Discovering Program Topoi via Hierarchical Agglomerative Clustering

Carlo Ieva, Arnaud Gotlieb, Souhila Kaci, Nadjib Lazaar

Discovering Program Topoi via Hierarchical Agglomerative Clustering

Carlo Ieva, *Member, IEEE*, Arnaud Gotlieb *Member, IEEE*, Souhila Kaci and Nadjib Lazaar

***Abstract*—In long lifespan software-systems, specification documents can be outdated or even missing. Developing new software releases or checking whether some user requirements are still valid becomes challenging in this context. This challenge can be addressed by extracting high-level observable capabilities of a system by mining its source code and the available source-level documentation. This paper presents FEAT, an approach that automatically extracts *topoi*, which are summaries of the main capabilities of a program, given under the form of collections of code functions along with an *index*. FEAT acts in two steps: (1) *Clustering*. By mining the available source code, possibly augmented with code-level comments, hierarchical agglomerative clustering groups similar code functions. In addition, this process gathers an index for each function; (2) *Entry-Point Selection*. Functions within a cluster are then ranked and presented to validation engineers as topoi candidates.**

We implemented FEAT on top of a general-purpose test management and optimization platform and performed an experimental study over 15 open-source software projects amounting to more than 1 MLOC proving that automatically discovering topoi is feasible and meaningful on realistic projects.

***Index Terms*—Program Analysis, Software Maintenance, Source Code Mining, Clustering, Program Topos .**

Notation

A	Accuracy
fn	False negative
fp	False positive
P	Precision
R	Recall
tn	True negative
tp	True positive

Abbreviations & Acronyms

HAC	Hierarchical Agglomerative Clustering
NLP	Natural Language Processing
PCA	Principal Component Analysis
VSM	Vector Space Model

I. Introduction

A. Context and Challenge

SOFTWARE systems are developed to satisfy an identified set of user requirements. When the initial version of a system is developed, contractual documents are produced to agree on its capabilities. However, when the system evolves over a long period of time, the initial user requirements can become obsolete or even disappear. This mainly happens because of evolution of systems, maintenance either corrective or adaptive and personnel turn-over. When new business cases are considered, software engineers face the challenge of recovering the main capabilities of a system from existing source code and low-level code documentation. Unfortunately, recovering user-observable capabilities is extremely hard since they are hidden behind the complexity of countless implementation details.

Our work focuses on finding a cost-effective solution to this challenging problem by automatically extracting *topoi*, which can be seen as summaries of the main capabilities of a program. These topoi are given under the form of collections of ordered code functions along with a set of words (*index*) characterizing their purpose. Unlike requirements from external repositories or documents, which may be outdated, vague or incomplete, topoi extracted from source code are an actual and accurate representation of the capabilities of a system. Our notion of topos makes the concept of *feature*, defined by [1] as *"…a feature is a set of logically related functional requirements that provides a capability to the user and enables the satisfaction of a business objective"*, more concrete and suitable for an automated computation.

This paper presents FEAT, an approach that automatically extracts *topoi*, which are summaries of the main capabilities of a program, given under the form of collections of code functions along with an *index*. FEAT acts in two steps: (1) *Clustering*. By mining the available source code, possibly augmented with code-level comments, hierarchical agglomerative clustering groups similar code functions. In addition, this process gathers an index for each function; (2) *Entry-Point Selection*. Functions within a cluster are then ranked and presented to validation engineers as topoi candidates;

Our work differs from automatic feature extraction (see a detailed overview in Sec.II) for two reasons. First, FEAT extracts topos which are structured summaries of the main capabilities of the program, while features are usually just informal description of software characteristics. Second, FEAT extracts topos by using an unsupervised machine learning technique not requiring any additional data to the bare source code.

B. Contribution of the paper

The contribution of this paper is three-fold:

C. Ieva and A. Gotlieb are with the Department of Software Engineering, Simula Research Laboratory, Oslo, Norway e-mail: carlo@simula.no; arnaud@simula.no.

Souhila Kaci and Nadjib Lazaar are with LIRMM, University of Montpellier, Montpellier, France email: kaci@lirmm.fr; lazaar@lirmm.fr.

Manuscript received April 19, 2005; revised August 26, 2015.

1) We present FEAT, a fully automated approach for topos extraction based on hierarchical agglomerative clustering functions. Our approach introduces an original, hybrid distance combining lexical and structural proximity between functions. This distance, through the definition of *graph medoids* (an extension of the classical notion of medoids [2]), can be applied also to set of functions. In addition, our approach makes use of *graph modularity* [3] to select the appropriate number of clusters;
2) Our method extracts topoi by sorting code functions through *Principal Component Analysis (PCA)*, which is a classical technique to deal with high-dimensional data [4]. PCA, in the context of a cluster, classifies functions as topos candidates;
3) We implemented FEAT on top of a general-purpose software testing platform and performed a large-scale experimental analysis over 15 open-source projects amounting to more than 1M lines of code. Our results show that automatic topos extraction is feasible on realistic projects and it can effectively assist human-based analysis of long time spanning software systems.

C. *Organisation of the paper*

The rest of the paper is organized as follows. Sec.II presents the most relevant works in the area of feature extraction. Sec.III gives the necessary background on clustering, distance notions and call graph. Sec.IV details the two main steps of our FEAT approach. Sec.V gives the experimental results obtained with FEAT on 15 open-source software projects. Finally, Sec.VI concludes the paper and draws some perspectives to this work.

II. Related work

Feature extraction [5], [6], [7] aims at automatically discovering the main characteristics of a software system by analysing its source code. It must be distinguished from *feature location*, whose objective is to locate where and how these characteristics are implemented [5]. Feature location requires the user to provide an input query where the searched characteristic is already known, while feature extraction tries to automatically discover these characteristics. Since several years, software repository mining is considered mainstream in feature extraction. However, we can distinguish between software-repository mining approaches dealing with software documentation only and, those dealing with source code only.

Mining Software Documentation. In [8], both text-mining techniques and flat clustering are used to extract feature descriptors from user requirements kept in software repositories. By combining association-rules mining and k-Nearest-Neighbour, the proposed approach makes recommendations on other feature descriptors to strengthen an initial profile. More recently, McBurney *et al.* in [9] presented four automatic generators of list of features for software projects, which select English sentences that summarize features from the project documentation.

Our approach, FEAT, has two distinguishing elements w.r.t. these techniques. Firstly, it deals with both software documentation and source code by applying at the same time code and text analysis techniques. Secondly, it uses hierarchical agglomerative clustering assuming that software functions are organized according to a certain (hidden) structure to be automatically discovered.

Mining Source Code. In [10], Linstead *et al.* propose dedicated probabilistic models based on code analysis using *Latent Dirichlet Allocation* to discover features under the form of so-called *topics* (main functions in code). McMillan *et al.* in [11] present a source-code recommendation system for software reuse. Based on a feature model (a notion used in product-line engineering and software variability modelling), the proposed system tries to match the description with relevant features in order to recommends the reuse of existing source code from open-source repositories. [12] proposes to use natural language parsing to automatically extract an ontology from source code. Starting from a lightweight ontology (a.k.a, *concept map*), the authors develop a more formal ontology based on axioms. Using natural language dependencies in sentences which are constructed from identifier names, the method allows one to identify concepts and relations among the sentences. [13] addresses the problem of determining the number of latent concepts (features) in a software system with an empirical method. By constructing clusterings with different topics for a large number of software-systems, the method uses a pair of measures based on source code locality and similarity between topics to assess how well the topic structure identifies related source code units.

Unlike these approaches, FEAT is fully automated and does not require any form of training or any additional modelling activity (e.g., feature modelling). It uses an unsupervised machine learning technique which makes its usage and application much simpler.

The closest approaches to FEAT are those of [14], [15] and [16]. [14] uses clustering and LSI (Latent Semantic Indexing) to assess the similarity between source artefacts and to create clusters according to their similarity. The most relevant terms extracted from the LSI analysis are reused for labelling the clusters. Unlike this approach, FEAT exploits both text mining and code structure analysis to drive the creation of clusters. In Sec.V, we show through experiments that using only lexical proximity is insufficient to create useful clusters for topoi discovery. [15] exploits a sequential combination of information retrieval (IR) technologies to reveal the basic connections between features and elements of the source code and then to refine afterwards these connections through the call graph of the program. Unlike this approach, FEAT exploits a clustering technique to group functions using at the same time both lexical and call graph elements. This gives a powerful combination to create clusters on very distinct projects, containing either meaningful call graph structure and lexical elements. By applying successfully FEAT on 15 unrelated open-source projects, we show that our approach is more versatile. Finally, Moreno *et al.* in [16] automatically extracts concepts from Java source code under the form of *stereotypes* which are low-level patterns about the design intent of a source code artefact. In [17], the same authors propose to generate summaries in natural language of complex code artifacts (i.e., classes and change sets). Unlike this approach, FEAT uses clustering to

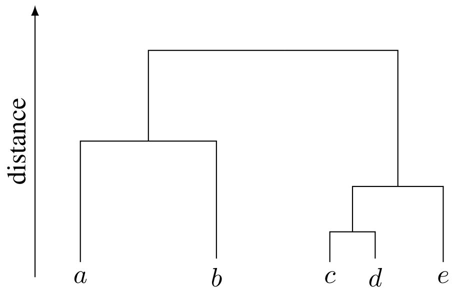


Fig. 1: *Dendrogram* showing the merging steps of HAC. The height of the branches is proportional to the distance of the two merged elements. In the example the first merge involves c and d because they are the closest data points.

mine software projects and applies a hybrid distance to group code functions. It also maximize software modularity to select the number of clusters and performs an automated analysis called entry-point selection to automatically extract topoi.

In summary, our analysis and results show that there is a variety of differences between FEAT and existing software repositories mining techniques.

III. Background

This section presents some background on clustering and graph notions used in FEAT.

A. Clustering

Clustering aims at automatically classify objects into clusters, and as such, it is considered as the most important unsupervised machine learning technique. Objects within the same cluster are expected to be as similar as possible w.r.t. a measure of practical similarity [18]. When clustering is applied to software artefacts (i.e., source code elements, binary code, requirements,…), it is called *software clustering* and it aims at learning regularities or meaningful properties of a software system. Over the last decade, a considerable amount of work has been carried out to solve various software-related problems with clustering including, *information retrieval* [19], [20], *software evolution and maintenance* [21], *reflexion analysis* [22], [23] and *feature extraction* [9].

Hierarchical Agglomerative Clustering (HAC) builds iteratively a tree-like structure by adopting a bottom-up approach to assemble the clusters. HAC results can be visualized through a diagram called *dendrogram*. For example, Fig.1 shows a dendrogram representing the clustering of a data set of 5 elements (from a to e). From the 5 initial singleton-clusters to the final root-cluster, which includes all points, HAC proceeds by merging points and clusters according to a distance measure.

As shown on the dendrogram of Fig.1, without any stopping criterion, HAC ends up with a single cluster. A challenge in clustering is thus to find an appropriate stopping criterion for the process, so that meaningful information can be extracted. Sec.IV-F explains how we handled this problem. A key notion in HAC is the distance between single data points and clusters. Let's provide some formal elements about distances. A function over two vectors $\mathbf{a}$ and $\mathbf{b}$ is called a *distance* if and only if it satisfies the following properties:

1) (symmetry) $d(\mathbf{a}, \mathbf{b}) = d(\mathbf{b}, \mathbf{a})$
2) (positive) $d(\mathbf{a}, \mathbf{b}) \geq 0$, $d(\mathbf{a}, \mathbf{b}) = 0 \Leftrightarrow \mathbf{a} = \mathbf{b}$
3) (triangular inequality) $\forall \mathbf{c}, d(\mathbf{a}, \mathbf{b}) \leq d(\mathbf{a}, \mathbf{c}) + d(\mathbf{c}, \mathbf{b})$

If we want to measure the distance between two sets of data points then we can use centroids. The *centroid* of a set of points [24], denoted by μ, lies at the average position of all points. In an n-dimensions space, the centroid of $C = \{\vec{v_1}, \dots, \vec{v_k}\}$ where each vector $\vec{v_i}$ has coordinates $(x_1, \dots, x_n)$ can be computed using the formula:

$$\mu(C) = \frac{1}{k}\left(\sum_{1}^{k} x_1, \dots, \sum_{1}^{k} x_n\right)$$

It is worth noticing that the centroid of a cluster is neither necessarily an element of the cluster nor an element of the data set. The concept of *medoid* [2] has thus been proposed when a "central" point must come from the data set. Note that in some cases, there may be more than one medoid.

B. Call Graph

A *Call Graph* (CG) is a convenient way of representing the function/method caller-callee relation in a software program. Given a program P composed of a collection of functions $\{f_i\}_{i \in 1 \dots n}$, the CG of P is a graph where each node is associated with a function f_i and there is an arc from f_i to f_j if and only if f_j is called by f_i at some location of the source code. Note that even though f_i may call f_j several times, there is only a single arc from f_i to f_j in the CG.

IV. The FEAT Approach

This section details our approach by first presenting a general overview of FEAT and then describing each of its main components.

A. A General Overview of the Approach

Fig.2 offers a general overview of the whole process. Some elements extracted from the source code of a software project, like comments, literals, names, etc., (noted 1 in Fig.2) are transformed into a general index of words and a call graph (2). Then, the clustering step (3), through a hybrid distance (4), which takes into account both lexical and structural aspects of the code (introduced in Sec.IV-B), will create clusters of closely related code units[1]. This process is interrupted when a cutting criterion based on graph modularity (5 and discussed in Sec.IV-F) is verified. The result of this phase is a set of disjoint clusters (6).

In the second phase (7,8), a set of entry point candidates is selected from every cluster by means of a set of structural properties of the call graph (discussed in Sec.IV-G) through principal component analysis (PCA) and presented (9) as a sorted list of code unit names with related indexes (a topos).

[1] By unit we mean a function or method in any programming language

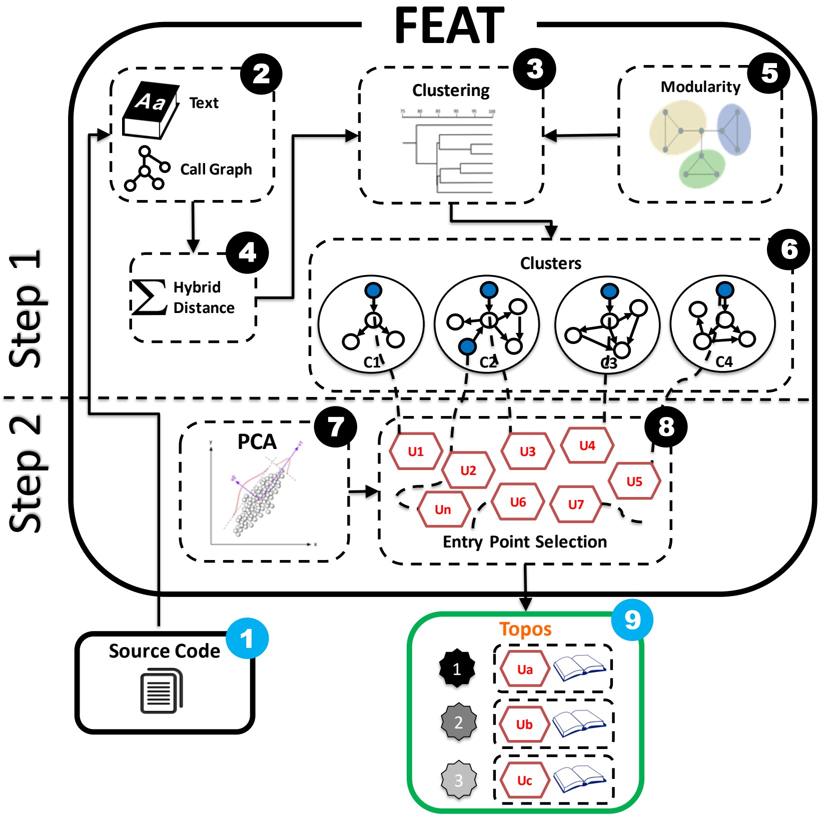


Fig. 2: FEAT process's overview

B. Assessing the lexical distance of code units

When only the source code of a software project is available, a challenge is to evaluate the distance between two or more units. Adopting the original vision of [25], where similarity between documents is characterized by the following assumption: "*words with similar meanings will occur with similar neighbours if enough text material is available*", we adopted a classical way to convert textual documents into vectors and tuned it to match source code peculiarities.

Every unit will be represented through a vector of real numbers whose values are based on the frequency of occurrences of words. Units are said to be represented as *bags of words* [18] to highlight that in this representation the original order of words is not preserved (refer to (2) in Fig.2).

In order to treat units of source code like text documents, they need to undergo some transformations. More details are given below, but before, let us call a *function-document* the source code text of a unit including its comments, variable names, called function names and string literals. Then, the whole set of function-documents can be defined as $\mathrm{D} = \{d_1, \ldots, d_m\}$. All chosen words obtained by scanning a function-document d go into an index V, which is just a set of words. So, given an index V of size n, $\vec{v}(d) \in \mathbb{R}^n$ denotes the vector derived from a function-document d. Elements of vectors represent weights obtained by transforming words' frequency of occurrences using a *weighting scheme* described below.

The distance between two function-documents can be computed with the angular distance[26]. In our context where all vectors' elements are non-negative, distance is in $[0, 1]$ and we name it *lexical distance* to reflect its role within FEAT's approach. Formally,

$$\mathrm{LexicalD}(a, b) = \frac{2}{\pi} \arccos(\mathrm{sim_c}(a, b)) \tag{1}$$

where $\mathrm{sim_c}(a, b)$ is the *cosine similarity*:

$$\mathrm{sim_c}(a, b) = \frac{\vec{v}(a) \cdot \vec{v}(b)}{\|\vec{v}(a)\| \|\vec{v}(b)\|}$$

The transformation of source code into a vector follows the steps shown below. During this process, an index V of words $\{w_1, \ldots, w_n\}$ (alphabetically sorted) is created.

1) *(Parsing step)* For every unit, FEAT parses the source code extracting comments preceding the unit (and optionally comments within the unit), literals, variable names and names of called function;
2) *(Tokenization, stemming and stop words removal step)* By analyzing the elements extracted in the previous step, some compound symbols are disassembled (e.g., "MACOS_print" is decomposed into "MACOS" and "print", "SaveFile" into "Save" and "File"), useless tokens and language-specific keywords are removed (e.g., "and", "if", "else", "while", etc) as they do not bring any value for the identification of topoi, inflected words are brought back to their root (e.g., cars, car's, cars' ⇒ car);

3) *(Weighting step)* In order to counteract some unwanted effects word frequencies need to be transformed [18]. To this end, a composite weighting scheme is used, namely *Term Frequency-Inverse Document Frequency* (tf-idf)[18]. For each extracted word w into a given document d, $\mathrm{tf}(w, d)$ counts the number of occurrences of w in d, while $\mathrm{idf}(w, D) = \log \frac{|D|}{|\{d \in D | w \in d\}|}$ is the logarithm of the inverse fraction of the number of documents in which the word w occurs. Then, using $\text{tf-idf}(w, d, D) = \mathrm{tf}(w, d)\mathrm{idf}(w, D)$, the vector for a function-document is:

$$\vec{v}(d) = [\text{tf-idf}(w_1, d, D), \ldots, \text{tf-idf}(w_n, d, D)] \quad (2)$$

C. Assessing the structural distance of code units

The lexical distance between function-documents given in equation (1) can be complemented by the addition of structural proximity information available in the call graph of the software under examination. Given a (undirected) call graph $\mathrm{CG} = (N, E)$, where N is the set of functions and E is the set of edges representing the caller-callee relationship, the distance between two nodes a and b can be computed by using the length of a shortest path between them. Let $\pi(a, b) = \langle e_1, \ldots e_k \rangle$ be a shortest path between a and b, and $|\pi(a, b)| = k$ be the length of that path (if π does not exist then $|\pi(a, b)| = \infty$) then the distance between a and b is:

$$\mathrm{PathD}(a, b) = \begin{cases} 0 & \text{if } a = b \\ \frac{1-\lambda}{1-\lambda^D} \sum_{i=0}^{k-1} \lambda^i & \text{if } a \neq b \text{ and } |\pi(a, b)| = k \\ 1 & \text{if } |\pi(a, b)| = \infty \end{cases} \quad (3)$$

where D is the graph diameter (the length of the *longest shortest* path) and $\lambda > 1$ is a parameter used to ensure an exponential growing of the distance.

D. Putting the two distances together

Both LexicalD and PathD are proper distance measures satisfying the three required axioms reported in Sec.III.

On the basis of the lexical distance LexicalD and, the distance over nodes in the call graph PathD, we devised a novel hybrid distance. Its objective is to mitigate some unwanted effects that might occur if we used only one of them. For instance, two units sharing similar words, but not connected in the call graph, would be evaluated with high similarity if only LexicalD was used, while, without any kind of structural relationship, they cannot belong to the same feature. Similarly, two close units in the call graph, but without any word in common, should not be clustered together because we assume that elements of a feature should share a common vocabulary.

Our hybrid distance (noted (4) in Fig.2), called FEAT distance, noted FeatD is defined as a linear combination of LexicalD and PathD using a real number α ranging in $[0, 1]$:

$$\mathrm{FeatD}(a, b) = \alpha \mathrm{LexicalD}(a, b) + (1 - \alpha)\mathrm{PathD}(a, b) \quad (4)$$

For any pair of units a and b, we have $\mathrm{FeatD}(a, b) \in [0, 1]$. The external parameter α is used to tune the impact of one distance value over the other. The choice of a value for α depends on some characteristics of the code under analysis like the quality of comments, naming conventions etc. Some concrete examples about this will be provided in Sec.V.

E. Distance Computation over Clusters

The previous definition (FeatD) applies to single function documents but in HAC we need also to compute distances between clusters. In Sec.III-A, we introduced cluster centroid where centroids lie at the average of all points. A simple idea is thus to compute centroids of clusters and to use distance between centroids during HAC. Unfortunately, centroids are defined over points lying in an Euclidean space and not over graphs like our PathD does. So, this section defines *graph medoid*[2], and explains how to compute the distance between two clusters.

Graph Medoids. Let W be a subset of nodes in a call graph $\mathrm{G} = (V, E)$ of n nodes, then the graph medoids of W are the nodes that minimize the length of their paths to the nodes of W while being in the most "central" position. Formally speaking, let $\pi(a, b)$ be a shortest path between a and b and $\mathrm{W} = \{w_1, \ldots, w_m\} \subseteq \mathrm{V}$ be a subset of nodes, then the graph medoids can be computed using:

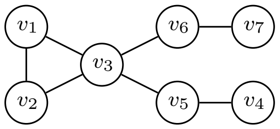
- The matrix $(|\pi(a, b)|)_{n \times n}$ which gathers the pairwise lengths between all the nodes of the graph;
- The function $\sigma(v) = \sum_{j=1}^{m} |\pi(w_j, v)|$;
- The subset $\mathrm{M} = \underset{v \in V}{arg\ min}\ \{\sigma(v)\}$ of V;
- The value $\overline{s} = \frac{1}{m} \min v \in \mathrm{V}\{\sigma(v)\}$;

The set of *graph medoids* of W, noted μ_{G}, is defined as:

$$\mu_{\mathrm{G}}(\mathrm{W}) = \underset{v \in M}{arg\,min} \left(\sum_{j=1}^{m} (|\pi(w_j, v)| - \overline{s})^2 \right) \quad (5)$$

The computation of graph medoids is easier to understand with a simple example.

Example. Let us consider the following graph and its all-shortest-paths matrix.



$$\mathrm{S}_{7,7} = \begin{pmatrix} 0 & 1 & 1 & 3 & 2 & 2 & 3 \\ 1 & 0 & 1 & 3 & 2 & 2 & 3 \\ 1 & 1 & 0 & 2 & 1 & 1 & 2 \\ 3 & 3 & 2 & 0 & 1 & 3 & 4 \\ 2 & 2 & 1 & 1 & 0 & 2 & 3 \\ 2 & 2 & 1 & 3 & 2 & 0 & 1 \\ 3 & 3 & 2 & 4 & 3 & 1 & 0 \end{pmatrix}$$

Assume that we want to compute the graph medoids of $\mathrm{W} = \{v_4, v_7\}$ then the needed steps are the following:

[2] Recall that, unlike centroids, medoids necessarily belong to the set of data points

1) $\sigma(v_i)_{\forall i \in 1..7} = \{3{+}3, 3{+}3, 2{+}2, 0{+}4, 1{+}3, 3{+}1, 4{+}0\}$
   $\sigma(v_i)_{\forall i \in 1..7} = \{6, 6, 4, 4, 4, 4, 4\}$
2) $\mathrm{M} = \{v_3, v_4, v_5, v_6, v_7\}$
3) $\overline{s} = \frac{1}{2} \cdot 4 = 2$
4) $\sum_{j=1}^{m} (|\pi(w_j, v_i)| - \overline{s})^2_{\forall i \in \{3,4,5,6,7\}} = \{0, 8, 2, 2, 8\}$
5) $\mu_{\mathrm{G}}(\mathrm{W}) = \{v_3\}$

Then the set of graph medoids of $\{v_4, v_7\}$ is reduced to a singleton $\mu = \{v_3\}$ which, in this simple example, could have been guessed directly on the graph.

Now, the $\mathrm{FeatD_C}$ distance between two clusters C_i and C_j is defined as follows:

$$\begin{aligned} \mathrm{FeatD_C}(\mathrm{C}_i, \mathrm{C}_j) = \alpha \mathrm{LexicalD}\left(\mu(\mathrm{C}_i), \mu(\mathrm{C}_j)\right) \\ + (1-\alpha)\mathrm{PathD}(\mu_{\mathrm{G}}(\mathrm{C}_i), \mu_{\mathrm{G}}(\mathrm{C}_j)) \end{aligned} \quad (6)$$

Interestingly, the definition, making use of the centroid of a set of vectors and the graph medoid, can be seen as a generalization of Eq.4 since it provides the same result when it is applied to singletons.

F. Maximizing modularity as HAC Cutting Criterion

One of the most effective approaches for detecting communities in networks is based on the optimization of a measure known as modularity [27]. Given a partition of vertices of a graph, modularity Q reflects the concentration of edges within communities compared with random distribution of links between all nodes regardless of communities. In social network research, graph modularity is described as a measure of the division of a network into modules. By definition, modules are graph partitions showing two interesting properties, namely inter-cluster sparsity and intra-cluster density. In our context, we use modularity as a way to obtain clusters containing pieces of the call graph which are highly cohesive. Modularity is defined as:

$$\mathrm{Q} = \frac{1}{2m} \sum_{i,j} \left(A_{i,j} - \frac{k_i k_j}{2m} \right) \delta(c_i, c_j) \quad (7)$$

where A is the adjacency matrix[3], k_i (resp. k_j) is the degree of node i (resp. j), c_i (resp. c_j) is the cluster of i (resp. j), and m is the total number of edges. Function δ is the Kronecker delta: $\delta(c_i, c_j) = 1$ iff $c_i = c_j$ (nodes i,j are in the same cluster), 0 otherwise. In short, the idea of modularity-based cluster detection, studied in the context of social networks [3], is to find partitions which maximize Q. Indeed, high values of modularity (knowing that $\mathrm{Q} \in [-\frac{1}{2}, 1]$) correspond to interesting partitions of a network into communities [27].

In our context we apply modularity in order to obtain densely connected sub-graphs of the call graph whose vertices are units closely related according to our hybrid distance (refer to (5) in Fig.2).

Applying modularity in HAC requires to compute, for every merging step i of the clustering process, Q_i. The cutting criterion for HAC will thus be the partition $C_i = \{c_1, c_2, \ldots, c_n\}$ where Q_i is maximized:

$$\underset{C_i}{arg\,max}\,(\mathrm{Q}_i) \quad (8)$$

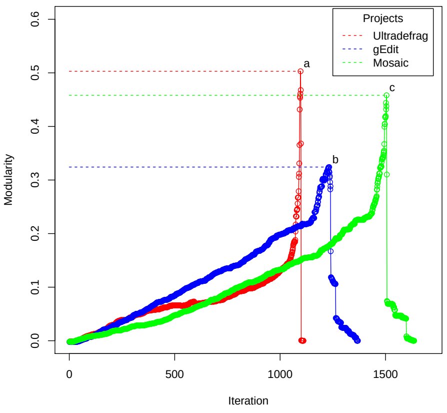


Fig. 3: Modularity values over HAC iterations.

After some experiments with modularity (see Fig.3) we observed the following behaviour of Q_i: at each iteration, modularity gradually grows until it reaches a maximum (points a, b and c in Fig.3). Any further merge leads to a significant decrease of the modularity. Stopping HAC at the maximum value for modularity, while driving the merging process through our FeatD distance, provides us with a set of clusters whose units show high structural regularities and high lexical proximity among them. HAC's output is a set of disjoint clusters where every cluster is made of a set of units, with their names, function-documents, and the related, induced sub-graph of the call graph (see Fig.2 noted (6)) but how can we identify topos?

G. Entry Point Selection

We now introduce the concept of *entry points*. In our view, an entry point is a unit that gives access to the implementation of an observable system functionality, such as, for example, the handler of a menu click in GUI, public methods of an API, etc. Some general considerations are meaningful to identify entry points:

(a) Entry points are called only by a small number of units;
(b) Entry points call many units, either directly or indirectly;
(c) Entry points have long calling chains;
(d) Entry points ends only short call chains;

Based on the subgraph associated to each cluster (see Fig.2 box noted (6)), these considerations can be translated into the six following vertex attributes:

1) Input Degree $\deg^-(v)$: number of incoming arcs;
2) Input Reachability $\mathrm{RI}(v)$: number of paths ending in v;
3) Output degree $\deg^+(v)$: number of outgoing arcs of v;
4) Output Reachability $\mathrm{RO}(v)$: number of paths starting from v;
5) Output Path Length $\mathrm{SO}(v)$: Sum of the lengths of all paths having v as source;
6) Input Path Length $\mathrm{SI}(v)$: Sum of the lengths of all paths having v as destination;

[3] $A_{ij} = 1$ if there exists an edge between vertices i and i and $A_{ij} = 0$ otherwise

Hence a vertex v is now represented as vector $\mathbf{v}$ whose components are the graph attributes listed above:

$$\mathbf{v} = \begin{pmatrix} \deg^+(v) \\ \deg^-(v) \\ \mathrm{RO}(v) \\ \mathrm{RI}(v) \\ \mathrm{SO}(v) \\ \mathrm{SI}(v) \end{pmatrix} \tag{9}$$

The FEAT method exploits Principal Component Analysis (PCA) to deal with the selection of attributes in order to extract a list of meaningfull entry points. PCA is a classical technique applied in data science to deal with high dimensionality [4]. It highlights the most relevant factors in a given vector space: those where the greater variations occur. By keeping only the first m dimensions, which is called low-rank approximation, PCA reduces the data dimensionality while retaining most of the data information, i.e. the variation in the data. Low rank approximation has several advantages: removal of "noisy", uninteresting dimensions, faster computation, ability of making data lying in high dimensional space displayable in 2-D or 3-D etc. On a cluster by cluster basis, PCA will help us to select the most promising graph attributes cited above without having to choose one (or a subset) of them a priori. So, thanks to PCA we now have code units, represented as vectors, projected into a space where their differences, respect to the set of properties which characterize an entry point, are highlighted to the maximum possible extent.

Our solution to identify which units can serve as entry-points is to create an artificial vector, representing the ideal entry point in a given cluster. Then, it is sufficient to compute its similarity against each other unit in the cluster. Every component of the vector relates to the six attributes mentioned above. According to our assumption, entry points should have high *out*-values and low *in*-values. Then given a cluster $\mathrm{C} = \{v_1, \cdots, v_n\}$, the query vector representing an ideal entry-point can be defined as:

$$\mathbf{q_C} = \begin{pmatrix} \underset{v\in C}{max}\{\deg^+(v)\} \\ \underset{v\in C}{min}\{\deg^-(v)\} \\ \underset{v\in C}{max}\{\mathrm{RO}(v)\} \\ \underset{v\in C}{min}\{\mathrm{RI}(v)\} \\ \underset{v\in C}{max}\{\mathrm{SO}(v)\} \\ \underset{v\in C}{min}\{\mathrm{SI}(v)\} \end{pmatrix} \tag{10}$$

Finally, the Euclidean distance of every unit's vector $\mathbf{v}$ respect to $\mathbf{q_C}$ in the PCA space is the key used to rank all the units in a cluster.

Example. Let us consider a graph G_1 represented in Fig.4 associated to a cluster extracted by HAC. By running PCA and selecting the first $m = 2$ components (covering in this case $\approx 85\%$ of the entire variance), we get the ordered list shown in Tab.I. Node v_2 is ranked first and indeed it was expected to be an entry point in G_1 while v_7, which is a dead-end of G_1, has been appropriately ranked last.

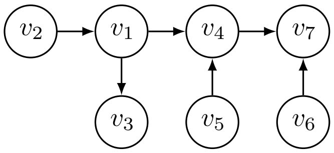


Fig. 4: Graph G_1 associated to the cluster of the example

Rank	Vertex	Distance
1	v_2	0.215
2	v_1	0.482
3	v_5	0.499
4	v_6	0.733
5	v_4	0.804
6	v_3	1.101
7	v_7	1.143

TABLE I: Sorted list of the vertices of G_1 (Fig.4)

Topoi are supposed to be presented to validation engineers but they can also be automatically extracted on the basis of the ranking. The criterion we adopted is the following: in a cluster $\mathrm{C} = \{\mathbf{v}_1, \cdots, \mathbf{v}_n\}$, all code units whose distance respect to the query vector is not larger than a given threshold β of the cases are classified as entry points. Formally, it has to satisfy the following: $P(\mathrm{X} \leq x) = \beta$ where X is the set of distances.

To recap, the entry point selection phase creates a set of topos, one for each cluster. Every topos, made of a sorted list of code units with their related index of words, represents a summary of the main capabilities of the system under analysis. Tab.II shows an example of the content of a topos extracted from a text editor called GEDIT(more details in Sec.V). Beside some common words, the two indexes show some bold-faced words which are useful to relate topos' content with GEDIT's functionality.

Unit	**Index**
`_gedit_cmd_file_open()`	**chooser**, **cmd**, connect, data, debug, default, **dialog**, **document**, fail, **file**, **folder**, gedit, gtk, location, modal, **open**, title, window
`_gedit_cmd_edit_copy()`	check, **clipboard**, **cmd**, **copy**, debug, **edit**, fail, focus, gedit, grab, gtk, return,type, view, widget, window

TABLE II: Part of a topos obtained from the analysis of GEDIT

V. Experimental Evaluation

This section gives an overview of our implementation and presents our experimental results.

A. Implementation

We implemented FEAT on top of a software testing platform called CRYSTAL, which is based on OSGi (Open Services Gateway initiative[4]) and BPMN (Business Process Modeling Notation[5]). Data persistence is based both on MySQL database

[4] OSGi Alliance www.osgi.org

[5] Object Management Group (www.bpmn.org)

and neo4j graph database. CRYSTAL is designed for enhancing reuse of software components and to create distributed architectures. All experiments were run on an Intel dual core i7-4510U CPU with 8GB RAM.

B. Experimental Subjects

We selected 15 Open Source C software projects from *SourceForge* having different sizes and application domains. Tab.III reports for each project number of lines of codes (**LOC**), number of code-units (**#Unit**), number of files (**#File**), dictionary size (**Dict**) and call graph density (ρ in ‰). Overall, we have more than $1MLOC$, more than $20K$ units with a dictionary of $11K$ words.

A finer-grained analysis of FEAT is presented through two projects:

- **Hexadecimal Viewer** (HEXDUMP), project 2 in Tab.III. It is a hexadecimal viewer, i.e. an application that displays binary data contained in files as a readable sequence of codes. The project contains more than 15 KLOC, 254 units, 13 C files with a dictionary size of 723 words. The corresponding call graph has 254 vertices (units) and 293 edges, with a density of $\rho = 9.12$‰.
- **GNU Editor v3.20** (GEDIT), project 7 in Tab.III. It is the default text editor of GNOME desktop environment. This project accounts for 42 KLOC, $1,370$ units, 59 C files with a dictionary of 931 words. We have a call graph of $1,370$ units, $2,058$ edges and a density of $\rho = 2.19$‰.

For HEXDUMP and GEDIT, we manually created an oracle with its list of entry points according to the following procedure: *(i)* we looked at the user's manual and identified the topoi, *(ii)* we inspected the source code searching for the entry points of those topoi (e.g., in desktop applications, we have usually event handler functions of either menus or other kind of GUI elements). To ease the automatic extraction of topoi we set the threshold β to 25% (see Sec.IV-G).

C. Research Questions

Generally speaking, the goal of our experiments was to assess the effectiveness of FEAT in extracting topoi. We compared FEAT with the two following baselines:

- **Random** A random classifier which selects at random a subset of entry points from a set of units. Results are reported on an average of 1,000 runs ;
- **No-HAC** A version of FEAT with the HAC step (clustering) deactivated. This version was mainly created to evaluate the benefits of clustering in the topoi extraction process.

Our experiments investigated the four following questions:

- **RQ1** How does FEAT compare with the two baselines when they are used to automatically extract topoi?
- **RQ2** How is the performance of FEAT impacted when α varies (α is the parameter used in the linear combination of our hybrid distance)?
- **RQ3** How is the performance of FEAT impacted when extracting various textual elements (i.e, source code, comments)?
- **RQ4** How are software project characteristics (#units, LOC, Dict, call graph density ρ) correlated with FEAT's running time?

#	Project	LOC	#Unit	#File	Dict	ρ (‰)
1	Linux FS EXT2	8,445	180	14	197	20.11
2	Hexadec. Viewer	12,053	254	13	723	9.12
3	GNU bc Calculator 1.06	12,851	215	20	597	20.47
4	Intel Ethernet Drivers and Util.	30,499	581	16	726	6.29
5	Ultradefrag v7.0	34,637	1,112	74	543	5.46
6	Zint Barcode Generator v2.3.0	38,095	345	43	196	13.43
7	GNU Editor v3.2	42,718	1,370	59	931	2.19
8	bash v1.0	70,955	1,477	128	885	2.75
9	Linux IPv4	84,606	2,216	127	1204	1.11
10	x3270 Terminal Emulator v3.5	91,449	1,881	136	964	2.51
11	Mem. Mgt Mod. Linux v4.9.8	93,888	2,883	102	1759	1.03
12	NCSA Mosaic Web Browser v2.7	98,715	1,637	134	980	2.60
13	Linux FS EXT2 Utilities	126,488	2,544	309	849	1.79
14	Vim Txt Editv7.0	134,082	2,116	56	1057	3.14
15	CLIPS core v6.3	171,913	2,794	164	886	2.22
	Total:	1,051,394	21,605	1,395	11,546	—

TABLE III: The 15 Open Source C software projects

D. Results and Analysis

In this section, we analyze the results of FEAT on HEXDUMP and GEDIT using their respective oracles. We first introduce the metrics that are widely used in machine learning classifiers' evaluation, namely *accuracy*, *precision* and *recall* [18]. Using the topoi automatically mined by FEAT and the topoi that we manually extracted from the available documentation (and referred to as "oracle"), we define *true positives (tp)* as the units which are correctly classified as entry points, *false positives (fp)* as the units which are incorrectly classified as entry points, *true negatives (tn)* as the units which are correctly not classified as entry points and finally *false negatives (fn)* as the units which are incorrectly not classified as entry points:

1) *Accuracy* is the percentage of entry-points which are correctly classified:
$$\mathrm{A} = \frac{tp + tn}{tp + fp + fn + tn}$$
2) *Precision* is the percentage of retrieved entry-points that are relevant:
$$\mathrm{P} = \frac{tp}{tp + fp}$$
3) *Recall* is the percentage of relevant entry-points that are retrieved:
$$\mathrm{R} = \frac{tp}{tp + fn}$$

For all these metrics, the higher, the better.

To answer **RQ1** and **RQ2**, we ran an experiment where $\alpha \in [0, 1]$ varies from 0 (case where only PathD is applied)

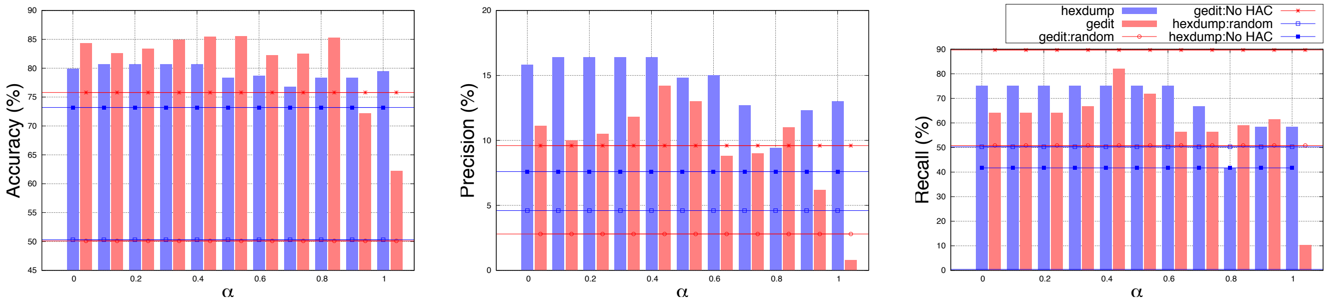


Fig. 5: FEAT effectiveness.

to 1 (case where only $\mathrm{LexicalD}$ is applied) by sampling on every tenth 0.1. Fig.5 shows accuracy, precision and recall of FEAT on HEXDUMP and GEDIT. We also report the results of the two baselines (**Random** and **No-HAC**).

1) Results and Analysis for **RQ1***:* First, from figure 5, we observe that FEAT drastically outperforms the random baseline. This means that FEAT is able to extract regularities from source code with its hybrid distance measure. Second, using $\alpha = 0.5$ and looking at the second baseline (where FEAT is launched without clustering), we observe that FEAT reaches an accuracy of 85% (resp. 80%) for GEDIT (resp. HEXDUMP), whereas the baselines respectively do not exceed 50% and 75% for the two projects. Using the "precision" metric, we draw the same conclusion that FEAT outperforms both baselines. Using the "recall" metric, we observe that **No-HAC** reaches a high recall value. This is explained by the rough over-approximation we get when no clustering is used, at the expense of precision loss (i.e., large number of false positives).

2) Results and Analysis for **RQ2***:* Let us take a close look at $\alpha = 0$ and $\alpha = 1$ in figure 5. When $\alpha = 0$, FEAT is driven by the call graph and only the structural distance PathD. Here, the results are fairly robust with an acceptable level of accuracy, precision and recall. However, when $\alpha = 1$, FEAT loses its effectiveness as it is driven only by lexical distance measure (see GEDIT project). That being said, combining PathD and $\mathrm{LexicalD}$ distances can significantly improve the performance of FEAT. Our experiments on GEDIT and HEXDUMP highlight clearly $\alpha \approx 0.5$ as a good balance between PathD and $\mathrm{LexicalD}$ distances.

3) Results and Analysis for **RQ3***:* Here the goal was to evaluate the impact of using only source code and/or comments in FEAT. The meaning of symbols in Fig.6 is:

- **Code:** Function and variable names as well as literals are extracted from source code;
- **Comments:** Only comments are extracted;
- **All:** Both code and comments are extracted;

Despite some irregularities shown in the graphs of Fig.6, we notice that HEXDUMP does not benefit from the addition of more textual elements (i.e., comments). Indeed "All" reaches its best accuracy with values of $\alpha \approx 0.1$ which means that almost only the graphical part of the distance is relevant. "Comments" and "Code" show similar performance. Providing higher values of α, such that $\alpha > 0.6$, leads to a greater importance of the lexical distance $\mathrm{LexicalD}$. In this case, the accuracy decreases. This is due to some poor adopted naming convention and also the low-informative quality of comments. We observe that 1) some comments are even wrong (their content does not correspond to the actual implementation) and 2) some comments are misspelled with a mixture of English and German.

GEDIT shows good performance in terms of accuracy with "Comments" and "All" when $0.6 < \alpha < 0.8$. In fact, GEDIT adopts consistent naming convention and good-quality comments. We thus observe that α needs to be adjusted according to the coding style adopted in the project. Both projects reach very good accuracy: above 85% for HEXDUMP and above 90% for GEDIT.

4) Results and Analysis for **RQ4***:* Our last experiment involves all the 15 projects listed in Tab.III. The objective is to measure the running time of FEAT w.r.t. different characteristics. Fig.7 (y-axis is logarithmic) reports the impact of several metrics describing the software projects (i.e., LOC, #unit, dictionary size and call graph density ρ). Total time is composed of two elements: preprocessing and clustering time (see the legend in Fig.7). Preprocessing time includes the call graph construction and the textual extraction. The remaining part is the execution time needed by clustering. Let us start with Fig.7.(a), where we report time in seconds w.r.t. the number of units. Notice that the time needed to extract topoi is heavily impacted by the number of units. This strong positive correlation is explained by the fact that the clustering process is the most costly step in FEAT whose complexity is $O(n^2 \log(n))$. Our main observation is that FEAT scales well on projects with up to $1K$ units with a running time less than 1 min. FEAT is still efficient on projects up to $2.2K$ units with a time not exceeding 12 min. Starting from $2.5K$ units, the clustering step becomes demanding for FEAT. For instance, it needs respectively more than 58 min., 23 min. and 75 min. to deal with projects 11, 13 and 15 of Tab.III. In Fig.7.(b), we report running time over LOC. We observe that for medium scale projects (i.e., up to $100K$ LOC) the preprocessing step can be quite expensive which can be explained by the fact that the time needed to parse source code is negatively impacted by the number of LOCs. Similarly, Fig.7.(c) shows an identical behaviour with the impact of dictionary size on the preprocessing step. Fig.7.(d) reports the relationship between running time and call graph density. Here, we have three outliers ($\rho = 1$‰, $\rho = 1.8$‰ and $\rho = 2.3$‰), corresponding to projects 11, 13 and 15 in

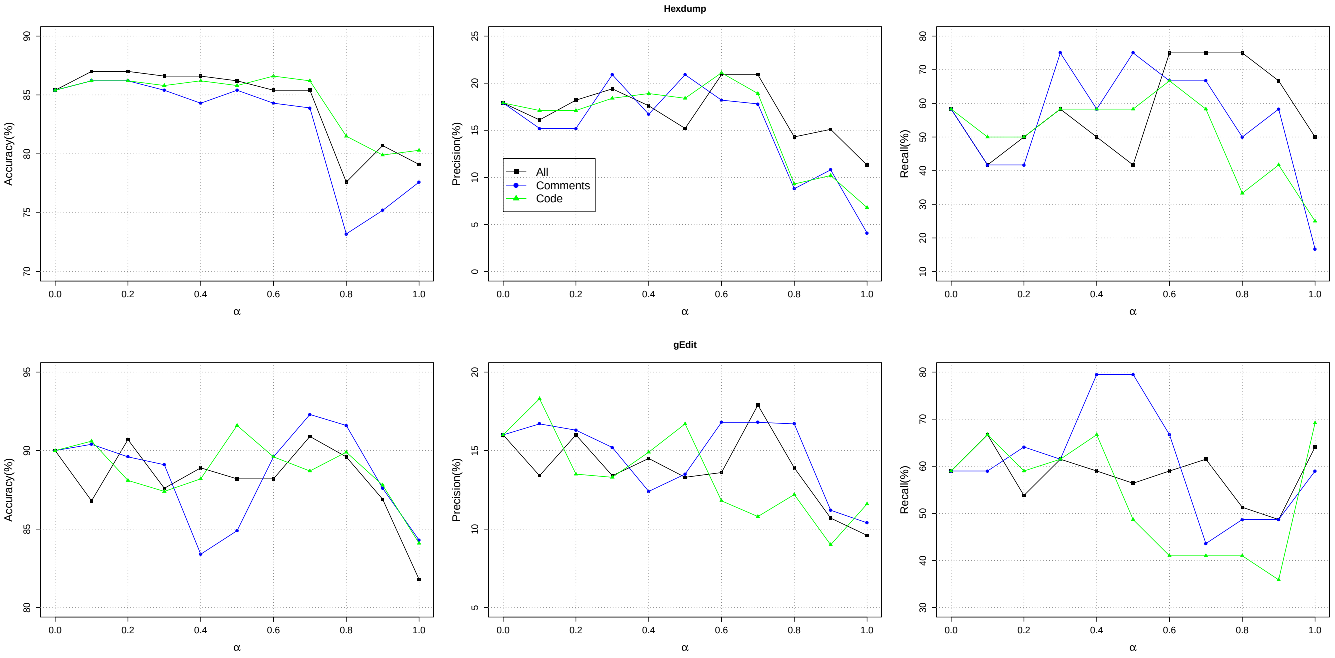


Fig. 6: Impact of textual elements on FEAT's performance while α varies between 0 and 1 (Q3)

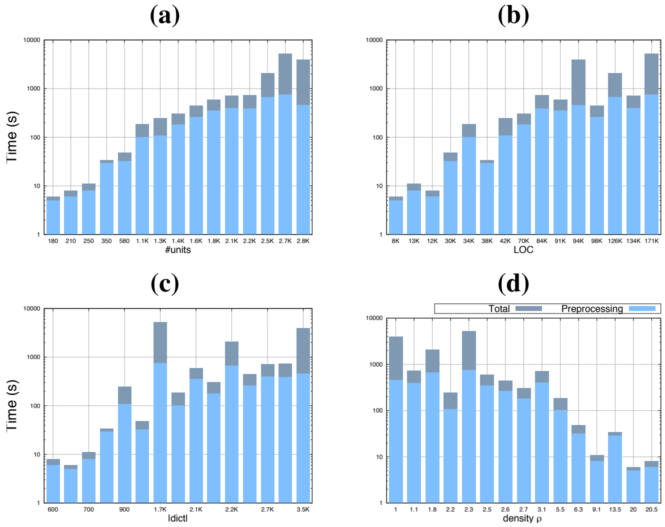


Fig. 7: FEAT running time (Q4)

Tab.III. Besides the three outliers, we can observe a negative correlation between the density of call graphs and the time needed to extract topoi.

E. Threats to validity

This section briefly discusses some threats to validity of our experimental evaluation.

- The value selected for α has a great impact on the accuracy, precision and recall. Choosing an appropriate value for α is a key point of our approach and, unfortunately, there is no theoretical result helping us deciding beforehand the best value for this input parameter. An approach would have been thus to run FEAT on more than two projects for which we had an oracle. Then, it could have been possible to decide on an appropriate value of α based on the experimental results. This was considered as a too demanding effort which would have required to understand the code of other projects ;
- The creation of oracles needed for the automatic assessment of FEAT can be biased by the author's knowledge of the experimental evaluation. In order to tame this risk, we have selected two software projects on which we ignored everything beforehand and have manually extracted an oracle for both of them. Of course, one could object that knowing that HEXDUMP is a hexadecimal converter and that GEDIT was a text editor helped us deciding of the extracted features, but it is important to stress that none of the authors knew the code of the project or results of FEAT beforehand ;
- The evaluation of FEAT is based on the comparison of runtime w.r.t. different project characteristics. It would also have strengthened the evaluation to perform a controlled experiment in order to evaluate the usefulness of topoi in program comprehension. We could have set-up a controlled experiment where half of the participants try to understand a software project with FEAT and another half without FEAT. Some measurements on the time needed to find software features could then have been reported and analyzed ;

VI. CONCLUSION

Topoi are concrete and useful representations of software features. When a software system has evolved over a long period of time, topoi extraction provides validation engineers an updated view on the system features which is a valuable asset to get more reliable systems. To address the challenge of topoi extraction, we presented FEAT a two-steps method based on hierarchical agglomerative clustering (HAC) and entry-points selection. In FEAT, HAC exploits a novel hybrid distance combining lexical and structural elements, and graph medoids which extend the concept of centroid to set of graph nodes. We addressed the so-called cutting criterion challenging aspect of HAC by maximizing modularity in order to achieve the best partition of clusters in terms of elements' cohesion. Finally, we defined a criterion based on principal component analysis to select entry-points as topoi representatives.

By using FEAT on 15 open-source projects, amounting to more than 1M LOC in total, we showed that FEAT is a feasible approach for automatically discovering program topoi directly from source code. This paper showed that HAC can deal with medium-sized software projects (more than 100,000 LOC) in a reasonable amount of time.

As further work, improving the accuracy of entry-points selection could be achieved by using other code structure representations. So far, we focused on using the call graph but some useful insights might come from dataflow representations. It would be interesting to investigate how a blend of attributes from both call graph and the program dependency graph can impact the quality of entry-points selection. We also believe that improving the visualization of the results provided by FEAT by providing an adequate user interface to display and analyse topoi would ease the adoption of the tool and maximize its impact.

References

[1] K. E. Wiegers, *Software Requirements*, 2nd ed. Redmond, WA, USA: Microsoft Press, 2003.

[2] V. Estivill-Castrol and A. T. Murray, "Discovering associations in spatial data — an efficient medoid based approach," in *Proc. of 2nd Pacific-Asia Conference on Research and Development in Knowledge Discovery and Data Mining (PAKDD'98), Melbourne, Australia*, X. Wu, R. Kotagiri, and K. B. Korb, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 110–121.

[3] L. Donetti and M. A. Muñoz, "Detecting network communities: a new systematic and efficient algorithm." *Journal of Statistical Mechanics: Theory and Experiment*, vol. 10, p. 8, 2004.

[4] J. Shlens, "A tutorial on principal component analysis," *Internet Article*, pp. 1–13, 2005.

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

[6] K. Chen and V. Rajlich, "Case study of feature location using dependence graph," in *Program Comprehension, 2000. Proceedings. IWPC 2000. 8th International Workshop on*, 2000, pp. 241–247.

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

[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 *33rd Int. Conf. on Sof. Eng. (ICSE'11)*, 2011, pp. 181–190.

[9] 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.

[10] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi, "Mining concepts from code with probabilistic topic models," in *Proc. of Automated Software Eng.*, Apr. 2007, p. 461.

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

[12] S. L. Abebe and P. Tonella, "Extraction of domain concepts from the source code," *Science of Computer Programming*, vol. 98, pp. 680–706, 2015.

[13] 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.

[14] 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.

[15] W. Z. W. Zhao, L. Z. L. Zhang, Y. L. Y. Liu, J. S. J. Sun, and F. Y. F. Yang, "SNIAFL: Towards a static non-interactive approach to feature location," in *Proc. of Int. Conf. on Soft. Eng. (ICSE'04)*, vol. 15, no. 2, 2004, pp. 293–303.

[16] L. Moreno and A. Marcus, "Jstereocode: Automatically identifying method and class stereotypes in java code," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012. New York, NY, USA: ACM, 2012, pp. 358–361. [Online]. Available: http://doi.acm.org/10.1145/2351676.2351747

[17] L. Moreno, "Summarization of complex software artifacts," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014. New York, NY, USA: ACM, 2014, pp. 654–657. [Online]. Available: http://doi.acm.org/10.1145/2591062.2591096

[18] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. New York, NY, USA: Cambridge University Press, 2008.

[19] M. Shtern and V. Tzerpos, "Clustering methodologies for software engineering," *Advances in Software Engineering*, pp. 1–18, 2012.

[20] A. Marcus, A. Sergeyev, V. Rajlich, and J. Maletic, "An information retrieval approach to concept location in source code," in *Proc. of the 11th Working Conference on Reverse Engineering*, 2004, pp. 214–223.

[21] A. Kuhn, S. Ducasse, and T. Gîrba, "Enriching reverse engineering with semantic clustering," in *Proc. of Working Conf. on Reverse Engineering (WCRE'05)*, no. 3, 2005, pp. 133–142.

[22] G. Murphy, D. Notkin, and K. Sullivan, "Software reflexion models: bridging the gap between source and high-level models," in *Proc. of 3rd ACM Symp. on Foundations of Soft. Eng. (FSE'95)*, Oct. 1995, pp. 18–27.

[23] A. Christl, R. Koschke, and M. Storey, "Automated clustering to support the reflexion method," *Information and Software Technology*, vol. 49, no. 3, pp. 255–274, 2007.

[24] É. D. Taillard, "Heuristic methods for large centroid clustering problems," *J. Heuristics*, vol. 9, no. 1, pp. 51–73, 2003. [Online]. Available: http://dx.doi.org/10.1023/A:1021841728075

[25] H. Schütze and J. Pedersen, "Information retrieval based on word senses," in *Proceedings of the 4th Annual Symposium on Document Analysis and Information Retrieval*, Las Vegas, USA, 1995, pp. 161–175.

[26] M. M. Deza and E. Deza, *Encyclopedia of Distances*. Springer Berlin Heidelberg, 2009.

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