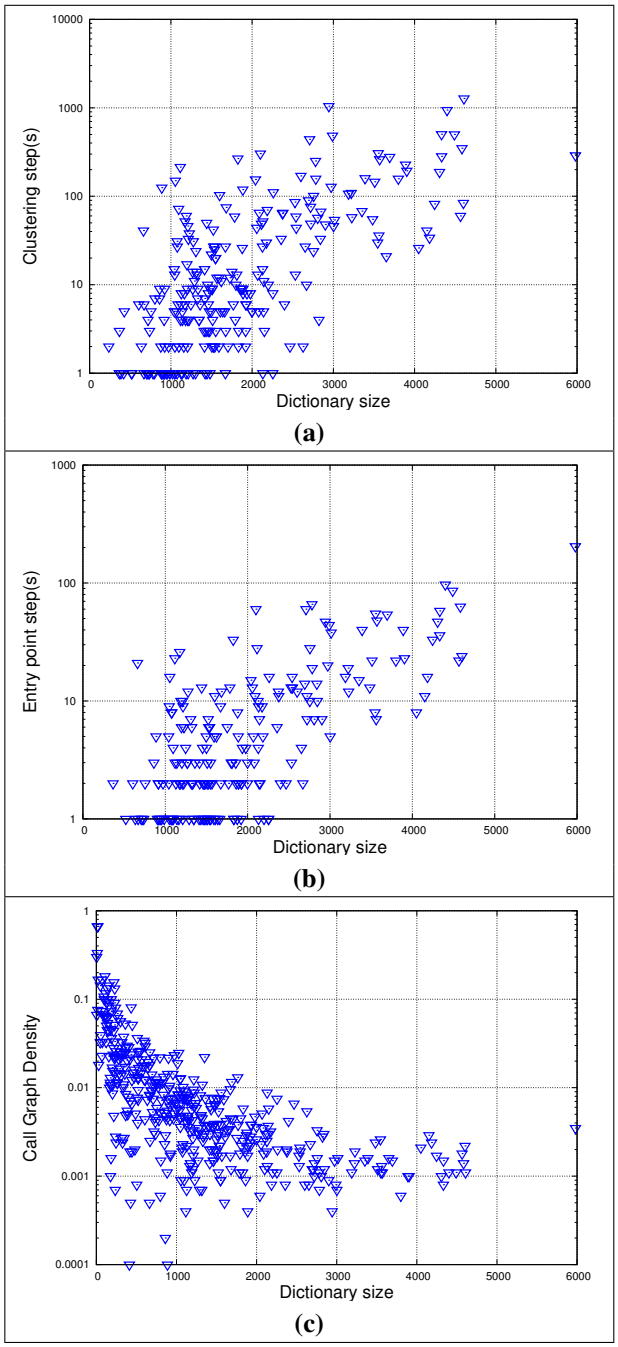

Fig. 6. Exploring the impact of the dictionary size on the CPU time taken by the hybrid clustering step of FEAT (a), the entry-point selection step (b) and the call graph density (c).

of more than 1 MLOC and 3,909 units, FEAT extracts 120 topoi with 18 minutes of clustering and only 47 seconds of entry-point selection.

As the size of the dictionary plays a role in both steps, we also report in figure 6 on its impact over the FEAT approach. The main observation is that the impact of the dictionary is exactly the same on both steps. The CPU time can increase by increasing the size of the dictionary. The observed variability

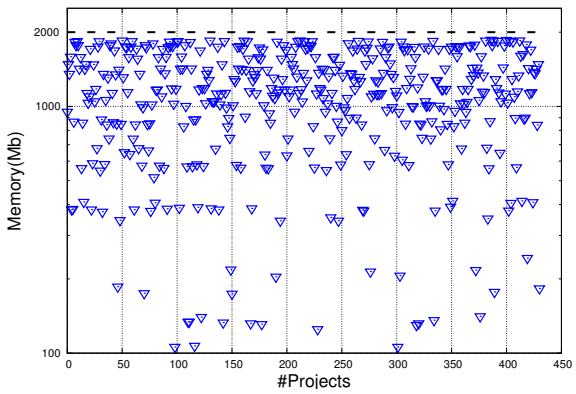

Fig. 7. Exploring the memory size taken by the FEAT approach w.r.t. the size of software projects extracted from Software Heritage.

can be explained by the variability found in the density of the call graphs (see figure 6.(3))

Our last experiment is on the memory usage of FEAT. Figure 7 reports the memory in Mb used to extract topoi from the 431 SH projects. The memory usage of FEAT seems to be quite acceptable, as long as no FEAT call consumes more than 2Gb.

In conclusion, FEAT shows acceptable performance with projects counting up to 4K units and dictionaries with size up to 6K words producing results in about 20 minutes and using less than 2Gb of memory. Interestingly, the results are consistent over multiple application domains and does not really suffer from other parameters than the number of units (or LOC as both are perfectly correlated) and size of dictionaries.

V. Conclusions

Searching very large repositories of source code for program understanding is challenging not only because of the volume of projects and files to examine, but also because it requires to perform both structural and semantic analysis in association. By leveraging an unsupervised machine learning technique such as hierarchical agglomerative clustering, the FEAT approach is able to process software projects in a meaningful way to extract automatically what we have called *program topoi*. These topoi can be seen as high-level user-observable program features which are materialized by an ordered list of functions together with an index of keywords. Even if a longer period of usage is required to confirm the importance of our approach in Software Heritage, we can already draw some interesting perspectives. As discussed earlier, it opens the door to ambitious applications and use cases such as the semantic classification of all project source code or the automatic detection of semantic clones at a very large scale, but also the mining of test scripts to find hidden links between test cases. Even if several improvements are necessary, the availability of semantic search in Software Heritage offers Software Engineering researchers a new scientific instrument

to explore the vast variety of source codes produced in many different software projects.

REFERENCES

[1] C. Ieva, A. Gotlieb, S. Kaci, and N. Lazaar, "Discovering program topoi through clustering," in *Proceedings of the Thirty-Second IAAI Conference on Innovative Applications of Artificial Intelligence, February 2-7, 2018, New Orleans, Louisiana, USA*. AAAI Press, 2018.

[2] ——, "Discovering program topoi via hierarchical agglomerative clustering," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 758–770, 2018.

[3] C. Aaron, M. Newman, and C. Moore, "Finding community structure in very large networks," *Physical Reviews E.*, vol. 70, 2004.

[4] R. Di Cosmo and S. Zacchiroli, "Software heritage: Why and how to preserve software source code," in *Proc. of the 14th International Conference on Digital Preservation (iPRES'17)*, 2017.

[5] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang, "SNIAFL: towards a static non-interactive approach to feature location," in *Proc. of the 26th International Conference on Software Engineering (ICSE'04), 23-28 May 2004, Edinburgh, United Kingdom*, 2004, pp. 293–303.

[6] A. Kuhn, S. Ducasse, and T. Gîrba, "Semantic clustering: Identifying topics in source code," *Information and Software Technology*, vol. 49, no. 3, pp. 230–243, 2007.

[7] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi, "Mining concepts from code with probabilistic topic models," in *Proc. of the IEEE Automated Software Engineering Conference (ASE'07)*, Apr. 2007, p. 461.

[8] H. Dumitru, M. Gibiec, N. Hariri, J. Cleland-Huang, B. Mobasher, C. Castro-Herrera, and M. Mirakhorli, "On-demand feature recommendations derived from mining public product descriptions," in *Proc. of the IEEE International Conference in Software Engineering (ICSE'11)*, 2011, pp. 181–190.

[9] C. McMillan, N. Hariri, D. Poshyvanyk, and J. Cleland-Huang, "Recommending Source Code for Use in Rapid Software Prototypes," in *Proc. of the IEEE International Conference in Software Engineering (ICSE'12)*, 2012, pp. 848–858.

[10] S. Grant, J. R. Cordy, and D. B. Skillicorn, "Using heuristics to estimate an appropriate number of latent topics in source code analysis," *Science of Computer Programming*, vol. 78, no. 9, pp. 1663–1678, 2013.

[11] S. L. Abebe and P. Tonella, "Extraction of domain concepts from the source code," *Science of Computer Programming*, vol. 98, pp. 680–706, 2015. [Online]. Available: https://doi.org/10.1016/j.scico.2014.09.012

[12] J. Rubin and M. Chechik, "A survey of feature location techniques," *Domain Engineering*, pp. 29–58, 2013.

[13] P. W. McBurney, C. Liu, and C. McMillan, "Automated feature discovery via sentence selection and source code summarization," *Journal of Software Evolution and Process*, vol. 28, no. 2, pp. 120–145, Feb. 2016.

[14] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *CoRR*, vol. abs/1803.09473, 2018. [Online]. Available: http://arxiv.org/abs/1803.09473

[15] L. Kaufman and P. J. Rousseeuw, *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley, 1990.

[16] P. W. Foltz, W. Kintsch, and T. K. Landauer, "The Measurement of Textual Coherence with Latent Semantic Analysis," *Discourse Processes*, vol. 25, no. 2-3, pp. 285–307, 1998.

[17] K. Chen and V. Rajlich, "Case study of feature location using dependence graph," in *Proc. of the 8th International Workshop in Program Comprehension (IWPC'00)*, 2000, pp. 241–247.

[18] A. Marcus and S. Haiduc, *Text Retrieval Approaches for Concept Location in Source Code*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 126–158.