Unveiling source code latent knowledge: discovering program topoi

Carlo Ieva

To cite this version:

Carlo Ieva. Unveiling source code latent knowledge: discovering program topoi. Programming Languages [cs.PL]. Université Montpellier, 2018. English. NNT : 2018MONTS024 . tel-01974511

HAL Id: tel-01974511

https://theses.hal.science/tel-01974511v1

Submitted on 8 Jan 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE POUR OBTENIR LE GRADE DE DOCTEUR DE L'UNIVERSITÉ DE MONTPELLIER

En Informatique

École doctorale : Information, Structures, Systèmes

Unité de recherche : LIRMM

Révéler le contenu latent du code source
A la découverte des topoi de programme

Unveiling Source Code Latent Knowledge
Discovering Program Topoi

Présentée par Carlo Ieva
Le 23 Novembre 2018

Sous la direction de Prof. Souhila Kaci

Devant le jury composé de

Michel Rueher, Professeur, Université Côte d'Azur, I3S, Nice	Président du jury
Yves Le Traon, Professeur, Université du Luxembourg, Luxembourg	Rapporteur
Lakhdar Sais, Professeur, Université d'Artois, CRIL, Lens	Rapporteur
Jérôme Azé, Professeur, Université de Montpellier, LIRMM, Montpellier	Examinateur
Roberto Di Cosmo, Professeur, Université Paris Diderot, INRIA	Examinateur
Samir Loudni, Maître de Conférences, Université de Caen-Normandie, GREYC, Caen	Examinateur
Clémentine Nebut, Maître de Conférences, Université de Montpellier, LIRMM, Montpellier	Examinatrice
Arnaud Gotlieb, Chief Research Scientist, Simula Research Lab., Lysaker, Norway	Co-encadrant
Nadjib Lazaar, Maître de Conférences, Université de Montpellier, LIRMM, Montpellier	Co-encadrant

UNIVERSITÉ
DE MONTPELLIER

I wish to dedicate this dissertation to my family who always trusts me.

Acknowledgements

"You will become clever through your mistakes" reads a German proverb and doing a Ph.D. has surely to do with making mistakes. Whether I have become a clever person, I'll let the reader be the judge, nonetheless, I learned a great deal. If I was asked to say what I learned in just one sentence then I would answer: a new way to look at things; this is for me what a Ph.D. really teaches you.

The path leading up to this point has not always been straightforward, learning is not a painless experience, and now is the moment to thank all those who helped me along the way. Thanks to my supervisors: Souhila Kaci, Arnaud Gotlieb and, Nadjib Lazaar, who patiently provided me an invaluable help. Thanks to Simula for creating the right conditions to make my Ph.D. possible. I would also like to express my very great appreciation to the reviewers Yves Le Traon and Lakhdar Sais who accepted to examine my thesis providing me their valuable feedback and thanks also to all jury members for being part of the committee.

Abstract

During the development of long lifespan software systems, specification documents can become outdated or can even disappear due to the turnover of software developers. Implementing new software releases or checking whether some user requirements are still valid thus becomes challenging. The only reliable development artifact in this context is source code but understanding source code of large projects is a time- and effort- consuming activity. This challenging problem can be addressed by extracting high-level (observable) capabilities of software systems. By automatically mining the source code and the available source-level documentation, it becomes possible to provide a significant help to the software developer in his/her program comprehension task.

This thesis proposes a new method and a tool, called FEAT (FEature As Topoi), to address this problem. Our approach automatically extracts *program topoi* from source code analysis by using a three steps process: First, FEAT creates a model of a software system capturing both structural and semantic elements of the source code, augmented with code-level comments; Second, it creates groups of closely related functions through hierarchical agglomerative clustering; Third, within the context of every cluster, functions are ranked and selected, according to some structural properties, in order to form program topoi.

The contributions of the thesis is three-fold:

1. The notion of program topoi is introduced and discussed from a theoretical standpoint with respect to other notions used in program comprehension ;

2. At the core of the clustering method used in FEAT, we propose a new hybrid distance combining both semantic and structural elements automatically extracted from source code and comments. This distance is parametrized and the impact of the parameter is strongly assessed through a deep experimental evaluation ;

3. Our tool FEAT has been assessed in collaboration with *Software Heritage* (SH), a large-scale ambitious initiative whose aim is to collect, preserve and, share all publicly available source code on earth. We performed a large

experimental evaluation of FEAT on 600 open source projects of SH, coming from various domains and amounting to more than 25 MLOC (million lines of code).

Our results show that FEAT can handle projects of size up to 4,000 functions and several hundreds of files, which opens the door for its large-scale adoption for program comprehension.

Résumé de la thèse

Introduction

Le développement de projets open source à grande échelle implique de nombreux développeurs distincts qui contribuent à la création de référentiels de code volumineux. À titre d'exemple, la version de juillet 2017 du noyau Linux (version 4.12), qui représente près de 20 lignes MLOC (lignes de code), a demandé l'effort de 329 développeurs, marquant une croissance de 1 MLOC par rapport à la version précédente. Ces chiffres montrent que, lorsqu'un nouveau développeur souhaite devenir un contributeur, il fait face au problème de la compréhension d'une énorme quantité de code, organisée sous la forme d'un ensemble non classifié de fichiers et de fonctions.

Organiser le code de manière plus abstraite, plus proche de l'homme, est une tentative qui a suscité l'intérêt de la communauté du génie logiciel. Malheureusement, il n'existe pas de recette miracle ou bien d'outil connu pouvant apporter une aide concrète dans la gestion de grands bases de code.

Nous proposons une approche efficace à ce problème en extrayant automatiquement des *topoi de programmes*, c'est à dire des listes ordonnées de noms de fonctions associés à un index de mots pertinents. Comment se passe le tri? Notre approche, nommée FEAT, ne considère pas toutes les fonctions comme égales: certaines d'entre elles sont considérées comme une passerelle vers la compréhension de capacités de haut niveau observables d'un programme. Nous appelons ces fonctions spéciales *points d'entrée* et le critère de tri est basé sur la distance entre les fonctions du programme et les points d'entrée. Notre approche peut être résumé selon ses trois étapes principales:

1. *Preprocessing.* Le code source, avec ses commentaires, est analysé pour générer, pour chaque unité de code (un langage procédural ou une méthode orientée objet), un document textuel correspondant. En outre, une représentation graphique de la relation appelant-appelé (graphe d'appel) est également créée à cette étape.

2. *Clustering.* Les unités de code sont regroupées au moyen d'une classification par clustering hiérarchique par agglomération (HAC).

3. *Sélection du point d'entrée.* Dans le contexte de chaque cluster, les unités de code sont classées et celles placées à des positions plus élevées constitueront un topos de programme.

La contribution de cette thèse est triple:

1. FEAT est une nouvelle approche entièrement automatisée pour l'extraction de topoi de programme, basée sur le regroupement d'unités directement à partir du code source. Pour exploiter HAC, nous proposons une distance hybride originale combinant des éléments structurels et sémantiques du code source. HAC requiert la sélection d'une partition parmi toutes celles produites tout au long du processus de regroupement. Notre approche utilise un critère hybride basé sur la *graph modularity* [17] et la cohérence textuelle [21] pour sélectionner automatiquement le paramètre approprié.

2. Des groupes d'unités de code doivent être analysés pour extraire le programme topoi. Nous définissons un ensemble d'éléments structurels obtenus à partir du code source et les utilisons pour créer une représentation alternative de clusters d'unités de code. L'analyse en composantes principales, qui permet de traiter des données multidimensionnelles, nous permet de mesurer la distance entre les unités de code et le point d'entrée idéal. Cette distance est la base du classement des unités de code présenté aux utilisateurs finaux.

3. Nous avons implémenté FEAT comme une plate-forme d'analyse logicielle polyvalente et réalisé une étude expérimentale sur une base ouverte de 600 projets logiciels. Au cours de l'évaluation, nous avons analysé FEAT sous plusieurs angles: l'étape de mise en grappe, l'efficacité de la découverte de topoi et l'évolutivité de l'approche.

Travaux connexes

Nos travaux s'inscrivent dans le domaine de la compréhension de programmes en se concentrant principalement sur *l'extraction de fonctionnalités* [68, 11, 45]. L'extraction de fonctionnalités vise à découvrir automatiquement les principales fonctionnalités d'un logiciel en analysant son code source ainsi que d'autres artefacts. L'extraction de fonctionnalités est différente de *la localisation de fonctionnalités*, dont l'objectif est de localiser où et comment des fonctionnalités données sont implémentées [68]. La localisation nécessite que l'utilisateur fournisse une requête d'entrée dans laquelle la fonctionnalité recherchée est déjà connue, tan-

dis que l'extraction de la fonctionnalité tente de la découvrir automatiquement. Depuis plusieurs années, l'extraction de fonctionnalités de logiciels est considérée comme une activité dominante dans la compréghension de programmes. Cependant, nous pouvons faire la distinction entre les approches de compréhension de programmes qui traitent la documentation des logiciels et celles qui traitent directement le code source.

Compréhension de programmes basée sur la documentation de logiciel

Dans [18], les techniques d'exploration de texte et de clustering sont utilisées pour extraire des descripteurs de fonctionnalités à partir des besoins de l'utilisateur conservés dans des référentiels de logiciels. En combinant l'extraction de règles d'association et le k-plus proche-voisin, l'approche en question propose des recommandations sur d'autres descripteurs afin de renforcer un profil initial. McBurney et al. [48] ont récemment présenté quatre générateurs automatiques de liste de fonctionnalités des projets logiciels. Ils sélectionnent des phrases en anglais de la documentation du projet résumant les fonctionnalités.

Compréhension de programmes basée sur le Code source. [41] propose des modèles probabilistes basés sur l'analyse de code en utilisant *Latent Dirichlet Allocation* pour découvrir des fonctionnalités sous la forme de *topics* (fonctions principales dans le code). [50] présente un système de recommandation de code source pour la réutilisation de logiciels. Basé sur un modèle de fonctionnalité (une notion utilisée dans l'ingénierie de ligne de produit et la modélisation de la variabilité logicielle), le système proposé tente de faire correspondre la description aux fonctionnalités pertinentes afin de recommander la réutilisation du code source existant à partir de référentiels de code source libre. [2] propose une analyse syntaxique en langage naturel pour extraire automatiquement une ontologie du code source. Partant d'une ontologie légère (it concept map), les auteurs développent une ontologie plus formelle basée sur des axiomes.

A l'inverse, FEAT est entièrement automatisé et ne nécessite aucune forme de d'entrainement de jeu de données ni aucune activité de modélisation supplémentaire. FEAT utilise une technique d'apprentissage automatique non supervisée, ce qui simplifie grandement son utilisation et son application.

[37] utilise le clustering et le LSI (Latent Semantic Indexing) pour évaluer la similarité entre des parties du code source. Les termes les plus pertinents extraits de l'analyse LSI sont réutilisés pour l'étiquetage des clusters. FEAT exploite à la place l'exploration de texte et l'analyse de la structure de code pour guider la création de clusters.

Comparant à ces techniques, FEAT a deux éléments distinctifs. Premièrement, FEAT traite à la fois la documentation du logiciel et le code source en appliquant simultanément des techniques d'analyse de code et de texte. Deuxièmement, FEAT utilise HAC en supposant que les fonctions logicielles sont organi-

sées selon une certaine structure (cachée) pouvant être automatiquement découverte.

Contexte

Clustering appliqué au logiciel

Les méthodologies de clustering appliquées au logiciel créent un groupe d'entités, telles que des classes, des fonctions, etc. L'objectif de ces dernières est de faciliter la compréhension de la structure d'un système logiciel large et complexe [73].

Appliquer du clustering au logiciel nécessite l'identification des entités qui font l'objet du groupement. Plusieurs artefacts peuvent être choisis, mais le plus populaire est le code source [51]. La sélection des entités dépend fortement de l'objectif de l'approche à utiliser en clustering. Pour la restructuration de programmes à un niveau plus fin, les instructions d'appel de fonction sont choisies comme entités [84], tandis que pour des problèmes de conception, les entités [6] sont souvent des modules logiciels mais également des classes ou des routines.

L'extraction de faits à partir du code source peut s'effectuer selon deux approches conceptuelles différentes: *structural et sémantique*. Les approches basées sur la structure reposent sur des relations statiques entre entités: références de variable, appels de procédure, héritage, etc. Les approches sémantiques prennent en compte les informations tirées du domaine de connaissances lié au code source [36]. Les recherches sur l'application du clustering au logiciel adopte largement les approches basées sur la structure, mais il convient de noter que le résultat produit par les approches sémantiques tend à être plus significatif. C'est pourquoi certains essayent de combiner les deux méthodes [78].

La création de clusters est réalisée via un algorithme de classification. Le clustering est la forme la plus courante d'apprentissage non supervisé et la clé de ce type d'approches est la notion de distance entre les éléments à grouper et à séparer. Différentes mesures de distance donnent lieu à différents regroupements

Il existe deux catégories d'algorithmes hiérarchiques: Ascendant (bottom-up) et descendant (top-down). Dans le clustering appliqué au logiciel et selon [33], les algorithmes descendants offrent un avantage par rapport aux algorithmes ascendants car les utilisateurs s'intéressent principalement à la structure révélée par les grands groupes créés au cours des premières étapes du processus. En revanche, les décisions erronées prises au cours des premières étapes peuvent affecter la manière dont les regroupements ascendants évoluent vers les grands regroupe-

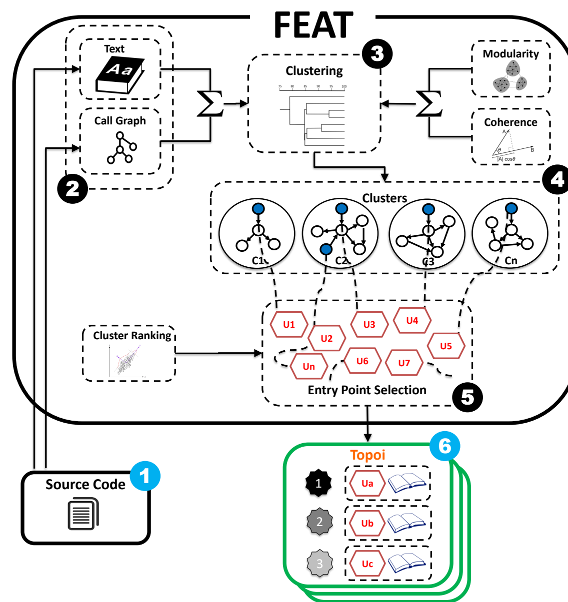

FIGURE 1 – FEAT *aperçu du processus*

ments. La classification hiérarchique ascendante est toutefois la plus utilisée [82].

Notre approche nommée FEAT

Au lieu d'utiliser uniquement du code source ou uniquement de la documentation d'un système logiciel, FEAT combine les deux dans une même perspective grâce à une métrique de distance fusionnant la partie sémantique et les éléments structurels contenus dans le code source.

Certaines méthodes nécessitent une assistance humaine. Pour ne citer que quelques exemples: les méthodes basées sur les LDA nécessitent la saisie de paramètres statistiques difficiles à définir à l'avance. Les utilisateurs finaux doivent sélectionner une partition en clusters parmi d'autres. Les approches d'apprentissage supervisé nécessitant l'étiquetage de la formation Des exemples qui demandent beaucoup de temps et sont sujets aux problèmes de subjectivité. À la différence de ces approches, FEAT est entièrement automatisé, il applique des critères définis formellement et sa sortie peut être directement utilisée pour l'extraction et/ou la localisation de fonctionnalités. Pour résumer les caractéristiques de FEAT:

- Les topoi de programme sont des structures concrètes résultantes d'une définition formelle, utiles pour relever les défis de la compréhension automatisée de programmes.
- En compréhension de programme, il existe une distinction entre les ap-

proches d'extraction et les approches de la localisation des fonctionnalités. FEAT ne fait aucune distinction et permet de répondre aux deux tâches.

- FEAT est basé sur un modèle de systèmes logiciels indépendant de tout langage de programmation.
- FEAT ne nécessite aucune entrée supplémentaire autre que le code source.
- FEAT est entièrement automatisé.

Un aperçu général de FEAT

La compréhension d'un logiciel à travers son code source peut être abordée par deux approches conceptuelles: structurelle ou sémantique. Les approches basées sur la structure se concentrent sur les relations statiques entre les entités tandis que les approches sémantiques incluent tous les aspects de la connaissance du domaine d'un système qui peuvent être obtenus à partir des commentaires et des noms des identifiants [73]. L'extraction des principales fonctionnalités d'un logiciel peut tirer parti d'informations structurelles permettant d'identifier une fonctionnalité en tant qu'ensemble de unités de code contribuant à son implémentation. D'un point de vue sémantique, les parties d'un système présentant des points communs en termes de mots en langage naturel peuvent également être considérées comme faisant partie d'une même fonctionnalité d'un système.

En d'autres termes, les approches structurelles et sémantiques véhiculent deux perspectives différentes et de valeurs inestimables. FEAT combine les deux pour obtenir une image plus précise des fonctionnalités proposées par un système logiciel. FEAT, dont les principaux éléments sont illustrés dans la Fig. 1, est basé sur un processus en trois étapes : pré-traitement (case notée 2 dans la Fig. 1), clustering (3 et 4 dans la Fig. 1) et sélection de points d'entrées (5 dans la Fig. 1). L'entrée de FEAT est un système logiciel représenté par du code source et des commentaires (1). À l'étape de prétraitement (2), FEAT analyse le code source et les commentaires, créant ainsi une représentation du système qui prend en charge la double hypothèse sous-jacente à l'approche.

Conclusion

FEAT automatise certaines pratiques courantes adoptées à la compréhension de programme (Sec. 2.2), telles que l'utilisation d'informations sémantiques et structurelles pour isoler les concepts sous la forme de clusters. Pour répondre au contexte de la compréhension du programme, nous avons adapté l'algorithme

de HAC en fournissant à la fois une nouvelle notion de distance (Sec. 4.5) et un critère d'arrêt (Sec. 4.7). L'application de PCA (Principal Component Analysis) à des unités de code a nécessité une étude approfondie des graphes d'appels du point de vue statistique (Sec. 4.8.1), révélant des modèles intéressants. Enfin, nous fournissons dans cette thèse une définition formelle de la notion nouvelle de topoi de programme (Def. 4.4). Nous montrons également que l'identification automatique de points d'entrée révélent certaines propriétés géométriques qui pourraient conduire d'intéressant développements. Enfin, la thèse présente une évaluation expérimentale approfondie incluant une expérience à grande échelle sur l'archive Software Heritage, soutenue par l'UNESCO.

Contents

1 Introduction **1**

2 State of the Art **5**
2.1 Introduction on Machine Learning 5
2.1.1 Supervised Learning 6
2.1.2 Unsupervised Learning 12
2.2 Program Comprehension 17
2.3 Program Comprehension via ML 21

3 Background **25**
3.1 Clustering 25
3.1.1 Distances and HAC 27
3.1.2 Merging criteria and HAC 28
3.1.3 Software Clustering 29
3.2 Principal Component Analysis 30
3.3 Vector Space Model 31
3.4 Latent Semantic Analysis 32
3.5 Call Graph 33

4 FEAT **Approach** **35**
4.1 Genesis of FEAT 35
4.2 A General Overview of FEAT 36
4.3 Semantic Perspective over Code Units 37
4.4 Structural Perspective over Code Units 44
4.5 Hybrid Distance 46
4.6 Distance over clusters 46
4.7 Selecting a Partition in HAC 48
4.7.1 Modularity 49
4.7.2 Textual Coherence 50
4.7.3 FEAT Cutting Criterion 53
4.7.4 HAC Revised Algorithm 53
4.8 Entry-Points Selection 53
4.8.1 Ranking Units through PCA 56
4.8.2 Geometric Aspects of Entry-points 66

4.9 Program Topoi 67
4.9.1 Entry-point Dictionary 68

5 FEAT **Tooling Support** **71**
5.1 Crystal Platform 71
5.1.1 OSGi 71
5.1.2 Business Process Modeling and Notation 72
5.2 Crystal.FEAT 73
5.2.1 Architecture 73
5.2.2 FEAT Process 74
5.2.3 User Interface 76

6 Experimental Evaluation **81**
6.1 Experimental Subjects 81
6.2 Goodness of FEAT in Selecting Partitions 82
6.3 Program Topoi Discovery Experiments 84
6.3.1 Random Baseline Comparison 86
6.3.2 No-clustering Experiment 86
6.3.3 Impact of α, β Parameters 87
6.3.4 Applicability of Program Topoi 91
6.4 Benefits of FEAT's Hybrid Distance 93
6.5 Scalability Evaluation 95
6.5.1 FEAT at Software Heritage 97
6.6 Threats to Validity 99

7 Conclusions **103**

List of Figures

2.1 High bias . . . 7
2.2 Good fit . . . 7
2.3 High variance . . . 7
2.4 Curse of dimensionality . . . 8
2.5 Kernel methods for support vector machines . . . 13
2.6 Separating hyperplane in SVM . . . 13
2.7 *K*-means and the shape of clusters . . . 15

3.1 HAC merging steps represented as a *dendrogram* . . . 27

4.1 Overview of FEAT . . . 38
4.2 Unit-documents as geometric entities . . . 43
4.3 Plot of words in concept space . . . 45
4.4 Graph medoids example . . . 47
4.5 Division of a graph into clusters . . . 49
4.6 Graphics of several experiments on modularity . . . 51
4.7 Graphics of several experiments on textual coherence . . . 52
4.8 Plots of entry-point attributes of GEDIT . . . 60
4.9 Plots of entry-point attributes of MOSAIC . . . 61
4.10 Histograms of the norm of attributes' vectors of two projects (GEDIT and MOSAIC, two clusters each). . . . 63
4.11 Example of entry-point selection . . . 66
4.12 Plot of the entry-points selection's example . . . 68
4.13 Plot of some entry-points extracted from GEDIT . . . 69

5.1 CRYSTAL's system architecture . . . 74
5.2 FEAT Process . . . 77
5.3 FEAT Web interface. List of projects . . . 78
5.4 FEAT Web interface. Program Topoi search . . . 79
5.5 FEAT Web interface. Program Topoi search, entry-point neighborhood . . . 79

6.1 FEAT's performance experiment with changing α and β . . . 89
6.2 FEAT's performance with changing α and textual elements . . . 90
6.3 "File open" entry-point neighborhood . . . 93

6.4 "File print" entry-point neighborhood 93
6.5 "Find text" entry-point neighborhood 93
6.6 "Clipboard copy" entry-point neighborhood 93
6.7 NCSA Mosaic web browser. Clustering of postscript printing feature. 95
6.8 NCSA Mosaic web browser. Clustering of browser window creation feature. 96
6.9 Correlation between running time and memory usage w.r.t. LOC, number of units, size of the dictionary and, density of CG 98
6.10 Experiment with Software Heritage 100
6.11 Running time estimator 102

List of Tables

4.1 Distances between unit-documents' vectors $\mathbf{d}_1, \ldots, \mathbf{d}_5$ and query vector $\mathbf{q}_k$ . . . 43
4.2 KS test definition . . . 59
4.3 KS test results . . . 59
4.4 Ranking of units in the example of entry-point selection . . . 67
4.5 Ranking example of a program topos . . . 67
4.6 Example of Entry-points' Dictionary . . . 70

5.1 CRYSTAL.FEAT commands' list . . . 80
5.2 CRYSTAL.FEAT RESTful API . . . 80

6.1 Experimental results of the clustering step's evaluation . . . 83
6.2 Random experiment . . . 87
6.3 No-HAC experiment . . . 87
6.4 Part of a program topos obtained from the analysis of GEDIT . . . 92
6.5 Feature location example in GEDIT . . . 92
6.6 Oracle used in the feature location experiment with MOSAIC . . . 94
6.7 Comparison of FEAT with a LSI-based approach, while running query: *"font"*. . . . 95
6.8 Open Source software projects used in the scalability evaluation experiment . . . 97
6.9 Summary of FEAT's experiment at Software Heritage . . . 99

Nomenclature

A Accuracy

AI Artificial Intelligence

ANN Artificial Neural Network

BPMN Business Process Modeling and Notation

CDF Cumulative Distribution Function

CG Call Graph

DNN Deep Neural Network

ECDF Empirical Cumulative Distribution Function

FEAT FEature As Topoi

fn False Negatives

fp False Positives

GDPR General Data Protection Regulation

HAC Hierarchical Agglomerative Clustering

IDC International Data Corporation

IR Information Retrieval

K-S Kolmogorov-Smirnov

kNN *k*-Nearest Neighbor

LDA Latent Dirichlet Allocation

LOC Lines of Code

LSA Latent Semantic Analysis

LSI Latent Semantic Indexing

ML Machine Learning

NB Naive Bayes

NLP Natural Language Processing

OSGi Open Services Gateway initiative

PCA Principal Component Analysis

PC Program Comprehension

P Precision

RDBMS Relational Data Base Management System

REST REpresentational State Transfer

RNN Recurrent Neural Network

R Recall

SOM Self Organizing Map

SVD Singular Value Decomposition

SVM Support Vector Machines

tf-idf Term Frequency-Inverse Document Frequency

tn True Negatives

tp True Positives

VSM Vector Space Model

I

Introduction

Context and Challenges

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 *program topoi*, which can be seen as summaries of the main capabilities of a program. Program topoi are given under the form of collections of ordered code functions along with a set of words (*index*) characterizing their purpose and a graph of their closer dependencies. 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.

Several disciplines in the context of program understanding make use of a similar concept called *feature*. Wiegers in his book [81] provides the following definition: *"...a feature is a set of logically related functional requirements that provides a capabil-*

ity to the user and enables the satisfaction of a business objective". This is a widely adopted definition in the literature but it is too abstract. Instead, program topoi are concrete objects targeted to source code, supported by a formal definition, and then suitable for automated computation. Nevertheless, extracting program topoi from source code is a complex task and to tackle this challenge we propose FEAT an approach and a tool for the automatic extraction of *program topoi*. FEAT acts in three steps:

1. *Preprocessing.* Creation of a model representing a software system. The model considers both structural and semantic elements of the system.

2. *Clustering.* By mining the available source code, possibly augmented with code-level comments, hierarchical agglomerative clustering (HAC) groups similar code functions.

3. *Entry-Point Selection.* Functions within a cluster are then ranked and those fulfilling some structural requirements will be selected as program topos' elements and stored.

Our work differs from those belonging to either feature extraction or location areas (see a detailed overview in Sec.2.3) for the following reasons. First, FEAT extracts topoi which are structured summaries of the main capabilities of the program, while features are usually just informal description of software characteristics. Second, in FEAT the difference between feature extraction and feature location is not so urgent; one can employ topoi to discover system capabilities but also for looking for those he/she already knows.

Contribution of the Thesis

FEAT is a novel, fully automated approach for program topoi creation based on clustering and ranking code functions directly from source code. Along the path which led us to the current definition of the approach, we devised some original contributions which are listed as follows:

1. For an effective application of HAC to software systems, we propose an original hybrid distance combining structural and semantic elements of source code (Sec. 4.5). HAC requires the selection of a partition among all those produced along the clustering process. FEAT makes use of a hybrid criterion based on *graph modularity* [17] and *textual coherence* [21] to automatically select the appropriate partition (Sec. 4.7).

2. Clusters of code functions need to be further analyzed to extract program topoi. We define the concept of entry-point to accomplish this. Entry-points

are an alternative way to look at functions, they are based on a set of structural properties coming from call graphs. Entry-points allow us: (*i*) to create a space in which functions can be treated as geometric objects, and (*ii*) to evaluate how representative they are in terms of a system's capabilities. We employ PCA (Principal Component Analysis) for entry-point selection (Sec. 4.8). We published this results in a conference paper presented at IAAI 30th Innovative Application of Artificial Intelligence [30] and in a paper published in IEEE Transactions on Reliability journal [31].

3. We implemented FEAT on top of a general-purpose software analysis platform and performed an experimental study over many open-source software projects. *Software Heritage* [1](SH) is an initiative owned by Inria whose aim is to collect, preserve and, share all publicly available source code. We processed 600 projects coming from various domains amounting to more than 25 MLOC (lines of code). During the evaluation we analyzed FEAT under several perspectives: the clustering step, effectiveness in topoi discovery and search, and scalability of the approach.

Organization of the Thesis

The thesis is organized as follows. Chap. 2 presents the current state of the art in the various disciplines touched by this thesis. Chap. 3 gives the necessary background on clustering, distance notions, PCA, call graphs, etc. Chap. 4 details the three main steps of FEAT with all the aspects we needed to handle in order to apply techniques such as HAC, PCA, etc. to source code analysis. Chap. 5 describes CRYSTAL the platform we designed to host FEAT. Chap. 6 gives the experimental results obtained with FEAT on several open-source software projects. Finally, Chap. 7 draws conclusions and presents some perspectives to this work.

1. www.softwareheritage.org

II

State of the Art

The chapter provides an overview of the state of the art of the main topics at the basis of this work: machine learning, program comprehension and, the research done so far in the domain located at their intersection.

2.1 Introduction on Machine Learning

Answering the question *"what is machine learning (ML)?"* can be tricky because ML is a really vast subject. In 1959 Arthur Samuel, a pioneer in the field of artificial intelligence (AI), coined the term *machine learning* [70]. He focused on *cognitive computing* [1] and, while he was working on a program capable of learning how to play chess, he gave the following definition: *"Machine Learning: Field of study that gives computers the ability to learn without being explicitly programmed."*. Tom Mitchell, another renowned researcher in ML, provided a more precise definition of machine learning in 1998: *"A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E."*

Hence, although machine learning is a field within computer science, it differs from traditional computational approaches. In traditional computing, algorithms are sets of explicitly programmed instructions used by computers to calculate or solve problems while in ML algorithms allow for computers to train on data inputs and build models to accomplish tasks.

1. Discipline that studies hardware platform and/or software systems that mimic the way the human brain works.

Both Samuel's and Mitchell's definitions clearly explain the goal for ML. The way we reach this goal is through ML algorithms. Let us see the main categories into which they are divided. The first category is (*i*) supervised learning, which trains algorithms based on example input and output data that is labeled by humans, and the second is (*ii*) unsupervised learning which provides the algorithm with no labeled data in order to allow it to find a structure within its input data.

2.1.1 Supervised Learning

Supervised learning algorithms can be further divided according to the output domain. If the output consists in one or more continuous variables we apply *regression* predictive modeling that is the task of approximating a mapping function (f) from input variables (X) to a continuous output variable (Y). On the other hand, a supervised learning problem where the answer to be learned is one of finitely many possible values is called *classification*.

Bias-variance Tradeoff

There are challenging aspects related to supervised learning algorithms. Behind any model there are some assumptions, but if they are too simplistic then the model generalizes well but does not fit the data (*underfitting*). This makes the model *biased* (see Fig. 2.1). On the other hand if the model fits the training data really well this is obtained at the expense of generalization which is called *overfitting* (see Fig. 2.3). Overfitting occurs when a model corresponds too closely to the training set but fails when classifies new, unseen data. The problem of satisfying both these two needs at the same time goes under the name of *bias-variance* tradeoff [44].

The Curse of Dimensionality

Modeling a classifier requires the selection of features. Features, also called variables or attributes, are the basic elements of the training data and their selection is needed for building the model representing a classifier. We can think of features as descriptors for each object in our domain. If we selected too few features then our classifier would perform poorly (i.e. classify animal species just by single animals' color). An obvious solution to this problem is to add more features in order to make our model more complex and flexible. But, we cannot add to the model as many features as we wish while the training set remains the same. If they are too many, we have to face the *curse of dimensionality* [7].

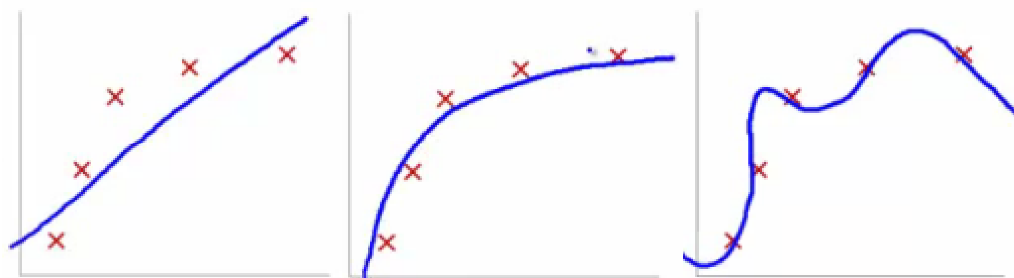

Figure 2.1 – *High bias* **Figure 2.2 –** *Good fit* **Figure 2.3 –** *High variance*

One side effect of the curse of dimensionality is overfitting. This happens because adding dimensions to our model will make the feature space grow exponentially and consequently it becomes sparser and sparser. Because of this sparsity, it is easier to find a hyperplane perfectly separating the training examples; our classifier is perfectly learning all the peculiarities of the training dataset even its exceptions but it will fail on real-world data because of overfitting.

Another side effect of the curse of dimensionality is that sparsity is not uniformly distributed. This is a bit surprising, let us try to explain it through an example. Imagine a unit square that represents a 2D feature space. The average of the feature space is the center of this square, and all points within unit distance are inside a unit circle inscribed into the square. In 2D the center of the search space is $\approx 78\%$ of the total. But, as we increase the number of dimensions, the center becomes smaller and smaller; in 3D it is $\approx 52\%$ and in 8D it is $\approx 2\%$. Fig. 2.4 shows a graph of the volume of 5 hyperspheres, with increasing value of radius, and its relationship with the number of dimensions. The graph clearly shows how fast the volume goes to zero as the number of dimensions increases. Though it is hard to be visualized, we can say that nearly all of the high-dimensional space is "far away" from the centre or, in other words, the high-dimensional unit hypercube can be said to consist almost entirely of the "corners" of the hypercube, with almost no "middle". In this scenario, where all the points are so distant from the center, distance metrics like the Euclidean distance are useless because there is not a significant difference between the maximum and minimum distance [3].

If we had an infinite number of training samples then we could use an infinite number of features to train the perfect classifier and then avoid the curse of dimensionality. In reality a rule of thumb is: *the smaller the size of the training dataset, the fewer the features that should be used*. It is important to highlight that the feature space grows exponentially with the number of dimensions [7] and so the number of examples should do accordingly. Feature selection can be a hard task to be accomplished. To tackle this challenge, techniques like Principal Component Analysis (PCA), which finds a linear subspace of the feature space with lower dimensionality (more on PCA in Chap. 3), can be used.

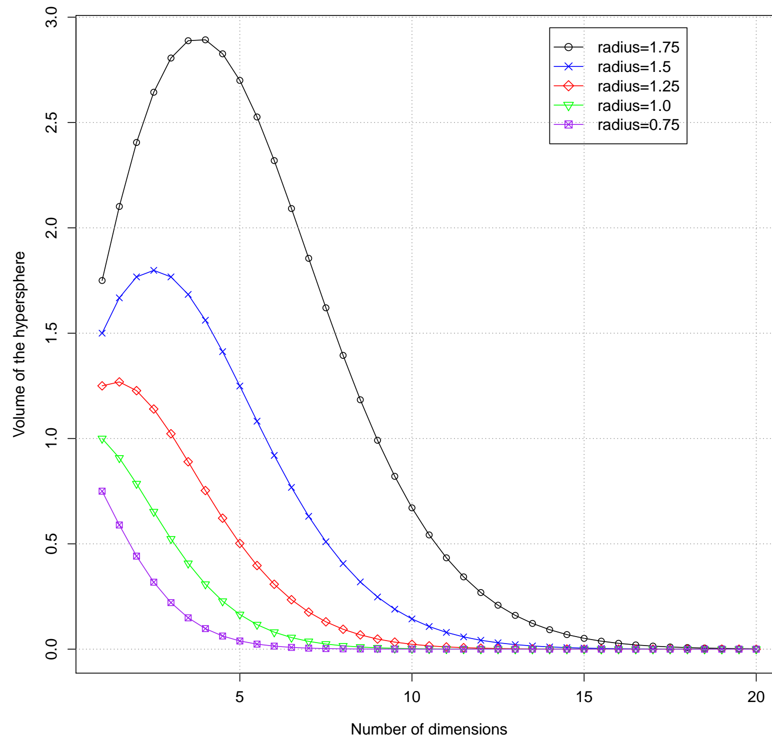

Figure 2.4 – *Volume of the hypersphere decreases when the dimensionality increases.*

Cross-validation

Overfitting is clearly a main concern for several supervised learning techniques. Another approach commonly adopted to overcome overfitting is *cross-validation* [52]. Basically, it splits the training dataset in two and the classifier is trained on just one subset of the examples. When the training is completed the remaining part is used for testing and evaluate the performance of the classifier. This approach called *holdout* is the basis for more advanced models like: *K-fold* and, *leave-one-out*.

After this overview about supervised learning and its challenging aspects, let us now see some practical examples of supervised learning methods through a selection of the most representative ones.

Decision Trees

A most widely used method for approximating discrete-valued functions is decision tree learning. Learned trees represent an approximation of an unknown target function $f : X \mapsto Y$. They belong to the family of inductive inference algorithms (i.e. inferring the general from the specific). Training examples are pairs

$\langle X_i, Y_i \rangle$ where every X_i is a feature vector and Y_i a discrete value. By walking the tree from the root to the leaves, decision trees classify instances. At each node the algorithm tests one attribute of the instance and selects the next branch to take on the basis of the possible values for this attribute. This step is repeated until the algorithm reaches a leaf indicating a Y value [52].

The basic algorithm for learning decision trees is ID3 [62], successively superseded by C4.5 [63]. ID3 builds a tree top-down by answering the question "which is the best attribute to be tested ?". The *best attribute*, that is the best one in separating the training examples according to their target classification, is selected on the basis of a statistical measure called *information gain*. Then, the algorithm moves down the tree and repeats the same question for each branch. This process goes on until all data is classified perfectly or it runs out of attributes.

There are some issues related to the basic ID3 algorithm, first of all overfitting which occurs with noisy data leading to the construction of more complex trees perfectly fitting the wrong data. Random forest [29] is an algorithm derived from decision trees and it is designed to address ID3 overfitting problem. Instead of applying the ID3 algorithm on the whole dataset, dataset is separated into subsets leading to the construction of several decision trees, decisions are made by a voting mechanism. C4.5 algorithm, though sharing the basic mechanism of ID3, brought some improvements to its ancestor like the capacity of dealing with missing attributes values or attributes with continuous values.

Artificial Neural Networks

The development of artificial neural networks (ANNs) has been inspired by the study of learning mechanisms in biological neural systems. These systems are made of a web of interconnected units (neurons) interacting through connections (axons and synapses) with other units.

The general structure of a ANN can be thought as made of three layers: input layer, hidden layer and output layer. The input layer is connected to the source which can be sensor data, symbolic data, feature vectors, etc. The units in the hidden layer connect inputs with outputs and it is where the actual learning happens. Each of these units computes a single real-valued value based on a weighted combination of its inputs. Units' interconnections in ANNs can form several type of graphs: directed or undirected, cyclic or acyclic. If the connections in a neural network do not form any cycle they are called *feedforward* neural network. In these networks the information flows from the input to the output layer through the hidden layer(s). No cycle or loops are present which distinguish them from recurrent neural networks (RNN).

There are several alternatives for primitive units in ANNs such as: *perceptron*,

linear unit and, *sigmoid*. Perceptron [49], which itself is a binary classifier, takes a vector of real-valued inputs and calculates a linear combination of these inputs. The output can be 1 or -1. Each input has an associated weight and learning a perceptron is about choosing values for the weights. A perceptron can represent only examples that can be linearly separated by a hyperplane in a n-dimensional space (n input points).

To overcome the limit about linearly separable samples it is not enough to add more layers; multiple layers of cascaded linear units still produce only linear functions. We need a unit whose output is a non-linear function of its inputs. One solution is the sigmoid unit (the name comes from the characteristic S-shaped curve) whose output ranges between 0 and 1, increasing monotonically with its input [52]. Sigmoid unit, like the perceptron, first computes a linear combination of its inputs and then applies a threshold to the result but, differently from perceptron, in sigmoid's case the output is a continuous function of its input. The algorithm to learn the weights in a multilayer network is *backpropagation* [8], based on *gradient descent* optimization algorithm. It attempts to minimize the squared error between the network output values and the target values for these outputs. When there are multiple hidden layers between the input and the output layers then we call the ANN a *deep neural network* (DNN).

One final remark about ANNs. The European Union recently introduced the General Data Protection Regulation (GDPR) that states the so-called *right to explanation*. Some of the articles of GDPR can be interpreted as requiring explanation of the decision made by a machine learning algorithm when it is applied to a human subject. This can clearly affect the adoption of ANNs in certain domains; provide an interpretation of the decisions made by neural networks it is very hard and practically impossible in DNNs.

Naive Bayes Classifier

Bayesian learning belong to a family of techniques based on statistical inference. The assumptions lying at their basis is that the input data of the classification problem follow some probability distribution. Bayesian learning estimates a mapping function between input and output data by means of the Bayes theorem. Its application can be challenging when dealing with many variables because of the difficulty of both computing and having samples of all joint probability combinations. A solution to this problem comes from an approach, which is widely used in the field of text classification, called *naive Bayes* (NB). It is called *naive* because it assumes that all input variables are conditionally independent [2] [26]

2. Two random variables X and Y are conditionally independent given a third random variable Z if and only if, given any value of Z, the probability distribution of X is the same for all values of Y and the probability distribution of Y is the same for all values of X.

hence dramatically simplifying the classification function (the number of needed parameters is reduced from exponential to linear in the number of variables).

NB belongs to a category called *generative* classifiers which is in contrast with those called *discriminative* classifiers. A generative classifier learns the model behind the training data making assumptions about its distribution. On this basis, it can even *generate* unseen data. Instead, discriminative classifiers make fewer assumptions on the data and just learn boundary between classes. The choice between the two categories is determined by the training set size. With a rich training set, discriminative classifiers outperforms generative ones [60]. But, in case of few data a generative model, with the addition of some domain knowledge, can become the first choice.

In NB the training set is made of objects (which are feature vectors) and classes to which the objects belong. The algorithm requires the prior probability of an object occurring in a class $P(C)$ and $P(x_k|C)$ that is the conditional probability of attribute x_k occurring in an object of class C. These probabilities are usually unknown and then estimated from the training set. This way NB indicates which class C best represents a given object x by the conditional probability $P(C|\mathrm{x})$ [44].

Despite its simplicity, NB, in some domains, showed performance similar to ANNs or decision trees.

K-nearest Neighbor

When it comes about making decisions on a ML approach it is really important to have a sufficient understanding on the problem at hand. Is it a nonlinear problem and its class boundaries cannot be approximated well with linear hyperplanes? Then a nonlinear classifier will be more accurate than a linear one. If the problem is linear, then it is best to use a simpler linear classifier. So far we have seen a nonlinear classifier (decision trees), a linear classifier (NB) and a mixed one depending on its structure (ANNs). The next two are nonlinear classifiers. Let us start with the simplest one.

k-nearest neighbor (kNN) [5] is one of the simplest classification algorithms. It is a *non parametric* classifier, meaning that it does not rely on any assumption about the distribution of the underlying data. Put simply, it builds a model for classification just from the data itself. This makes kNN a good candidate for a preliminary classification study when there is little a priori knowledge about the data distribution.

kNN is also defined a *lazy* algorithm because it does not have any training phase; it does not try to infer any generalization from the data.

kNN's output is a class membership assigned to a test element by the majority class of its *k* closest neighbors. We can have 1NN, but it is too sensitive to outliers or misclassified elements, usually a value $k > 1$ is chosen [44].

Support Vector Machines

Support vector machines (SVM) [79] are binary, discriminative classifiers. They learn an optimal hyperplane separating the training examples in two classes. SVMs are known to be capable of classifying data which are not linearly separable and they accomplish this through their key element: kernel functions [72]. Kernel functions take the input space and transform it into a higher dimensional one. This transformation allows linear classifier to separate non linear problems. The key aspects of the approach based on SVMs with kernel functions are:

- Data items are embedded into a vector space called the feature space.
- Linear relations are sought among the images of the data items in the feature space.
- The algorithm for the computation of a separating hyperplane does not need the coordinates of the embedded points, only the pairwise inner products.
- The pairwise inner products can be computed efficiently directly from the original data items using a kernel function.

These aspects are illustrated in Fig. 2.5. The left side of the figure shows a non-linear pattern but by applying a kernel like $\phi(x, y) = x^2 + y^2$ the data becomes linearly separable as it is shown on the right side plot. In the feature space, where data can be linearly separated, SVM, of all possible decision boundaries that could be chosen to separate the dataset for classification, chooses the decision boundary (separating hyperplane) which is the most distant from the points (support vectors) nearest to the said decision boundary from both classes (see Fig. 2.6).

In conclusion, SVM is a very effective tool; it uses only a small subset of training points (support vectors) to classify data, it is versatile: different kernel functions can be specified to fit domain specific problems.

2.1.2 Unsupervised Learning

In unsupervised learning, the training data consists of a set of input vectors without any corresponding target values. The goal in such problems may be to dis-

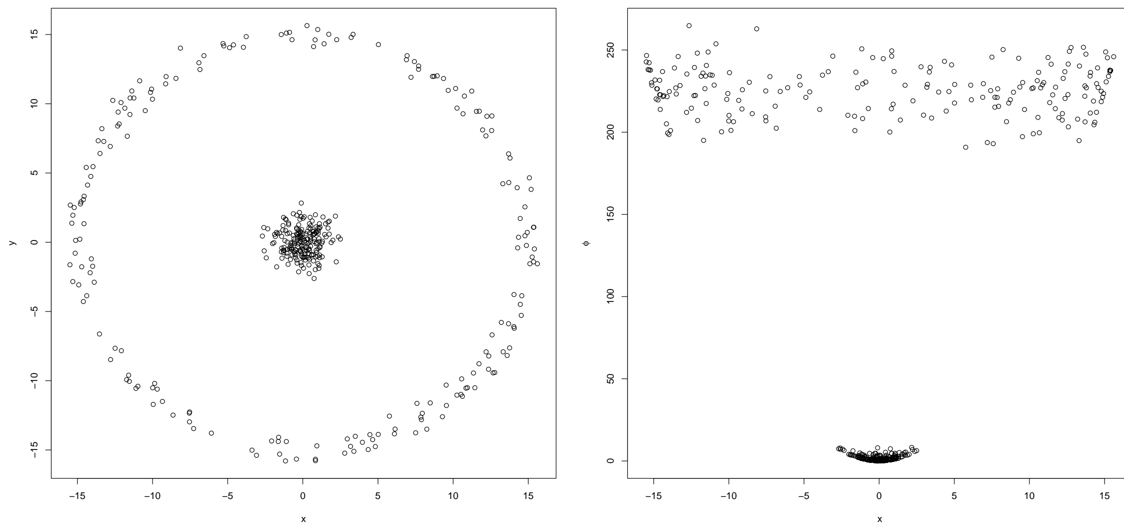

Figure 2.5 – *The function $\phi = x^2 + y^2$ embeds the data into a feature space where the nonlinear pattern now becomes linearly separable.*

cover underlying structures of a dataset like groups or create summaries of it. Two representative tasks, achievable through unsupervised learning algorithms, are clustering data into groups by similarity and dimensionality reduction which compresses the data while revealing its latent structure.

It is not always easy to understand the output of unsupervised learning algorithms; they can figure out on their own how to sort different classes of elements, but they might also add unforeseen and undesired categories to deal with creating clutter instead of order.

Since there are no labeled examples, a challenging aspect of such approaches is

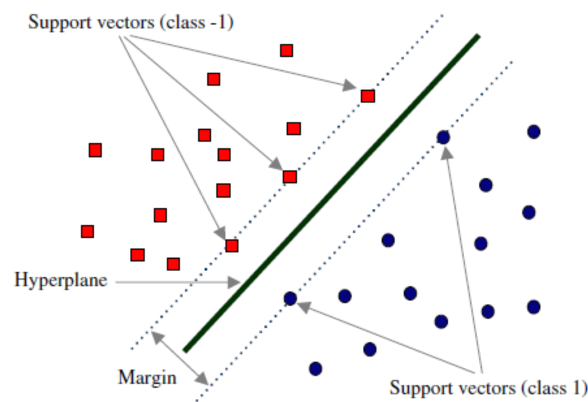

Figure 2.6 – *SVM's separating hyperplane maximizing margin among support vectors belonging to the two classes.*

evaluating their performance; what is usually done is to create either ad-hoc metrics or exploiting some domain-specific knowledge.

Let us see an overview of some of the most widely adopted algorithms in unsupervised learning.

K-means

Clustering is the most common form of unsupervised learning. It organizes data instances into similarity groups, called clusters such that the data instances in the same cluster are similar to each other and data instances in different clusters are very different from each other [42, 44]. One might wonder whether there is such a big difference between classification and clustering; after all, in both cases we have a partition of data items into groups. But the difference between the two problems lies at their foundation. In classification we want to replicate the criterion a human used to distinguish elements of the data. In clustering there is no such guidance and the key input is the distance measure.

Clustering algorithms are divided into two broad categories: *flat* clustering and *hierarchical* clustering. The former algorithms do not create any structure among clusters whereas the latter provide an informative hierarchy [44].

K-means [27] is the most important flat clustering algorithm. The goal of this algorithm is to find K groups in the data. The algorithm works iteratively to assign each data point to one of K groups based on the features that are provided. Data points are clustered based on feature similarity. K-means aims at minimizing the average squared Euclidean distance of data points from their cluster centers. A cluster center is represented by the mean or *centroid* of the elements in a cluster.

The K in K-means indicates the number of clusters and it is an input parameter. The algorithm (1) randomly chooses K data points (seeds) as centroids and (2) iteratively reassigns data points to clusters and update centroids until a termination condition is met. Some drawbacks of this simple and effective algorithm are: (1) a prespecified number of clusters that can be hard to be provided, (2) sensitivity to outliers, (3) the selection of initial seeds affects the search which can get stuck in a local minimum and, (4) the assumption about the shape of the clusters; K-means works best with non-overlapping spherical clusters (see Fig. 2.7).

Hierarchical Clustering

Hierarchical clustering establishes a hierarchy among clusters which can be created either with top-down or bottom-up algorithms. Top-down clustering re-

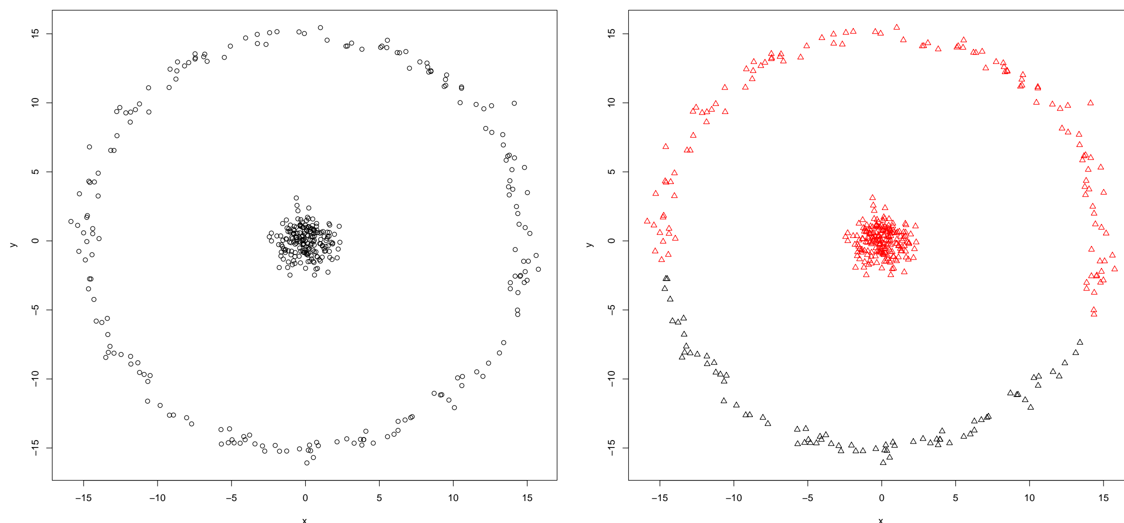

Figure 2.7 – *It is easy for a human to see the two clusters in the left plot. The right one shows the clustering (red and black triangles) created by* K*-means. The algorithm fails in finding the two clusters in this dataset because it tries to find two centers with non-overlapping spheres around them.*

quires a criterion to split clusters. It proceeds by splitting clusters recursively until individual data points are reached. Bottom-up algorithms, also called *Hierarchical Agglomerative Clustering* (HAC), start by considering each data point as a singleton cluster and then incrementally merge pairs of clusters until either the process is stopped or all clusters are merged into a single final cluster [44].

Unlike the K-means algorithm, which uses only the centroids in distance computation, hierarchical clustering may use anyone of several methods to determine the distance between two clusters.

Hierarchical clustering has several advantages over K-means. It can take any distance measure, can handle clusters of any shape, the resulting structure can be really informative. Hierarchical clustering on the other hand are computationally demanding both in terms of space and time. Also some criteria used to merge clusters suffer from the presence of outliers. More details about hierarchical clustering will be given in Chap. 3.

One-class SVM

In the supervised learning section we have seen SVMs. They train a classifier on a training set so that examples belong to one of two classes. This is clearly a supervised learning model but actually SVM can be used also in contexts where we can have a large majority of positive examples and we want to find the negative

ones. This peculiar application of SVMs is called *one-class SVM* [71].

In one-class SVM, the support vector model is trained on data that has only one class, which is the *normal* class. It infers the properties of normal cases and from these properties can predict which examples are unlike the normal examples. This is useful for anomaly detection because the scarcity of training examples is what defines anomalies: that is, typically there are very few examples of the network intrusion, fraud, or other anomalous behavior. Then when new data are encountered their position relative to the *normal* data (or inliers) from training can be used to determine whether it is *out of class* or not; in other words, whether it is unusual or not.

One-class SVM have been applied to several different contexts like: anomaly detection, novelty detection, fraud detection, outlier detection, etc. proving that is a very versatile and useful approach to unsupervised learning.

Autoencoder

An *autoencoder* is a neural network that has three layers: an input layer ($\mathbf{x}$), a hidden (encoding) layer ($\mathbf{h}$), and a decoding layer ($\mathbf{y}$). The network is trained to reconstruct its inputs by minimizing the difference between input and output, which forces the hidden layer to try to learn good representations of the inputs. Autoencoders are designed to be unable to perfectly reproduce the input. They are forced to learn an approximation, a representation that resembles the input data. Being forced to set higher priority to some aspects of the input usually leads autoencoders to learn useful properties of the data [28].

One way to obtain useful features from the autoencoder is to constrain $\mathbf{h}$ (hidden layer) to have a smaller dimension than $\mathbf{x}$ (input layer). An autoencoder whose code dimension is less than the input dimension is called *undercomplete*. Learning an undercomplete representation forces the autoencoder to capture the most salient features of the training data.

Learning autoencoders proved to be difficult because the several encodings of the input code compete to set the same small amount of dimensions. This has been solved through *sparse* autoencoders [66]. In a sparse autoencoder, there are actually more hidden units than inputs, but only a small number of the hidden units are allowed to be active at the same time.

Traditionally, autoencoders have been used for dimensionality reduction, feature learning and, noise removal. Autoencoders are quite similar to PCA (principal component analysis) (see Chap. 3 for a more detailed discussion on PCA) in terms of potential applications. One advantage of autoencoders over PCA is the capacity of learning nonlinear transformations.

Self Organizing Map

Self organizing map (SOM) are a ANN-based technique for dimensionality reduction. More specifically they are used for creating visual representation of high dimensional data in 2D. Sometimes they are also called Kohonen maps or networks [35] according to the name of the Finnish professor who introduced them in the 1980s. Teuvo Kohonen writes *"The SOM is a new, effective software tool for the visualization of high-dimensional data. It converts complex, nonlinear statistical relationships between high-dimensional data items into simple geometric relationships on a low-dimensional display. As it thereby compresses information while preserving the most important topological and metric relationships of the primary data items on the display, it may also be thought to produce some kind of abstractions."*

The visible part of a SOM is the map space, which is either a rectangular or hexagonal grid of nodes (neurons). Each node has an associated weight vector with the same dimensionality of the input data and it represents node's position in the input space. After an initialization step, where weight vectors are initialized at random, SOM algorithm picks an input vector and select the *best matching unit* (BMU) that is the closest node to the input vector. Then, BMU and its neighborhood is moved towards the input vector. The algorithm iterates until it meets a *stabilization* criterion.

SOM are said to preserve the topological structure of the input data set which simply means that if two input vectors are close together, then the neurons related to those input vectors will also be close together. SOM can also act as classifier for new data by looking for the node with the closest weight vector to the data vector.

Finally, SOM can be considered as a nonlinear generalization of principal component analysis (PCA).

2.2 Program Comprehension

Program comprehension (or program understanding) aims to recover high-level information about a software system and it is an essential part of software evolution and software maintenance disciplines. It is characterized by both the theories about how programmers comprehend software as well as the tools that are used to assist in comprehension tasks [75]. The earliest approaches to program comprehension (PC) were based on two theories called: *top-down* and *bottom-up*. Top-down theory explains PC as follows [9]: programmers when try to comprehend a program, make some hypothesis which can be confirmed by finding the so-called *beacons*. Beacons are lines of code which serve as typical indicators of

a particular structure or operation [80]. An example of a beacon, is a swap of values, which is a beacon for a sort in an array, especially if it is embedded in program loops. While rejected hypothesis are discarded, programmers retain the confirmed one to build their program's comprehension. The bottom-up theory is based on chunking [40]. Chunks are pieces of code which are familiar to the programmer. They are associated to a meaning and a name. The programmer builds his understanding of the program by assembling larger chunks through smaller ones.

The underlying objective of those theories was to achieve a complete comprehension of programs but as they become larger and larger this is clearly an unfeasible target. Despite all the differences, there is one point shared among the various theories: experienced programmers act very differently from novice ones; a different view, based on how experienced programmers approach PC, was then proposed. When given a task, experienced programmers focus on *concepts* and how they are reflected in the source code [34]. They do not try to understand each and every small detail of the program but they seek the minimum understanding for the task at hand. Concepts play a key role in such context [65]. Requests to change a program are formulated in terms of concepts, for example: "Add HTML printing to the browser". Hence, the main task is to find where and how the relevant concepts are implemented in the code. Several definitions of *concept* have been provided. One that can be directly applied to PC is: *"Concepts are units of human knowledge that can be processed by the human mind (short-term memory) in one instance"* [65].

The set of concepts related to a program is not fixed. We can have one set of concepts during the specification phase. Others are added in the design and implementation phases. New concepts may emerge during the maintenance when unexpected usages of the system arise.

Concepts are a key element of human learning [61]. Learning is an active process and humans extend their knowledge on the basis of some pre-existing knowledge. We have *assimilation* when the new facts are incorporated without changing the pre-existing knowledge. *Accommodation* occurs when new facts are acquired but they require a reorganization of the pre-existing knowledge. This theory about learning has been directly applied to PC. Programming knowledge has many aspects but in PC the most relevant one encompasses domain concepts and their implementation in the code. PC's objective is to bridge the gaps in that knowledge [65].

Rajlich [64] represents a concept as a triple consisting of a *name*, *intension* and, *extension*. The name is the label that identifies the concept, intension explains the meaning of the concept and extension is a set of artifacts that realize the concept. Concept location is then formalized as the function $\text{Location} : \text{intension} \mapsto \text{extension}$.

For programmers acquiring the knowledge of concepts is usually easier than locate them in the code. By using a word processor or reading the user manual, a programmer can learn about copy-paste, print and other concepts of the domain but he/she knows nothing about their implementation. Concept location is the process of finding the mapping between concepts and the fragments of code implementing them.

Change requests are expressed in terms of domain's concepts like:*"The system raises an exception when converting a file in PDF"* then, to start fixing the issue, a programmer needs first to locate the code where the concepts *converting*, *file* and, *PDF* are located. This is how the change process actually starts. At this point experienced programmers use their intuitions to find the code and if they fail then they start looking for string correspondence of identifiers. If the search succeeds then they start examining the surrounding code to decide whether this is truly the location that implements the concept. Concept location, exemplified before, is a common and frequent activity especially in software maintenance and evolution.

Program comprehension is not an end goal, but rather a necessary step in achieving some other objective as we have already seen, such as fixing an error, reusing code, or making changes to a program. In practice PC is employed in recovering the following information of a software system:

- Structure (components and their interrelationships)
- Functionality (what operations are performed on what components)
- Dynamic behavior (how input is transformed to output)
- Rationale (how was the design process and what decisions have been taken)
- Construction, modules, documentation, and test suites

In many practical applications of PC we find the term *feature* which is related to concepts. Features are associated with the behavior of a software system. They represent the set of functionality. We would not say that *bubble sort*, though it might belong to the set of domain's concepts, is a specific feature of the system. All features are concepts but not all concepts are features [46]. The notion of feature is a very relevant one in this context and a widely adopted definition is: *"...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"* [81].

Let us now give a closer look to the most common activities in PC: *feature discovery* and *feature location*.

Feature Discovery

Feature discovery (or extraction)[3] [2] is the task of identifying the key functionality that a program implements.

At present, feature discovery is mainly a manual process. There are three options usually adopted by programmers [67]: (1) Read software documentation, such as requirement documents, (2) read the source code and execute it, (3) communicate directly with code's authors.

Unfortunately, requirements are usually out-of-date or incomplete, communicating with the authors is often not realistic, because they can have left the company or they are not even known. Therefore, programmers usually have to manually read the source code and interact with the program.

Manual feature discovery process can be time consuming. That is why several automatic approaches have been proposed. In order to automatically find the main characteristics of a software system, several software resources are used, such as documentation, bug reports, requirements documents, and source code (some examples in Sec. 2.3).

Feature Location

Software maintenance and evolution involves adding new features to programs, improving existing functionalities, and removing bugs. Identifying an initial location in the source code that corresponds to a specific feature is called *feature location*. It is one of the most frequent activity undertaken in software maintenance. Programmers use feature location to find where in the code the first change to complete a task needs to be done. Then, impact analysis will guide the process in order to cover the full extent of the change [16]; starting from the source code found with feature location, the process will proceed by finding all the code affected by the change.

Similarly to what happens in the context of feature discovery, also in feature location, most of the time programmers have to run a manual search in the code repository.

3. Here, and in the rest of the chapter, we refer to the program comprehension domain's sense of *feature* in software engineering that should not be confused with its meaning seen previously in the context of machine learning.

2.3 Program Comprehension via ML

This sections presents an organized review of the main works at the crossroad between machine learning and program comprehension.

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

Dumitru *et al.* [18] uses text-mining techniques and flat clustering 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.* [48] presented four automatic generators of list of features for software projects, which select English sentences that summarize features from the project documentation.

Mining Source Code

Linstead *et al.* [41] propose dedicated probabilistic models based on code analysis using *Latent Dirichlet Allocation* (LDA) to discover features under the form of so-called *topics* (main functions in code). McMillan *et al.* [50] present a source-code recommendation system for software reuse. Based on a feature model (a notion used in product-line engineering and software variability modeling), the proposed system tries to match the description with relevant features in order to recommend the reuse of existing source code from open-source repositories. Abebe *et al.* [2] propose 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 for the identification of concepts and relations among the sentences. Grant *et al.* [25] address 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.

Kuhn *et al.* [37] use clustering and latent semantic indexing (LSI) to assess the similarity between source artifacts and to create clusters according to their similarity. The most relevant terms extracted from LSI analysis are reused for labeling the clusters.

Zhao *et al.* [85] exploit 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.

Finally, Moreno *et al.* in [54] 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. Moreno [53] proposes to generate summaries in natural language of complex code artifacts (i.e., classes and change sets).

Related to the domain of source code driven program comprehension are the works about reversing software product lines (SPLs) [4]. SPLs are modeled through *feature models* (FM) which are basically a AND-OR graph with some constraints. A feature model represents the combinations of the common base and features leading to the creation of allowed variants of the software system. Damasevicius et al. [14] define an approach for the automated derivation of feature models from existing software artifacts (components, libraries, etc.). Their approach accepts as input as set of `Java` classes and extracts a basic FM. Instead of considering just one project Al-Msie'deen et al. [4] propose an approach for the identification of feature implementations from a collection of software product variants based on formal concept analysis.

Conclusion

We presented in this chapter the state of the art of the disciplines touched in this thesis, namely: machine learning, program comprehension and the combination of the two. There are several challenging aspects in both supervised learning (curse of dimensionality, overfitting, legal issues, etc.) and unsupervised learning (evaluating performance when no labeled data are available, creation of unforeseen, even undesirable categories of data which are hard to interpret, etc.). But, despite all these difficulties, machine learning algorithms, when appropriately applied, proved to be capable of solving hard problems that would be impossible to face with traditional computing approaches.

Program comprehension defines the theories and practices lying behind the ac-

4. SPLs encompass modeling, techniques and tools that, starting from a common and stable base of characteristics, can create different versions of a software system called *software product variants*.

tivities of retrieving high-level information about a software-system. The chapter presented some basic notions in PC like beacons, concepts, feature extraction, feature location and how experienced programmer deal with this challenging task. This elements will be used further on in the thesis.

Finally, Sec.2.3 presented the current state of the art about the intersection of PC and ML. This has been clearly a source of inspiration at the beginning of this thesis' work. The main relevant approaches have been divided into two categories according to the data they mine: documentation based approaches and source code based approaches.

III

Background

This chapter presents the background behind the basic elements upon which FEAT is built. The material will be presented starting from hierarchical clustering in general and then focusing on software clustering. The rest of the chapter will introduce the notion of call graph and discuss some machine learning approaches used in FEAT to cluster functions under both a semantic and structural perspective.

3.1 Clustering

Organizing data into sensible grouping is one of the most fundamental modes of understanding and learning [32]. For example, classifying organisms into taxonomies has been a common scientific scheme to analyze: animals, peoples, diseases, etc. Cluster analysis, or clustering, groups a set of elements such that those in the same group (cluster) are more similar in some sense to each other than to those in other groups. Cluster analysis has a long history in a wide variety of scientific fields. The first references to one of the most popular algorithm, *K-means*, come from the fifties. Successively, in 1967, James MacQueen [43] laid the ground for the current version of the algorithm.

Clustering offers a viable approach to run exploratory analysis to find structure in data especially when dealing with huge amount of data. Nowadays growth in data production requires advances in methodology to automatically understand, process and, summarize data. According to the latest published study

(April 2014) from International Data Corporation (IDC) [1] on the size of the *digital universe*, in 2013 4.4 ZB (ZB is zettabytes and $1\text{ZB} = 10^{21}$bytes) of digital data have been produced; this means that the number of bits in the digital universe is almost as the number of stars in the physical universe. The digital universe is clearly huge and it is growing exponentially: they estimated a growth factor of ten leading to a size of the data of 44 ZB in 2020.

When clustering is applied to software artifacts (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* [73, 47], *software evolution and maintenance* [36], *reflexion analysis* [55, 12] and *feature extraction* [48].

Clustering can be mainly divided into two broad categories of algorithms: flat and hierarchical clustering.

Flat Clustering Algorithms. In these algorithms, each cluster is represented by a single center point. By updating iteratively the center of each cluster, these algorithms can reach a convergence state. The absence of a structure, showing the relationship among clusters, makes this approach *flat*. K-means is the most important flat clustering algorithm. There are some assumptions behind K-means involving the shape of the data: the ideal cluster is a sphere (or a hyper-sphere in dimensions higher than 3) and clusters do not overlap (more details in Sec. 2.1.2).

Hierarchical Clustering Algorithms. Hierarchical clustering establishes a hierarchy among clusters which can be created either with top-down or bottom-up algorithms. Top-down clustering requires a criterion to split clusters. It proceeds by splitting clusters recursively until individual data points are reached. Bottom-up algorithms, also called *Hierarchical Agglomerative Clustering* (HAC), start by considering each data point as a singleton cluster and then incrementally merge pairs of clusters until either the process is stopped or all clusters are merged into a single final cluster (see Fig. 3.1) [44].

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*. Fig.3.1 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.3.1, without any stopping criterion, HAC ends up with a single cluster. Defining how to merge clusters and an appropriate stopping criterion, so that a meaningful partition can be selected, are challenging

1. Source www.idc.com/

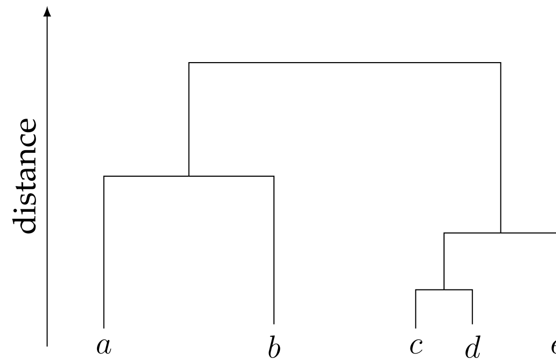

Figure 3.1 – Dendrogram *showing the merging steps of HAC. The height of the branches is proportional to the distance of the two merged elements. In the example the first merge involves* c *and* d *because they are the closest data points.*

aspects of HAC (Sec.4.6 and Sec.4.7 will explain how we handled these problems in FEAT). A key notion in HAC is the distance between single data points and clusters. Let's provide some formal elements about distances. A function d over two vectors $\mathbf{a}$ and $\mathbf{b}$ is called a *distance* if and only if it satisfies the following properties:

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

In cases where the third property (triangular inequality) needs to be relaxed, the function $\mathrm{d}(\mathbf{a}, \mathbf{b})$ is a *dissimilarity* [56].

3.1.1 Commonly Applied Distances to HAC

Before the actual clustering phase can take place the notion of distance between data points needs to be defined. Several distances can be applied in the context of clustering and choosing the *right* one is a key step in applying hierarchical clustering since they can strongly affect clustering's outcome. When working in a vector space a commonly adopted way to measure distances is a Minkowski distance, which is a family of metrics defined as:

$$L_p(\mathbf{a}, \mathbf{b}) = \left(\sum_{i=1}^{n} |\mathbf{a}_i - \mathbf{b}_i|^p \right)^{1/p} \; ; \forall p \geq 1, \; p \in \mathbb{Z}^+$$

where $\mathbb{Z}^+$ is the set of positive integers. The Manhattan and Euclidean distances are special cases of the Minkowski distance when correspondingly $p = 1$ and $p = 2$. These distances suite very well to contexts where clusters should contain data points on the basis of their geometric proximity.

Cosine similarity, which gives the angle between two vectors, is widely used in text retrieval to match vector queries to the dataset. The smaller the angle (θ) between a query vector and a document vector, the closer a query is to a document. The normalized cosine similarity is defined as follows:

$$\mathrm{d}(\mathbf{a}, \mathbf{b}) = \cos(\theta) = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \|\mathbf{b}\|}$$

Cosine similarity can be transformed into a distance called *angular distance* (See Eq. 4.3). One can apply angular distance in vector spaces when more than in the magnitude he/she is interested in the direction of vectors (Sec.4.3 provides some insights about the choice between angular distance and Euclidean distance when dealing with natural text documents).

When the objective of clustering is creating groups of samples showing a similar behavior (that is if the greater values of one sample mainly correspond with the greater values of the other sample, and the same holds for the lesser values) then correlation-based distances can be applied. Pearson distance is based on the coefficient of the same name:

$$\rho_{\mathbf{a},\mathbf{b}} = \frac{\mathrm{cov}(\mathbf{a}, \mathbf{b})}{\sigma_{\mathbf{a}} \sigma_{\mathbf{b}}}$$

where cov is the covariance and σ is the standard deviation. Since from the absence of correlation to the total positive correlation the coefficient ranges between $[-1, 1]$, Pearson's distance is defined as $\mathrm{d}(\mathbf{a}, \mathbf{b}) = 1 - \rho_{\mathbf{a},\mathbf{b}}$. Closely related to Pearson's distance there is the distance based on Spearman's correlation coefficient. Spearman correlation is also seen as the Pearson correlation of the ranked sample values. The ranking is the relative position label of a sample's value like: 1st, 2nd, etc. Since the Spearman's coefficient is calculated over rankings, it may allow the computation of distances of variables whose values are described by qualitative data.

3.1.2 How Merging Criteria Affect the Behavior of HAC

In HAC, the creation of clusters proceeds bottom-up treating every data point as a cluster (singleton) and then progressively merging pairs of clusters until all of them have been merged into a single cluster. This can involve not only two data points but also a data point and a cluster or two clusters. Without any loss of generality, let's now consider only the case of a distance over two clusters. In order to deal with this extended concept of distance that can involve two sets of data points, several criteria have been developed. Some of them just select a pair of data points, one for every cluster, and then the distance computed over those two points is considered to be representative of the distance between the two clusters.

Single-link and *complete-link* are the most widely known notions of cluster distance based on single data points. In *single-link* clustering, the distance between two clusters is the distance of their *most similar* members. It's a local merging criterion: we pay attention just to the area where the two clusters come closest to each other. Other, more distant parts of the cluster and the clusters' overall structure are not taken into account. In *complete-link* clustering the distance of two clusters is the distance of their *most dissimilar* members. This is equivalent to choosing the cluster pair whose merge has the smallest diameter. This criterion is not local: the entire structure of the clustering can influence merge decisions. This results in a preference for compact clusters with small diameters over long, straggly clusters, but also causes sensitivity to outliers. A single data point far from the center can increase the diameters of candidate merge clusters dramatically and completely change the final clustering.

Single-link and complete-link clustering reduce the assessment of cluster quality to a single distance between a pair of data points. But a measurement based on one pair cannot fully reflect the distribution of data points in a cluster. The obvious solution to this problem is to involve all data points in assessing clusters' similarity.

Centroid clustering [76] is an approach which considers the centroid (center of mass) of each cluster and computes the distance between clusters by considering the distance between their centroids. Hence, centroids offer an adequate representation for a set of points. The *centroid* of a set of points, denoted by μ, lies at the average position of all points. In an n-dimensions space, the centroid of $\mathcal{C} = \{\vec{v_1}, \ldots, \vec{v_k}\}$, where each vector $\vec{v_i}$ has coordinates $(x_{i_1}, x_{i_2}, \ldots, x_{i_n})$, can be computed using the formula: $\mu(\mathcal{C}) = \frac{1}{k}\left(\sum_{i=1}^{k} x_{i_1}, \ldots, \sum_{i=1}^{k} x_{i_n}\right)$. 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* [19] 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.

3.1.3 Software Clustering

Software clustering methodologies create group of entities, such as classes, functions, etc. of a software system in order to ease the process of understanding the high-level structure of a large and complex software system [73]. The basis for any cluster analysis to group entities is their set of attributes.

The application of clustering to a software system requires the identification of the entities which are the object of the grouping. Several artifacts can be chosen but the most popular one is source code [51]. The selection of entities is affected

by the objective of the method. For program restructuring at a fine-grained level, function call statements are chosen as entities [84] while for design recovery problems [6] entities are often software modules but also classes or routines.

Extracting facts from source code can be done following two different conceptual approaches: *structural and semantic*. Structure-based, approaches rely on static relationships among entities: variable references, procedure calls, inheritance etc. Semantic approaches take into account the domain knowledge information contained in source code conveyed from comments and identifier names [36]. The software clustering community widely adopts structure-based approaches but it has to be noted that the output produced by semantic approaches tends to be more meaningful. That is why some try to combine the strengths of both methods like Tzerpos et al. [78].

The actual cluster creation is accomplished through a clustering algorithm and the key input to a clustering algorithm is the distance measure. Different distance measures give rise to different clusterings.

We have already seen that there are two categories of hierarchical algorithms: agglomerative (bottom-up) and divisive (top-down). In software clustering, according to Kaufman et al. [33], divisive algorithms offer an advantage over agglomerative ones because users are mostly interested in the structure revealed by the large clusters created during the early stages of the process. On the other hand, the way agglomerative clustering proceeds towards large clusters may be affected by unfortunate decisions made in the first steps. Agglomerative hierarchical clustering are most widely used however; it is infeasible to consider all possible divisions of the first large clusters [82].

3.2 Principal Component Analysis

Principal Component Analysis (PCA) is a powerful technique from data analysis world, and it has a very convenient capability for our own purposes: it can highlight the most relevant dimensions in a given space. The notion of relevance here relates to revealing dimensions where most of the variations (more precisely variance) occur.

These dimensions (principal components) are not the original ones any more but a linear combination (rotation and scaling) of them. This new space, usually called feature [2] space, has the ability of highlighting the differences among the original data set in the best possible way by maximizing the variance along any axis. As a result of the transformation, the first principal component has the

2. Not to be confused with the features previously mentioned in the context of program understanding.

largest possible variance; each succeeding component has the highest possible variance under the constraint that it is orthogonal to (i.e., uncorrelated with) the preceding components.

The application of PCA requires a transformation of the input matrix trough the Singular Value Decomposition (SVD). SVD is a linear algebra method for the factorization of real (and complex) matrices. If we have a $m \times n$ matrix $\mathbf{X}_{m \times n}$, SVD factorizes it into the product of three matrices $\mathbf{X} = \mathbf{U\Sigma V}^T$, where $\mathbf{U}_{m \times m}$ and $\mathbf{V}_{n \times n}$ are in column orthonormal form [3] and $\mathbf{\Sigma}_{m \times n}$ is a diagonal matrix of singular values [24]. If $\mathbf{X}$ is of rank r, then $\mathbf{\Sigma}$ is also of rank r. An interesting application of SVD, called *low-rank approximation*, consists in keeping only the first $k < r$ components of $\mathbf{X}$. Truncated SVD is a key element of PCA. Let $\mathbf{\Sigma}_k$, where $k < r$, be the diagonal matrix formed from the top k singular values, and let $\mathbf{U}_k$ and $\mathbf{V}_k$ be the matrices obtained by selecting the corresponding columns from $\mathbf{U}$ and $\mathbf{V}$, then the matrix $\mathbf{X}_k = \mathbf{U}_k\mathbf{\Sigma}_k\mathbf{V}_k^T$ is the matrix of rank k that best approximates the rank r original matrix $\mathbf{X}$.

Keeping only the first $k < r$ components of the SVD factors presents several advantages: latent meaning discovery, noise reduction, sparsity reduction [77], faster computation, ability of making data lying in high dimensional space displayable in 2-D or 3-D, etc. In FEAT, PCA is a key ingredient for entry-point selection (see Sec. 4.8).

3.3 Vector Space Model

Vector Space Model (VSM) represents natural language documents in a way that can be understood by computers, it is part of a broad range of semantic technologies including Latent Semantic Analysis (LSA). The idea of VSM is to represent each document in a corpus as a point in a space. Points that are close together in this space are semantically similar and points that are far apart are semantically distant.

There are several vector-based representations but we focus here on the *term-document* matrix. In a term-document matrix rows correspond to terms and columns correspond to documents.

VSM represents textual documents as *bag of words*. The term *bag* (also multiset) comes from mathematics and identifies sets where multiple occurrences of elements are allowed. For example, $\{a, a, b, c, c, c\}$ is a bag containing a, b, and c. Order is not relevant in bags; the bags $\{a, a, b, c, c, c\}$ and $\{c, a, c, b, a, c\}$ are equivalent. We can represent bags with vectors, by conventionally define that the first

3. In orthornormal form columns are orthogonal and have unit length i.e. $\mathbf{U}^T\mathbf{U} = \mathbf{V}^T\mathbf{V} = \mathbf{I}$.

element of the vector is the frequency of a in the bag, the second element is the frequency of b in the bag, and the third element is the frequency of c in the bag. Like in this example where the bag $\{a, a, b, c, c, c\}$ becomes $\mathbf{x} = [2, 1, 3]$.

A set of bags can be represented as a matrix $\mathbf{X}$ where columns are vectors representing bags and rows are terms (words' frequencies) so each element x_{ij} is the frequency of the i-th term in the j-th bag.

The basis for applying VSM to information retrieval is the *bag of word hypothesis* [69]. The belief expressed by the hypothesis is that a column vector in a term-document matrix captures (to some extent) the meaning underlying the document; what the document is about [77].

Vectors in a term-document matrix are obviously an over simplified representation of the original documents; they just tell us about the frequency of words while their original order is lost. Sentences, paragraphs, chapters, the whole structure of the document is lost as well. Nevertheless, there are countless applications of VSM that work surprisingly well. Despite their crudeness, vectors seem to capture a relevant aspect of semantics.

Salton et al [69] provide an intuitive justification for the meaning of the term-document matrix; the topic of a document will influence the author and it will be reflected by his choice of words when writing the document. Put simply, when two column vectors show similar patterns of numbers then the two documents have similar topics.

3.4 Latent Semantic Analysis

In 1998 Dumais et al. published a paper [22] where they proposed latent semantic analysis (LSA) as a new approach for dealing with the vocabulary problem in human-computer interaction. The main objective of LSA is to address the limitation of VSM-based approaches where the retrieval of textual documents depends on a lexical match between words in users' requests and the documents themselves. Synonymy and polysemy [4], which are classical problems arising when dealing with natural language, cannot be handled through the vector space representation because it assigns a separate dimension to each term (that is terms are considered unrelated).

LSA overcomes these limitations by applying J.R. Firth's notion about the meaning of words [20] exemplified by his renowned sentence: "You shall know a word by the company it keeps" or, in other words, a word's meaning is given by the

4. Synonymy occurs when different words refer to the same meaning, polysemy when a word can have multiple meanings.

words that frequently appear close-by. Let us see how LSA implements this concept about the meaning and similarity of words. LSA's input is a term-document matrix and, similarly to PCA, LSA employs truncated SVD to create a reduced space approximating the term to document association data: $\mathbf{X} \approx \mathbf{X}_k = \mathbf{U}_k \mathbf{\Sigma}_k \mathbf{V}_k^T$. Since the number of dimensions in the reduced model (k) is much smaller than the number of unique words, minor differences in terminology are ignored. In this reduced space, the closeness of documents is determined by the overall pattern of term usage, so documents can be classified together regardless of the precise words that are used to describe them, and their description depends on a kind of consensus of their term meanings. As a result, *terms that did not actually appear in a document may still end up close to it, if that is consistent with the major patterns of association in the data* [22]. Landauer et al. [38] describe truncated SVD as a method for discovering *high-order* co-occurrence. Direct co-occurrence (first-order co-occurrence) is when two words appear in identical contexts; they are typically nearby each other. Indirect co-occurrence (high-order co-occurrence) is when two words appear in *similar* contexts; they have similar neighbors. Similarity of contexts may be defined recursively in terms of lower-order co-occurrence. Lemaire et al. [39] demonstrate that truncated SVD can discover high-order co-occurence.

After our introduction to PCA and LSA one might wonder whether PCA and LSA are just the same thing. It is true that they share the same factorization technique and low-ranking step but the input data is different; in PCA we apply SVD to either a covariance or a correlation matrix (in the next chapter we will see more details about this through examples), while in LSA we apply SVD to a term-document matrix. So, since the input matrix is different then PCA and LSA compute different things.

3.5 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 Prg composed of a collection of units of code[5] $\{u_i\}_{i \in 1 \ldots n}$, the CG of Prg is a graph where each node is associated with a unit u_i. There can be an arc going from u_i to u_j if and only if u_j is called by u_i at some location of the source code. If there is a loop then this cycle represents a recursive call. CGs are in the general case multigraphs because in addition to recursive calls they model also multiple calls. In this case we have multiple arcs between u_i and u_j.

Call graphs can be created either dynamically or statically. A dynamic call graph is created by tracing the execution of a program usually requiring some code instrumentation. Dynamic CGs are exact but they describe only one run of exe-

5. Either function or methods.

cution while static ones are an over-approximation of the real CG because they can represent calls in the graph that might never occur during the execution of the program.

A static call graph requires the analysis of source code. In languages like `Java` or `C++` the identification of the caller-callee relation can be difficult because of polymorphism. Also in C programs the usage of function pointers makes the creation of CG challenging. In such contexts, if a higher accuracy is required, one can employ *alias analysis* [6] results.

Conclusion

This chapter provides a detailed introduction to the main basic elements of FEAT. Hierarchical clustering algorithms' outcome is strongly affected by both the distance metric and merging criterion driving the creation of clusters; this implies that their choice should be determined by some domain knowledge of the problem at hand.

When clustering is applied to software systems is called software clustering. Software clustering creates group of entities like: classes, modules, functions, etc. Approaches to software clustering are mainly divided into two groups: semantic and structural according to the kind of information they target to. FEAT uses both approaches based correspondingly on natural text contained in source code and the structure modeled through the call graph (Chap.4 will present this in great details).

Measuring the semantic proximity of natural language texts in an automated way can be achieved through the vector space model. One serious limitation of the VSM-based approaches is related to the lexical match expected among words in documents to evaluate them as similar. Latent semantic analysis can overcome this limitation by the so-called high-order co-occurrence where dimensions are not represented by single words anymore but by contexts of usage of words.

Call Graphs are graphs representing the caller-callee relationship in programs and they can appropriately be used to represent the structure of a software system. Several properties of functions can be obtained through the analysis of call graphs but if one needs to retain just the most relevant ones in a given context then principal component analysis can be helpful. PCA is a dimensionality reduction technique but at the same time can highlight the most relevant dimensions in a multi-dimensional space and in the rest of the thesis (Chap.4) will be showed its usage in FEAT.

6. Alias analysis helps to detect whether a given memory location can be accessed in more than one way.

IV

FEAT Approach

In this chapter we provide a detailed explanation of our approach, FEAT, starting from its origins then presenting a general overview and detailing each one of its components.

4.1 Genesis of FEAT

At the very beginning of this thesis' work we went through many papers addressing various challenges in the area of program comprehension and it can be helpful to go through the main differences between FEAT and those approaches (see Sec. 2.3) because it explains why we made certain decisions.

Instead of using either source code or source documentation, FEAT combines them in a single perspective through a novel distance metric merging semantics and structural elements contained in source code. Many approaches use just one of them, entailing that some key information will be lost. In Chap. 6 we will see through experiments on real projects what are the advantages of considering both elements at the same time.

Some methods require human assistance under several respects. Just to mention a couple of examples: LDA based methods require the input of statistical parameters which are hard to define beforehand, end users have to select one partition of clusters among many others in hierarchical clustering based methods, supervised learning approaches require the labeling of training examples which is time demanding and prone to subjectivity issues. Differently to these approaches FEAT

is fully automated, it applies criteria which are formally defined, and its output can be directly used for discovering and locating features, tracing source code back towards other artifacts like requirements, test cases, etc. FEAT makes use of hierarchical clustering as well, but it solves the problem of selecting a partition exploiting its hybrid perspective on source code.

To sum up the characteristic traits of FEAT:

- Program topoi are concrete structures, supported by a formal definition, useful to address the challenges of automated program comprehension.
- Whereas in program understanding there is a distinction between approaches either for feature discovery or feature location, FEAT removes this distinction offering one single approach for both tasks.
- It is based on a model of software systems that does not require a specific programming language in order to be applied.
- It requires no additional input other than source code.
- It is fully automated.

let us now give a closer look to FEAT.

4.2 A General Overview of FEAT

Understanding a software system through its source code can be pursued following two conceptual approaches: structural and semantic. Structure-based approaches focus on the static relationships among entities while semantic ones include all aspects of a system's domain knowledge that can be obtained from comments and identifier names [73]. Discovering the main capabilities of a software system can benefit from structural information that can be used to identify a capability as a set of structurally *close* code units [1] contributing to its implementation. On the other hand, under the semantic view, those parts of a system showing commonalities in terms of natural language words can be considered part of a system's capability as well.

In other words, structural and semantic approaches convey two different, both valuable, perspectives of a system and FEAT combines them in order to achieve a more accurate picture. FEAT, whose main elements are depicted in Fig.4.1, is based on a three steps process: preprocessing (box noted 2), clustering (3, 4)

1. In the whole thesis we refer to both functions (in procedural languages) and methods (in object oriented languages) with the generic term *code unit*.

and, entry-point selection (5). The input to FEAT is a software system (noted Sw) considered as its source code and comments (1). In the preprocessing step (2), FEAT parses source code and comments creating a representation of the system which supports the twofold assumption lying behind the approach. Hence, if Sw is a software system counting n code units its representation can be defined through the call graph of Sw and the set of unit-documents $\mathcal{D}$ as follows:

Definition 4.1. (FEAT model) $\mathrm{Sw} \triangleq \langle \mathrm{CG}, \mathcal{D} \rangle$

The first element (structural) of the pair is the call graph $\mathrm{CG} = (\mathcal{U}, \mathcal{E})$ where $\mathcal{U} = \{u_1, u_2, \cdots, u_n\}$ is the set of units and $\mathcal{E}$ the set of edges representing the caller-callee relationship. The second one (semantic) is the set of *unit-documents*: $\mathcal{D} = \{\mathbf{d}_1, \mathbf{d}_2, \cdots, \mathbf{d}_n\}$.

4.3 Semantic Perspective over Code Units

The creation of the semantic part of FEAT requires that, for every unit, we extract the following elements:

- Referenced code unit names.
- Variable names.
- String literals [2].
- Comments; both those immediately preceding and within the body of a unit.

All these elements contribute to the creation of a set of textual documents; one for each unit. This preliminary text, where the original words' order is ignored (bag of words), undergoes several transformations (tokenization, stop words removal, stemming, weighting) producing a *unit-document* noted as $\mathbf{d}_u = [w_1, w_2, \cdots, w_m] \in \mathrm{I\!R}^{\mathrm{m}}$ where w_i is a real number representing the weight associated to the *i*th-word.

All words selected by scanning source code are used to create an alphabetically sorted index $\mathcal{V}$, which is the set of words of the analyzed system. So, given an index $\mathcal{V}$ of size m, $\mathbf{d}_u \in \mathrm{I\!R}^m$ denotes the vector representing a unit-document u. The process going from source code to unit-documents is detailed as follows:

1. *Parsing*. For every unit, FEAT parses the source code extracting comments

2. Literals are quoted sequences of characters representing string values i.e. `x="foo"`.

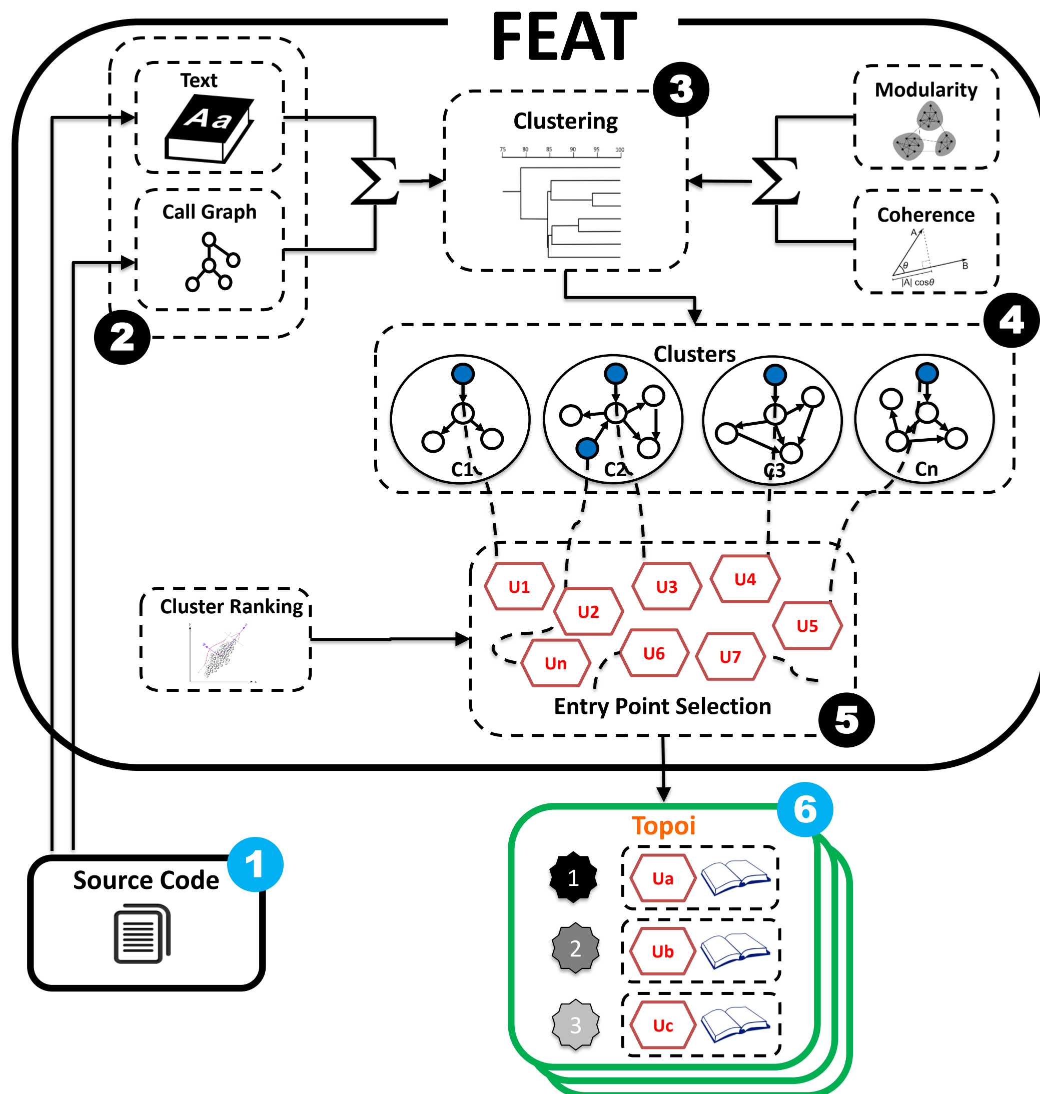

Figure 4.1 – FEAT *process's overview*

preceding the unit (conventionally these remarks contains sentences about the following unit's purpose), comments within the unit, literals, variable names and names of referenced (called) units.

2. *Tokenization*. Tokenization is the process of breaking up a given text into units called tokens. Tokens may be words, numbers or punctuation marks. Identifiers in source code usually follow some kind of convention (that is *camel case*[3] and C-style), to deal with these compound symbols we added a further step to the standard tokenization any time we reveal the occurrence of these widespread conventions (e.g., `MACOS_print` is decomposed into

3. Camel case is a code writing convention where compound words are written capitalizing each initial word in the middle of the sentence like in `printFile`, `openHtmlDocument`, etc.

`macos` and `print`, `saveFile` into `save` and `file`). To avoid duplications all words in the dictionary are in lowercase.

3. *Stemming*. Inflected words are brought back to their root (e.g., "cars", "car's", "cars" ⇒ "car" or "fishing", "fished", and "fisher" to the root word "fish").

4. *Stop word removal*. Useless tokens and language-specific keywords are removed (e.g., "and", "if", "else", "while", etc) as they do not bring any value in terms of units' semantics.

5. *Weighting*. Basically a weighting scheme translates words into numbers. The simplest scheme is counting the occurrences of a word in a document (term frequency) but the drawback of such a simple approach is that highly common, unspecific words will affect the ranking of a document. To counteract this issue a composite weighting scheme is used, namely *Term Frequency Inverse / Document Frequency* (tf-idf)[44]; the idf part of the scheme diminishes the weight of terms that occur very frequently in the document set and increases the weight of terms that occur rarely. For each extracted word t into a given unit u, $\mathrm{tf}(t,u)$ counts the number of occurrences of t in u, while $\mathrm{idf}(t,\mathcal{U}) = \log \frac{|\mathcal{U}|}{|\{u \in \mathcal{U} | t \in u\}|}$ is the logarithm of the inverse fraction of the number of units in which the word t occurs. Then, using $\text{tf-idf}(t,u,\mathcal{U}) = \mathrm{tf}(t,u)\mathrm{idf}(t,\mathcal{U})$, the vector for a unit-document is defined as follows:

$$\mathbf{d}_u = [\text{tf-idf}(t_1,u,\mathcal{U}),\ldots,\text{tf-idf}(t_m,u,\mathcal{U})] \tag{4.1}$$

Computing the Semantic Distance

The semantic distance between units is computed by means of LSA (see Sec. 3.4). In order to apply LSA, we need to create the term-document matrix:

$$\mathbf{C}_{m,n} = \begin{pmatrix} d_{1,1} & d_{1,2} & \cdots & d_{1,n} \\ d_{2,1} & d_{2,2} & \cdots & d_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ d_{m,1} & d_{m,2} & \cdots & d_{m,n} \end{pmatrix}$$

where each (i,j) element describes the occurrence (weighted according the aforementioned process) of word i in unit-document j. Hence, every column of $\mathbf{C}$ is a vector $\mathbf{d}_j$ corresponding to the unit-document of the unit u_j (see Eq.4.1) and every row is a vector representing the relation of a term t_i to each unit-document: $\mathbf{t}_i^T = [d_{i,1},\ldots,d_{i,n}]$.

Through the application of SVD we obtain the following factorization: $\mathbf{C} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$. Without going into deep mathematical details, we can say that $\mathbf{U}$ describes the relationship between words (rows) and concepts (columns). $\mathbf{\Sigma}$ is a diagonal matrix whose elements $(\sigma_1,\sigma_2,\cdots,\sigma_n)$ are sorted in decreasing order of values and

they represent the relative strength of every concept with respect to the overall space.

$$\mathbf{\Sigma} = \begin{pmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_n \end{pmatrix}$$

Finally, we have $\mathbf{V}^T$ describing the relationship between concepts (rows) and unit-documents (columns).

The last step is low-rank approximation. Let us assume that the rank of $\mathbf{C}$ is r then *shrinking* it to a number of dimensions $k < r$ will accomplish the step. The choice of a value for k is key and it is done as follows. $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n$ represents the relative relevance among concepts and we define the *cumulative concepts relevance* (ccr) of the first k concepts as:

$$\mathrm{ccr}(k) = \frac{\sum_{i=1}^{k} \sigma_i}{\mathrm{tr}(\mathbf{\Sigma})}$$

$K \in (0, 1)$ is the desired cumulative concepts relevance. Hence, we can now compute k by fixing K and solving the constraint

$$\begin{aligned} \underset{k}{\text{minimize}} \quad & \mathrm{ccr}(k) \\ \text{s.t.} \quad & \mathrm{ccr}(k) \geq K, \\ & k < r. \end{aligned} \tag{4.2}$$

The solution of the constraint requires linear time on k. Hence the low rank approximation to $\mathbf{C}$ is: $\mathbf{C}_k = \mathbf{U}_k \mathbf{\Sigma}_k \mathbf{V}_k^T$.

The element of the truncated SVD factorization used for the computation of the distance is:

$$\mathbf{V}_k^T = \begin{pmatrix} d_{1,1} & d_{1,2} & \cdots & d_{1,n} \\ d_{2,1} & d_{2,2} & \cdots & d_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ d_{k,1} & d_{k,2} & \cdots & d_{k,n} \end{pmatrix}$$

$\mathbf{V}_k^T$ contains the unit-documents' projections into the concept space. The semantic distance between two code units can now be computed with the angular distance[15]:

$$\mathrm{d}_{\mathcal{D}}(u_a, u_b) = \frac{\arccos(\mathrm{sim_c}(u_a, u_b))}{\pi} \tag{4.3}$$

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

$$\mathrm{sim_c}(u_a, u_b) = \frac{\mathbf{d}_{u_a} \cdot \mathbf{d}_{u_b}}{\|\mathbf{d}_{u_a}\| \|\mathbf{d}_{u_b}\|}$$

and $\mathbf{d}_{u_a}$, $\mathbf{d}_{u_b}$ are column vectors of $\mathbf{V}_k^T$. Angular distance is a proper distance metric bounded between 0 and 1 inclusive.

Many texts about information retrieval propose the Euclidean distance as a viable metric for assessing documents' similarity. The problem with Euclidean distance, when working with documents having uneven lengths, is that it is affected by the number of occurrences of words; documents sharing many words but differing in terms of their occurrences would be classified as not similar by the Euclidean distance. Cosine similarity overcomes this problem because it is not sensitive to the magnitude of vectors since it depends only on the angle between them. A simple working example will make the whole process more clear.

Example

Let us consider that we have extracted the following text from five units contained in the source code of a software system.

1. u_1: *Open file through a modal window*
2. u_2: *Print text file or save it as PDF*
3. u_3: *Copy selected text into the clipboard*
4. u_4: *Paste text into the editor*
5. u_5: *Cut selected text and put it into the clipboard*

The documents undergo the process described earlier (for the sake of simplicity we run just stop word removal) producing the following dictionary:
$\mathcal{V} = \{$clipboard, copy, cut, editor, file, modal, open, paste, pdf, print, put, save, selected, text, window$\}$ ($|\mathcal{V}| = 15$).

The term-document matrix is then:

$$\mathbf{C} = \begin{array}{c} \begin{matrix} \vec{u_1} & \vec{u_2} & \vec{u_3} & \vec{u_4} & \vec{u_5} \end{matrix} \\ \begin{pmatrix} 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array} \begin{array}{l} \\ \text{clipboard} \\ \text{copy} \\ \text{cut} \\ \text{editor} \\ \text{file} \\ \text{modal} \\ \text{open} \\ \text{paste} \\ \text{pdf} \\ \text{print} \\ \text{put} \\ \text{save} \\ \text{selected} \\ \text{text} \\ \text{window} \end{array}$$

we computed only term frequencies; again to keep the example more readable. The next steps are: computing SVD factorization and low-rank approximation. $\mathbf{C}$ has rank 5, let us directly fix $k = 2$ and compute the low-rank approximation $\mathbf{C} \approx \mathbf{C}_2 = \mathbf{U}_2\mathbf{\Sigma}_2\mathbf{V}_2^T$. Recall that $\mathbf{V}^T$ represents the set of coordinates of unit-documents in the concept space, which after the *shrinking* to two dimensions, has now become:

$$\mathbf{V}_2^T = \begin{pmatrix} 0.09 & 0.42 & 0.56 & 0.28 & 0.64 \\ -0.61 & -0.67 & 0.25 & -0.04 & 0.33 \end{pmatrix}$$

Documents are now expressed as column vectors in $\mathbf{V}_2^T$ and they can also be shown on a diagram as in Fig. 4.2. We can query the concept space with expressions like "clipboard". To accomplish this we first create a query document

$$\mathbf{q} = [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$$

and then we map it into the LSA space by the transformation [44]:

$$\mathbf{q}_k = \mathbf{\Sigma}_k^{-1}\mathbf{U}_k^T\mathbf{q}$$

so that we can apply Eq.4.3.

Fig. 4.2 shows both unit-documents' vectors ($\mathbf{d}_1, \ldots, \mathbf{d}_5$) and query's one ($\mathbf{q}$). One can easily notice that vectors $\mathbf{d}_3$ and $\mathbf{d}_5$ are the closest ones to the query and, indeed, their unit-documents contain a direct reference to "clipboard" but $\mathbf{d}_4$, whose unit-document does not contain it, is quite close to the query as well. This is the indirect co-occurrence capability of LSA; u_3, u_4 and, u_5 have similar neighborhood which is the word "text". u_2 contains the word "text" as well but its

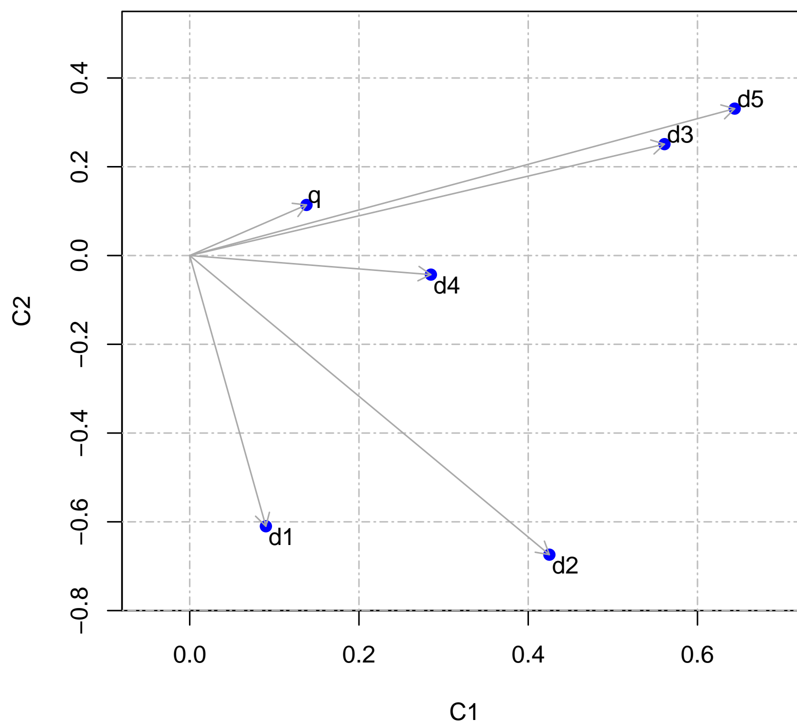

Figure 4.2 – *Geometric representation of unit-documents' vectors* $\mathbf{d}_1, \ldots, \mathbf{d}_5$ *and query vector* $\mathbf{q}_k$ *of the example in Sec.4.3*

neighbors make it less similar to the query. This is something that a human user would expect; u_3, u_4 and, u_5 deal with clipboard management having, as it is shown in Tab. 4.1, a lower distance with the query respect to u_2 which implements file printing.

Document	$\mathrm{d}_{\mathcal{D}}(\mathbf{q}_k, \cdot)$
$\mathbf{d}_5$	0.068
$\mathbf{d}_3$	0.086
$\mathbf{d}_4$	0.268
$\mathbf{d}_2$	0.540
$\mathbf{d}_1$	0.673

Table 4.1 – *Distances between unit-documents' vectors* $\mathbf{d}_1, \ldots, \mathbf{d}_5$ *and query vector* $\mathbf{q}_k$

The concept of similarity is not limited to documents. The words in the concept

space are represented by the row vectors of $\mathbf{U}_k$.

$$\mathbf{U}_2 = \begin{pmatrix} 0.41 & 0.26 \\ 0.19 & 0.11 \\ 0.22 & 0.15 \\ 0.10 & -0.02 \\ 0.18 & -0.57 \\ 0.03 & -0.27 \\ 0.03 & -0.27 \\ 0.10 & -0.02 \\ 0.14 & -0.30 \\ 0.14 & -0.30 \\ 0.22 & 0.15 \\ 0.14 & -0.30 \\ 0.41 & 0.26 \\ 0.65 & -0.06 \\ 0.03 & -0.27 \end{pmatrix}$$

LSA allows us to assess how close, in terms of meaning in a given context, words are. We plotted the words in the concept space of our example in Fig. 4.3 and it is interesting to notice how words like: "copy", "cut", "paste", "clipboard", etc. even without providing any prior domain knowledge, but just on the basis of high-order co-occurrences are considered similar (recall that our notion of distance is based on angles' cosine not on Euclidean distance.).

4.4 Structural Perspective over Code Units

The semantic distance between unit-documents given in equation (4.3) can be complemented by the addition of structural information available in the call graph of a software system under examination. The key element for the computation of the structural distance is the call graph. In FEAT we do not need all the characteristics of the complete call graph then some of them have been removed. First, the call graph is a multi-graph which means that between two nodes there can be more than one edge in case of multiple calls. Since we want to assess only whether a dependency between two units exists or not, in FEAT we consider at most one edge between two units. Second, for reasons which are similar to the preceding point, recursive calls are not represented in CG. The call graph is a directed graph, this would prevent the fulfillment of the symmetry property (see Sec. 3.1) of a distance metric then we transform the call graph into an undirected graph. From now on all references to call graphs are related to our simplified version described here.

Given a (undirected) call graph $\mathrm{CG} = (\mathcal{U}, \mathcal{E})$, where $\mathcal{U}$ is the set of units and $\mathcal{E}$ is the set of edges representing the caller-callee relationship, the distance between

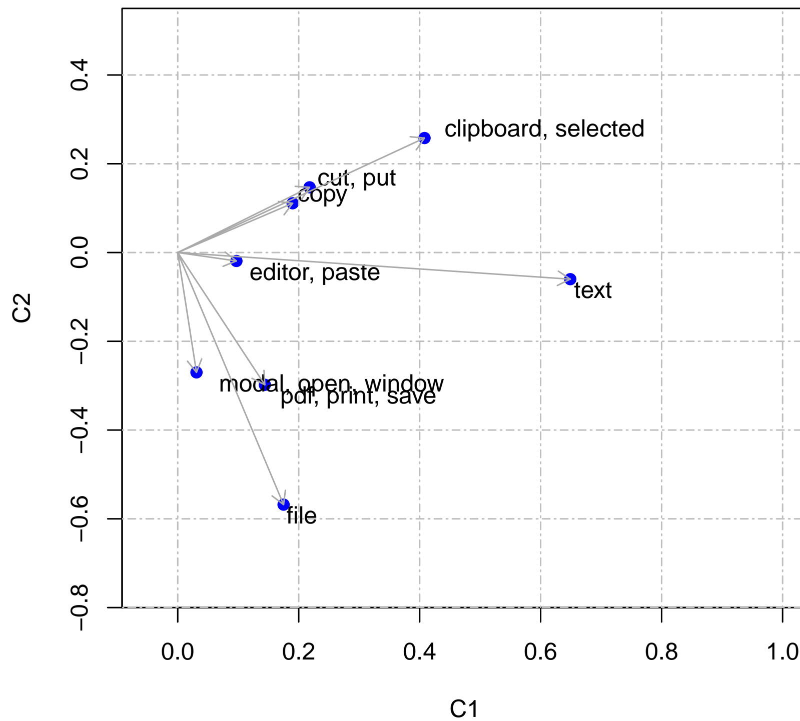

Figure 4.3 – *Geometric representation of unit-documents' words in the concept space of the example in Sec.4.3*

two units u_a and u_b can be computed by using the length of a shortest path between them. Let $\pi(u_a, u_b) = \langle e_1, \dots e_k \rangle$, where each e_i is an edge, be a shortest path between u_a and u_b, and $|\pi(u_a, u_b)| = k$ be the length of that path then the distance between u_a and u_b is:

$$\mathrm{d}(u_a, u_b) = \frac{1-\lambda}{1-\lambda^D} \sum_{i=0}^{k-1} \lambda^i$$

The factor before the summation[4] makes the distance ranging in $[0, 1]$. D is the graph diameter or in other words the length of the *longest shortest* path, k is the length of the shortest path between a and b and λ, under the constraint $\lambda > 1$, sets how fast we want our distance to grow; the range of graph distances we expect, even in large programs, is not so high thus we boost the distance by making it grow exponentially. The distance has to span from 0 to 1 so we normalize it. This is done by dividing it by the maximum distance in a given graph that is: $\frac{1-\lambda^{\mathrm{D}}}{1-\lambda}$. If there is no path between u_a and u_b then $|\pi(u_a, u_b)| = \infty$. The complete definition

4. This is the reciprocal of the summation of a (slightly modified) geometric series.

of the structural distance $\mathrm{d_{CG}}$, covering all cases, is as follows:

$$\mathrm{d_{CG}}(u_a, u_b) = \begin{cases} 0 & \text{if } u_a = u_b \\ \frac{1-\lambda}{1-\lambda^{\mathrm{D}}} \sum_{i=0}^{k-1} \lambda^i & \text{if } u_a \neq u_b \text{ and } |\pi(u_a, u_b)| = k \\ 1 & \text{if } |\pi(u_a, u_b)| = \infty \end{cases} \quad (4.4)$$

4.5 Hybrid Distance

Both $\mathrm{d}_{\mathcal{D}}$ and $\mathrm{d_{CG}}$ are proper distance measures satisfying the three axioms: symmetry, positive definiteness and, triangular inequality. On their basis we present a novel hybrid distance with the objective of producing clusters whose elements show high internal cohesion under both perspectives: structural and semantic. This combination can mitigate some unwanted effects that may occur if we use only one distance. For instance, two units sharing many words, but not connected in the call graph (or very far from each other), would be evaluated with high similarity if only $\mathrm{d}_{\mathcal{D}}$ be 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 capability should share a common vocabulary.

The hybrid distance is defined as a linear combination of $\mathrm{d}_{\mathcal{D}}$ and $\mathrm{d_{CG}}$ using a real number α ranging in $[0, 1]$:

$$\mathrm{d_{FEAT}}(u_a, u_b) = \alpha \mathrm{d}_{\mathcal{D}}(u_a, u_b) + (1 - \alpha)\mathrm{d_{CG}}(u_a, u_b) \quad (4.5)$$

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. More details about this will be provided in the experimental evaluation chapter.

4.6 Distance over Clusters of Code Units

So far we have seen how to compute the distance between single units, but HAC, in order to merge clusters of units, requires to compute the distance between sets of units. In Sec. 3.1.1 we presented some of the most relevant approaches to the merging of clusters, in FEAT we model a set of units as its centroid hence, the distance between two clusters corresponds to the distance of their centroids. Following the concept of a hybrid representation of structural and semantic elements we define a hybrid centroid distance. Unit-documents lie in a Euclidean

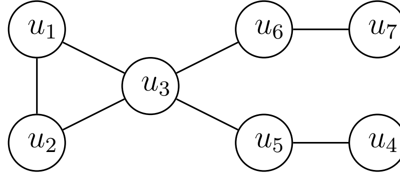

Figure 4.4 – *Graph for the example on graph medoids computation*

space then the centroid of a cluster $\mathcal{C}$ is $\mu_{\mathcal{D}}(\mathcal{C}) = \frac{\mathbf{d}_1+\mathbf{d}_2+\cdots+\mathbf{d}_n}{|\mathcal{C}|}$. Instead, the structural part of $\mathcal{C}$ is represented as a *graph medoid*. Medoids are representatives of a discrete set of elements and in FEAT they are defined as:

Definition 4.2. (Graph Medoid) Let $\text{CG} = (\mathcal{U}, \mathcal{E})$ be a call graph, $\mathcal{C} = \{u_1, u_2, \ldots, u_m\}$ a cluster of units, $|\pi(u_a, u_b)|$ a shortest path between u_a and u_b, $\sigma_{\mathcal{C}}(u) = \sum_{\forall u_i \in \mathcal{C}} |\pi(u_i, u)|$,
$\mathcal{M} = \{u \mid \arg\min_{u \in \mathcal{U}} \{\sigma_{\mathcal{C}}(u)\}\}$ and, $\overline{s} = \frac{1}{|\mathcal{C}|} \min_{u \in \mathcal{U}} \{\sigma_{\mathcal{C}}(u)\}$ then the graph medoid of $\mathcal{C}$ is:

$$\mu_{\text{CG}}(\mathcal{C}) = \arg\min_{u \in \mathcal{M}} \left(\sum_{\forall u_i \in \mathcal{C}} (|\pi(u_i, u)| - \overline{s})^2 \right)$$

In other words, the graph medoid of $\mathcal{C}$ is the unit $u \in \mathcal{U}$ lying at the most *central* position w.r.t. all units in $\mathcal{C}$.
Finally, the hybrid distance between two clusters $\mathcal{C}_i$ and $\mathcal{C}_j$ is:

$$\text{d}_{\text{FEAT}}(\mathcal{C}_i, \mathcal{C}_j) = \alpha \text{d}_{\mathcal{D}} \left(\mu_{\mathcal{D}}(\mathcal{C}_i), \mu_{\mathcal{D}}(\mathcal{C}_j) \right) + (1 - \alpha) \text{d}_{\text{CG}} (\mu_{\text{CG}}(\mathcal{C}_i), \mu_{\text{CG}}(\mathcal{C}_j)) \quad (4.6)$$

Eq. 4.5 can now be seen as a special case of the more general Eq. 4.6 occurring when $|\mathcal{C}_i| = |\mathcal{C}_j| = 1$.

Example

The computation of graph medoids is easier to understand with an example. Let us consider the graph in Fig. 4.4 and its all shortest paths matrix:

$$S = \begin{array}{c} \\ \end{array}\begin{pmatrix} 0 & 1 & 1 & 3 & 2 & 2 & 3 \\ 1 & 0 & 1 & 3 & 2 & 2 & 3 \\ 1 & 1 & 0 & 2 & 1 & 1 & 2 \\ 3 & 3 & 2 & 0 & 1 & 3 & 4 \\ 2 & 2 & 1 & 1 & 0 & 2 & 3 \\ 2 & 2 & 1 & 3 & 2 & 0 & 1 \\ 3 & 3 & 2 & 4 & 3 & 1 & 0 \end{pmatrix} \begin{matrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \end{matrix}$$

(columns: $u_1 \; u_2 \; u_3 \; u_4 \; u_5 \; u_6 \; u_7$)

Assume that we want to compute the graph medoids of $\mathrm{W} = \{u_4, u_7\}$ then the needed steps are the following:

1. $\sigma(u_i)_{\forall i \in 1..7} = \{3+3, 3+3, 2+2, 0+4, 1+3, 3+1, 4+0\}$
   $\sigma(u_i)_{\forall i \in 1..7} = \{6, 6, 4, 4, 4, 4, 4\}$
2. $\mathrm{M} = \{u_3, u_4, u_5, u_6, u_7\}$
3. $\overline{s} = \frac{1}{2} \cdot 4 = 2$
4. $\sum_{j=1}^{m} \left(|\pi(w_j, u_i)| - \overline{s}\right)^2_{\forall i \in \{3,4,5,6,7\}} = \{0, 8, 2, 2, 8\}$
5. $\mu_{\mathrm{CG}}(\mathrm{W}) = \{u_3\}$

Then the set of graph medoids of $\{u_4, u_7\}$ is reduced to a singleton $\mu = \{u_3\}$ which, in this simple example, could have been guessed directly on the graph.

4.7 Selecting a Partition in HAC

HAC does not require a prespecified number of clusters, it creates a hierarchy of them through an iterative process producing a new partition of clusters at every merging step. The leaves of this hierachy are cluster singletons and the root is a cluster including all data points. This structure is usually called *dendrogram*. A dendrogram is a tree, every node in the tree is a merging point of either individual data points or clusters. The length of the branches is proportional to the distance of the two merged elements (see Fig. 3.1).

In our application of HAC we are mainly interested in one specific disjoint set of clusters not in the whole hierarchy, hence we need a criterion to cut the hierarchy

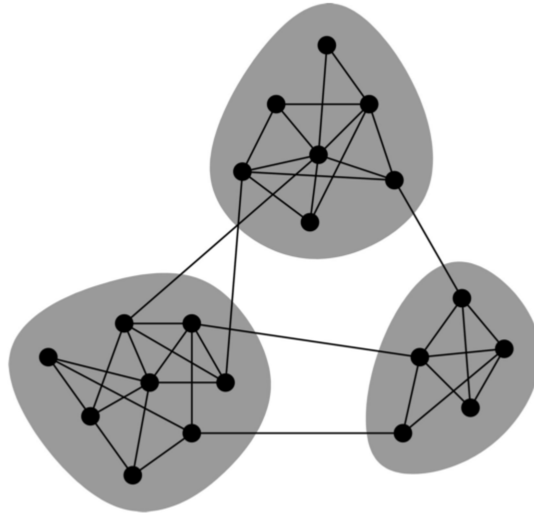

Figure 4.5 – *Illustration of how a graph can be divided into clusters having inter-cluster sparsity and intra-cluster density*

at some point. The choice of a partition $\mathcal{P} = \{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_m\}$ among the $n-1$ possible ones (where n is the number of data points) depends on the objectives of the domain at hand [44]. Following the idea of a combination of structural and semantic aspects of source code, we introduce our *hybrid cutting criterion* made again by a structural and semantic part. Let us start with the structural part.

4.7.1 Modularity

In social sciences, biology, computer networks etc. there is a big interest around the detection of modules (also called communities, groups or clusters). *Modularity* is a measure of the division of a network into modules. By definition modules are partitions showing exactly the two properties we are looking for in clusters of units: inter-cluster sparsity and intra-cluster density. The modularity function of Newman and Girvan [59] is:

$$\mathrm{Q}(\mathcal{P}) = \frac{1}{2m} \sum_{i,j} \left(A_{i,j} - \frac{k_i k_j}{2m} \right) \delta(\mathcal{C}_i, \mathcal{C}_j) \tag{4.7}$$

where A is the adjacency matrix [5], k_i (resp. k_j) is the degree of node i (resp. j), $\mathcal{C}_i$ (resp. $\mathcal{C}_j$) is the cluster of i (resp. j), and m is the total number of edges. Function δ is the Kronecker delta defined as:

$$\delta(\mathcal{C}_i, \mathcal{C}_j) = \begin{cases} 1 & \text{if nodes } i,j \text{ are in the same cluster} \\ 0 & \text{otherwise} \end{cases}$$

5. $A_{ij} = 1$ if there exists an edge between vertices i and j and $A_{ij} = 0$ otherwise.

In other words, given a certain partition of the nodes of a graph in a number of clusters, modularity is defined as the fraction of edges that falls within the given clusters minus the expected fraction if edges were distributed at random.

The idea of modularity-based cluster detection, studied in the context of social networks [17], is to find partitions that 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 [1]. A potential strategy we can pursue is again related to the informational content conveyed by the call graph. So, in our context we apply modularity in order to obtain a clustering of densely connected sub-graphs of the call graph.

After several experiments on modularity (see Fig. 4.6) over software systems we observed the following behavior: at each iteration, modularity gradually grows until it reaches a maximum. 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 $\mathrm{d}_{\mathrm{FEAT}}$ distance, provides us with a set of clusters whose units show high structural regularities.

Higher values of the modularity function indicate a better community structure. The idea of modularity-based cluster detection, applied to FEAT, therefore is to find the partition $\mathcal{P} = \{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_m\}$ that maximizes Eq. 4.7. These clusters are considered to represent the optimal community structure for the given software system.

4.7.2 Textual Coherence

The semantic part of FEAT cutting criterion exploits a measure adopted in natural language processing (NLP): textual coherence [21]. This measure is 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. We expect developers, within the context of the units participating in the implementation of a system capability, to use a consistent language revealed through the choice of names for variables, the text in comments, etc. So, while looking at clusters as they were textual documents, we want to find the partition showing the highest coherence defined as:

$$\mathrm{H}(\mathcal{P}) = \sum_{\forall \mathcal{C} \in \mathcal{P}} \left(1 - \frac{2}{|\mathcal{C}|(|\mathcal{C}| - 1)} \sum_{k=1}^{|\mathcal{C}|} \sum_{j=k+1}^{|\mathcal{C}|} \mathrm{d}_{\mathcal{D}}(u_k, u_j) \right) \tag{4.8}$$

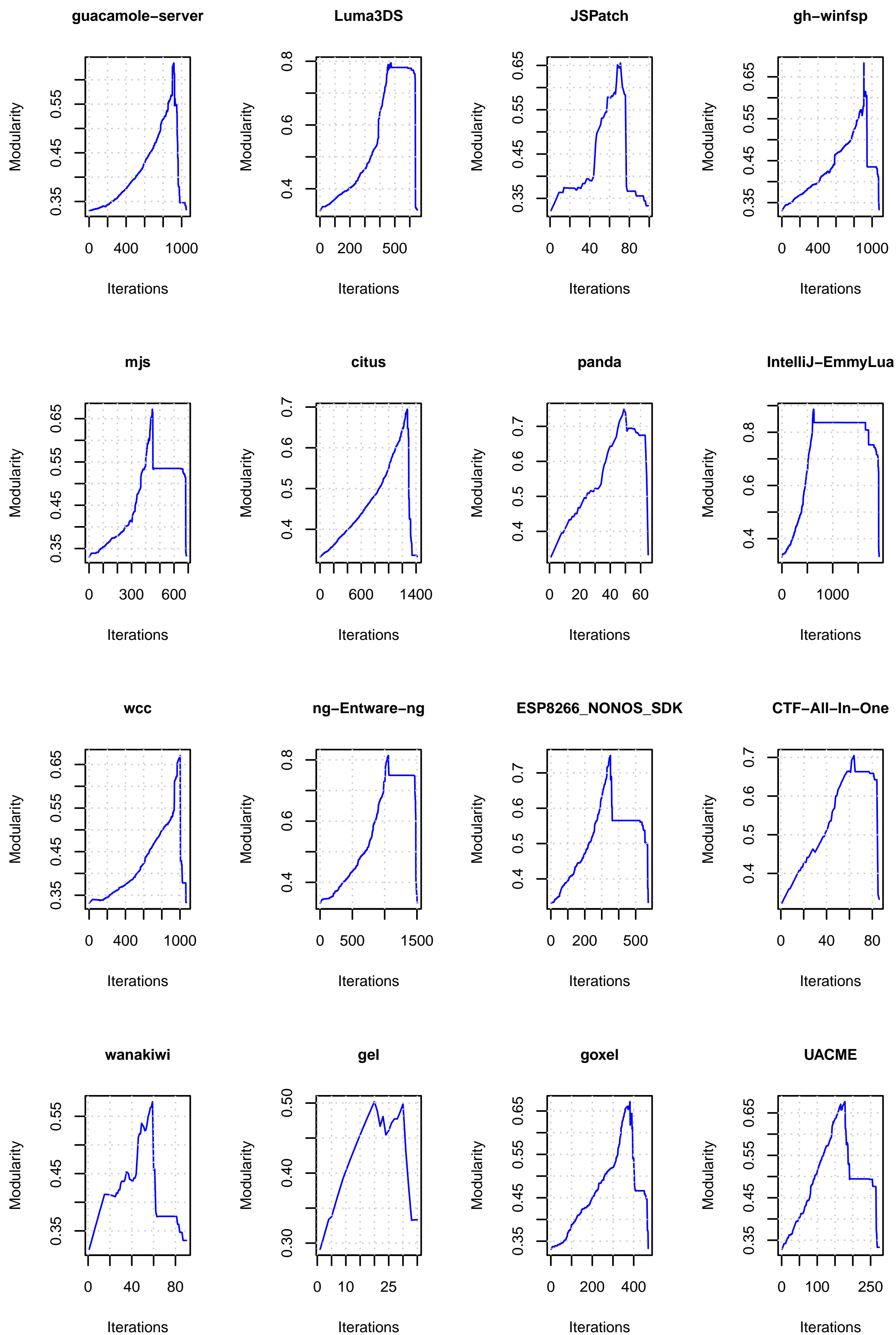

Figure 4.6 – *Graphics showing modularity values over clustering iterations across several projects.*

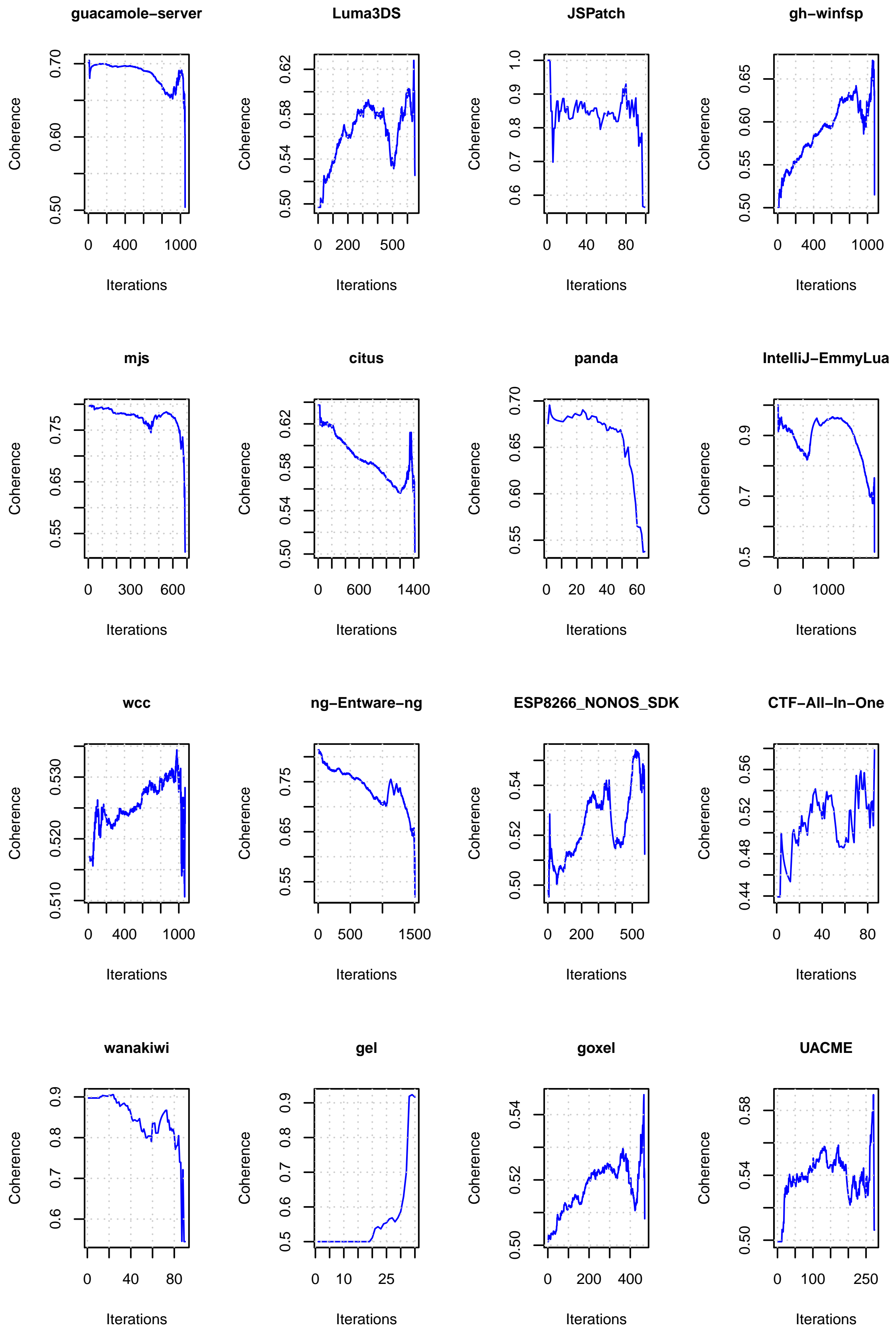

Figure 4.7 – *Graphics showing coherence values over clustering iterations across several projects.*

Fig. 4.7 shows the results obtained after running experiments on coherence over the same set of projects used for the experiment on modularity. Here the behavior does not always show the same pattern encountered with modularity. This might be explained by the fact that some development teams may pay less attention to text consistency in documenting code which leads to the observed trends.

4.7.3 FEAT Cutting Criterion

Finally, we can define a new hybrid measure as the combination of normalized coherence and modularity:

$$\mathrm{T_{FEAT}}(\mathcal{P}) = \alpha \frac{\mathrm{H}(\mathcal{P})}{|\mathcal{P}|} + (1-\alpha)\frac{2\mathrm{Q}(\mathcal{P})+1}{3} \quad (4.9)$$

The cutting criterion is then determined as the HAC's iteration where the maximum value of $\mathrm{T_{FEAT}}$ is reached. The two measures are combined through the same parameter α used in Eq. 4.6.

4.7.4 HAC Revised Algorithm

Algorithm 1 presents our clustering step with a hybrid representation of units. We use the *priority queue* version of HAC having a time complexity of $\Theta(n^2 \log(n))$. Alg.1 starts by considering each unit as a cluster at lines 4-6. It computes the pairwise distances between clusters at lines 7-8. Then, it iteratively merges pair of clusters according to a minimal $\mathrm{d_{FEAT}}$ distance value (lines 11-12) until either a partition is reduced to one cluster or the cutting criterion is reached (i.e., $\mathrm{T_{FEAT}}$ value cannot be improved) (line 9). At each iteration, the algorithm updates the pairwise distances Δ of the new partition (lines 14-15). At the end, the algorithm returns a partition $\mathcal{P}$ of m clusters.

4.8 Entry-Points Selection

The output of HAC is a set of disjoint clusters where every cluster is made of a set of units. Recall that units are identified through their name, obtained from source code, and have an associated unit-document (see Eq. 4.1). The units within a cluster are also the nodes of the related induced subgraph of the call graph (see Fig. 4.1 noted (4)).

Algorithm 1: Priority Queue HAC with hybrid cutting criterion

```
1  In U = {u_1, ..., u_n}: n units; α ∈ [0,1], τ < 0;
2  Out P = {C_1, ..., C_m}: partition of m clusters;
3  P ← ∅; Δ ← ∅; ⟨y_p, y_c⟩ ← ⟨1, 0⟩;
4  foreach u_i ∈ U do
5  |   C_i ← {u_i};
6  |   P ← P ∪ C_i;
7  foreach C_i, C_j ∈ P : i < j do
8  |   Δ ← Δ ∪ d_FEAT(C_i, C_j, α);
9  while (|P| ≠ 1) ∧ (y_c − y_p > τ) do
10 |   pick C_i, C_j ∈ P s.t., d_ij = min(Δ);
11 |   C_i ← C_i ∪ C_j;  P ← P \ C_j;
12 |   Δ ← Δ \ d_i*;  Δ ← Δ \ d_*i;
13 |   y_p ← y_c;  y_c ← T_FEAT(P);
14 |   foreach C_j ∈ P : j ≠ i do
15 |   |   Δ ← Δ ∪ d_FEAT(C_i, C_j, α);
16 |   return P
```

During the early stages of FEAT's design, we tried to use directly the partition provided right after the clustering step but what we observed through our experiments is that clusters' content is too rough to be used as program topoi. For example, they contain units which are just very specific primitives which cannot be used as main capabilities' representatives. So, we developed the idea that not all units are equal; some of them are *special*, more useful in terms of representativeness of a system and we called them *entry-points*. They can be considered related to the more general concept of *beacon* used in program comprehension (see Sec. 2.2). Put simply, entry-points are units giving access to the implementation of observable system functionalities, such as handlers for user visible characteristics in desktop applications, public methods of a software library, etc. Identifying entry-points among thousands of units is challenging. We present here some general assumptions that allow us to define a formal procedure for the selection of entry-points.

Let us provide some definitions. A cluster is defined as: $\mathcal{C} = \{u_1, u_2, \ldots, u_n\}$ and its induced subgraph[6] is: $\mathrm{CG}_{\mathcal{C}} = (\mathcal{C}, \mathcal{E}_{\mathcal{C}})$ where $\mathcal{E}_{\mathcal{C}}$ is the set of edges of the call graph connecting pairs of code units in $\mathcal{C}$. Recall, here we look for code units covering a special role in the call graph. Thinking to the examples mentioned above, entry-points are expected to implement more complex (high level) capabilities and, as such, they employ many other units generating deeper and wider calling trees respect to usual units. Conversely, entry-points are expected neither

6. An induced sub graph is another graph made by a subset of the vertices of the original graph and all the edges connecting pairs of vertices in that subset.

to participate in many diverse calling paths nor to end long calling chains. This properties of entry-points can be summarized as follows:

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 are called in short calling chains;

d) Entry-points calls originate long calling chains;

Based on the sub-graph associated to each cluster (see Fig.4.1 box noted (4)), these considerations can be translated into the six following node attributes (letters in parentheses show the relationship with the properties listed above) of the unit u:

1. Output degree (b) $\deg^+(u)$: number of outgoing arcs of u;

2. Input Degree (a) $\deg^-(u)$: number of incoming arcs of u;

3. Output Reachability (b) $\mathrm{RO}(u)$: number of paths having u as source;

4. Input Reachability (a) $\mathrm{RI}(u)$: number of paths having u as destination;

5. Output Path Length (d) $\mathrm{SO}(u)$: Sum of the lengths of all paths having u as source;

6. Input Path Length (c) $\mathrm{SI}(u)$: Sum of the lengths of all paths having u as destination;

Let's take a closer look to all these attributes. Given a unit u, input and output degrees represent correspondingly the number of incoming and outgoing arcs of u. They tell us about the number of local paths involving u.

Reachability is the ability to reach a vertex u_j from a vertex u_i. In other words it occurs when it exists a path from u_i to u_j. The two attributes, $\mathrm{RI}(u)$ and $\mathrm{RO}(u)$, count the number of global paths passing by u_i.

Let's make it a bit more formal. Since we need to consider whether a unit is a caller or a callee, from now on we remove the relaxation introduced for the definition of the structural distance in Eq. 4.4 and consider the call graph $\mathrm{CG}_\mathcal{C}$ as a directed graph. Given a (directed) call graph $\mathrm{CG}_\mathcal{C} = (V, E)$, the all-shortest-paths matrix $\mathbf{S} := (s_{i,j})_{n\times n}$ containing the graph distance from any vertex u_i to any vertex u_j, and a cluster $\mathcal{C}$ we can define the following attributes:
Output degree:

$$\deg^+(u_i) = \sum_{u_j \in \mathcal{C}} \delta(u_i, u_j);\ \delta(u_i, u_j) = \begin{cases} 1 & \text{if } s_{i,j} = 1 \\ 0 & \text{otherwise} \end{cases} \tag{4.10}$$

Input degree:

$$\deg^-(u_i) = \sum_{u_j \in \mathcal{C}} \delta(u_j, u_i);\, \delta(u_i, u_j) = \begin{cases} 1 & \text{if } s_{i,j} = 1 \\ 0 & \text{otherwise} \end{cases} \tag{4.11}$$

Output reachability:

$$\mathrm{RO}(u_i) = \sum_{u_j \in \mathcal{C}} \delta(u_i, u_j);\, \delta(u_i, u_j) = \begin{cases} 1 & \text{if } s_{i,j} \neq 0 \wedge s_{i,j} \neq \infty \\ 0 & \text{otherwise} \end{cases} \tag{4.12}$$

Input Reachability:

$$\mathrm{RI}(u_i) = \sum_{u_j \in \mathcal{C}} \delta(u_j, u_i);\, \delta(u_i, u_j) = \begin{cases} 1 & \text{if } s_{i,j} \neq 0 \wedge s_{i,j} \neq \infty \\ 0 & \text{otherwise} \end{cases} \tag{4.13}$$

The last two attributes deal with the length of paths passing by the vertex u_i.
Output path length:

$$\mathrm{SO}(u_i) = \sum_{u_j \in \mathcal{C}} \mathrm{d}(u_i, u_j)\delta(u_i, u_j);\, \delta(u_i, u_j) = \begin{cases} 1 & \text{if } s_{i,j} \neq 0 \wedge s_{i,j} \neq \infty \\ 0 & \text{otherwise} \end{cases} \tag{4.14}$$

where $\mathrm{d}(u_i, u_j) = s_{i,j}$ is the length of a shortest path from u_i to u_j.
Input path length:

$$\mathrm{SI}(u_i) = \sum_{u_j \in \mathcal{C}} \mathrm{d}(u_j, u_i)\delta(u_j, u_i);\, \delta(u_i, u_j) = \begin{cases} 1 & \text{if } s_{i,j} \neq 0 \wedge s_{i,j} \neq \infty \\ 0 & \text{otherwise} \end{cases} \tag{4.15}$$

4.8.1 Ranking Units through PCA

The problem now is how to use these six attributes in order to produce a list of entry-points starting from the units in a cluster. Should we look for vertices with the highest output reachability or the lowest input reachability? How about the other attributes? Should we combine them? Unfortunately, our experiments showed that it does not exist a unique best attribute or subset of attributes capable of identifying entry-points in all situations: in some cases input/output degree are fine but in some others input/output reachability are better and so on.

Since it seems like all attributes can be useful in different situations, why don't we use all of them and delegate to PCA the selection of the most relevant components? First we define the following vector:

$$\mathbf{v}_u = \begin{pmatrix} \deg^+(u) \\ \deg^-(u) \\ \mathrm{RO}(u) \\ \mathrm{RI}(u) \\ \mathrm{SO}(u) \\ \mathrm{SI}(u) \end{pmatrix} \tag{4.16}$$

whose elements are the attributes of unit u, included in cluster $\mathcal{C}$, represented as a node over the call graph $\mathrm{CG}_{\mathcal{C}}$. Now, we can look at a cluster as a space of six dimensions where every unit is a point in that space. More formally, a cluster $\mathcal{C} \in \mathcal{P}$ is represented by the matrix:

$$\underset{\forall u \in \mathcal{C}}{\mathbf{\Phi}} = [\mathbf{v}_u] \tag{4.17}$$

Before we move to the actual search for entry-points in a cluster, it is important to provide some key elements about the content of PCA's input matrix. The analysis accomplished to extract the principal components of a data set is based on the *covariance* matrix. Given two multi-variate random variables $\mathbf{X}$ and $\mathbf{Y}$, covariance provides a measure of their dependency; that is how much they vary together. A large value indicates high redundancy while two independent random variables have null covariance. Covariance is defined as $\mathrm{cov}(\mathbf{X}, \mathbf{Y}) = \mathrm{E}[\mathbf{XY}] - \mathrm{E}[\mathbf{X}]\,\mathrm{E}[\mathbf{Y}]$, then the covariance matrix is the matrix whose (i, j) entry is the covariance $\mathbf{\Sigma}_{ij} = \mathrm{cov}(\mathbf{X}_i, \mathbf{X}_j)$.

Scaling of the Data and PCA

PCA is sensitive to the scaling of the variables. For example, using variables with different units (i.e. some expressed in meter and others in kilometer) can lead to arbitrary results where principal components are strongly affected by the variables with higher variance. In these cases PCA should be based on the *correlation* matrix instead, which can be seen as the covariance matrix of standardized random vectors, defined as: $\mathrm{corr}(\mathbf{Z}_{ij}) = \mathrm{cov}(\mathbf{Z}_i, \mathbf{Z}_j)$ and $z_{ij} = \frac{x_{ij} - \mu_i}{\sigma_i}$ where μ_i and σ_i are correspondingly mean and standard deviation of the random variable $\mathbf{X}_i$. This way we transformed the original data set into one which is zero-centered and with unitary variance.

So, now the question is: how is our data distributed? Do the six attributes defined above follow some particular distribution? If it were a Gaussian distribution then we could simply compute the sample mean and sample variance but unfortunately this is not the case.

Statistical Patterns in Call Graphs

By plotting histograms of some experiments, we noticed that the six attributes described from Eq. 4.10 to Eq. 4.15 follow a common distribution law which at first sight appear to be the *power-law*. Power laws are described by the following probability distribution: $p(x) \propto x^{-\alpha}$ where α is a constant parameter of the distribution known as the *exponent* or *scaling parameter*. They are exponential distributions showing a peak followed by a peculiar long tailed shape which make

them easily recognizable. On the other hand, proving that a data sample follows a power-law can be tricky. In practice, few empirical phenomena obey power laws for all values of x. More often the power law applies only for values greater than some minimum $x_{\min}$. In such cases we say that the *tail* of the distribution follows a power law and it is mathematically defined as:

$$p(x) = \frac{\alpha - 1}{x_{\min}} \left(\frac{x}{x_{\min}} \right)^{-\alpha}.$$

Clauset et al. [13] provide a detailed analysis for the assessment of power-law distributions. In short the procedure can be summarized as follows:

1. Estimate the parameters $x_{\min}$ and α of the power-law model.

2. Calculate the goodness-of-fit between the sample data and the power-law model through the Kolmogorov-Smirnov test. If the test statistic D is less than the critical value, the power-law is a plausible hypothesis for the data, otherwise it is rejected.

Let us see in the next section how we applied the Kolmogorov-Smirnov test to call graphs attributes and the outcome of the test.

Analysis of the Distribution of Units' Graph Attributes

The Kolmogorov-Smirnov (K-S) test [10] is used to decide whether a sample comes from a population with a specific distribution and is based on the empirical cumulative distribution function (ECDF). It does not depend on the underlying cumulative distribution function being tested and it simply measures the maximum distance between the ECDF of the data and the fitted model:

$$\mathrm{D} = \max_{x \geq x_{\min}} |S(x) - P(x)|. \tag{4.18}$$

Here $S(x)$ is the ECDF of the data for the observations with value at least $x_{\min}$, and $P(x)$ is the cumulative distribution function (CDF) for the power-law model that best fits the data in the region $x \geq x_{\min}$. Our K-S test is summarized in Tab. 4.2 and the criterion is: Accept H_0 (the null hypothesis) if $D \leq C$. Let us see how we estimate the parameters $x_{\min}$ and α of the power-law model. Clauset et al. [10] mention several methods for the estimation of $x_{\min}$ and the simplest, and most common one, is based on a visual approach: by looking at the ECDF on a log-log plot identify the point beyond which the distribution becomes roughly straight. In our case, since we deal with six different random variables whose distributions have quite diverse variances, we need to identify a proper lower bound value for each one of them. After a visual inspection of the several ECDFs, we realized that for all six attributes we could have applied a simple criterion: cut the initial part of the distribution as follows: let $X = \{x_1, x_2, \ldots, x_n\}$ be the set of observations of

H_0	The data follow a power-law distribution
H_a	The data do not follow a power-law distribution
D	$D = \max_{x \geq x_{\min}} \lvert S(x) - P(x) \rvert$
α [7]	$\alpha = 0.05$
n	Number of samples in the region $x \geq x_{\min}$
C [8]	For $n > 35$ and $\alpha = 0.05$ $C = \frac{1.358}{\sqrt{n}}$

Table 4.2 – *KS-test definition for the units' attributes in a call graph*

a given attribute then $x_{\min} = 0.02 \cdot \max(X)$. Finally, the coefficient α for a given attribute is computed as [58]:

$$\alpha = 1 + n \left[\sum_{i=1}^{n} \ln \frac{x_i}{x_{\min}} \right]^{-1}$$

We have now all needed elements to compute the power-law model cumulative distribution whose formula is:

$$P(x \leq X) = 1 - \frac{x}{x_{\min}}^{(-\alpha+1)} \quad (4.19)$$

and run the K-S test.

We downloaded 50 open source projects from GitHub, they have been selected without any special requirement other than the followings: written in C language and created after January, 1st 2010. The empirical data of the six unit attributes have been gathered through our tool CRYSTAL and the results are shown in Tab. 4.3. The results show that the six attributes' distributions can be approxi-

Attribute	Acceptance [%]
$\deg^+$	100.00%
$\deg^-$	100.00%
RO	68.29%
RI	68.29%
SO	43.90%
SI	60.98%

Table 4.3 – *Results of the K-S test over 50 open source projects*

mated by a power-law. SO and SI reach the lowest acceptance rate, nevertheless looking at the shape of their empirical density and cumulative distribution one can easily recognize the peculiar characteristics of a power-law: on the left side of the density histogram we have few values that dominates the others and on the right side a long tail. This is exemplified also in figures 4.8 and 4.9 showing the

7. Significance level. The significance level α defines the maximum probability of rejecting the null hypothesis H_0 when it is true.
8. Critical value. Obtained from tables available in literature.

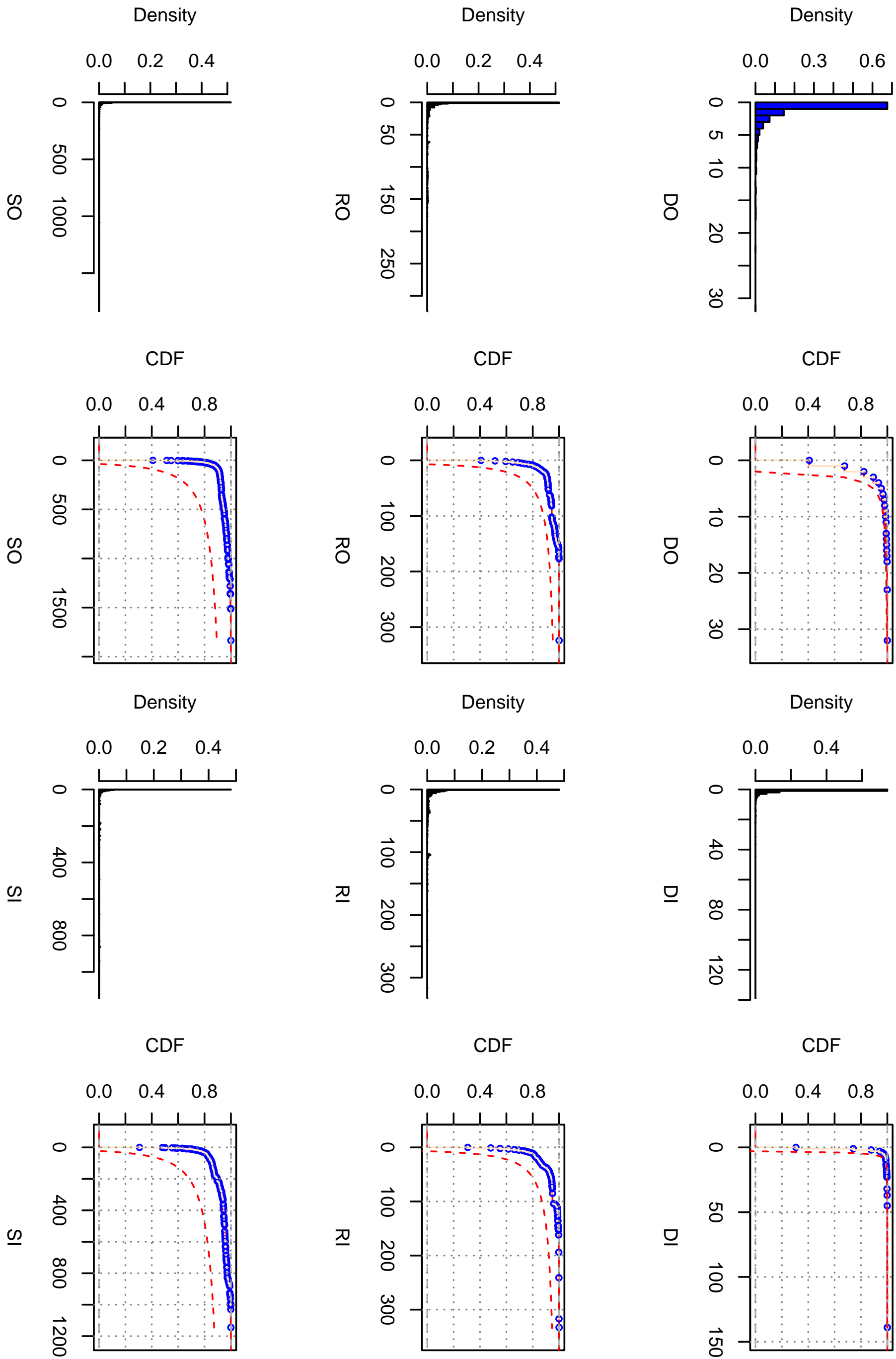

Figure 4.8 – *Plots of the six attributes of* GEDIT: deg$^+$ *(DO),* deg$^-$ *(DI),* RO, RI, SO *and* SI*. For each attribute we have two plots: the empirical density and a plot with both ECDF and the power-law CDF model fit (dashed, red line).*

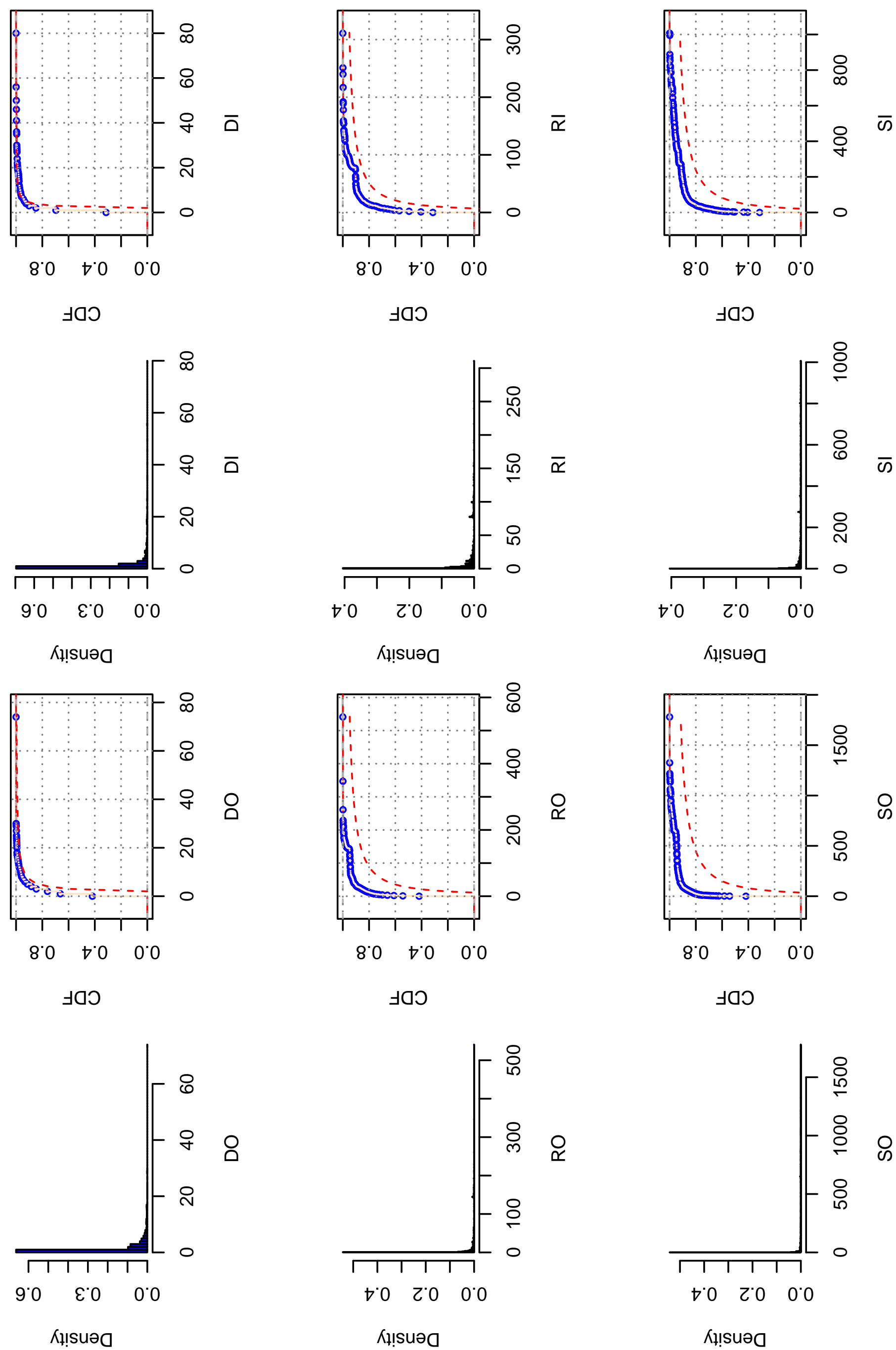

Figure 4.9 – *Plots of the six attributes of* MOSAIC: $\deg^+$, $\deg^-$, RO, RI, SO *and* SI. *For each attribute we have two plots: the empirical density and a plot with both ECDF and the power-law CDF model fit (dashed, red line).*

experimental results respectively for GEDIT and MOSAIC. In both projects SO and SI null hypothesis have been rejected nevertheless their ECDFs still look a good approximation of a power-law.

We believe that what we found is interesting and to some extent novel, at least in the context of program understanding. A similar work has been done by Myers [57], he examined six open-source software systems and he found them to possess a *scale-free*[9] degree distribution. To verify the hypothesis he plotted the log-log plot of the distributions and checked that they were straight lines. Our work extends the one done by Myers under several respects: we (1) examined a larger number of projects, (2) applied K-S test, which is an objective test compared to the subjective one used by Myers (observe a straight line on a log-log plot) and, (3) proved that also the distributions of the attributes RO, RI, SO and SI of a vertex can be approximated through a power-law.

Going back to our initial questions about the distribution of the input data to PCA, we can now draw the conclusion that our data does not follow a Gaussian distribution and, instead of the standardization of data (subtracting the mean and dividing by the standard deviation), which will squash the distribution too much, hiding the massive difference between a value in the long tail and one near the peak, we have to adopt normalization which simply rescales variables in the range $[0, 1]$ preserving the original distribution.

Hence, we apply PCA to the normalized matrix:

$$\mathbf{N} = \text{norm}(\mathbf{\Phi}) = \begin{pmatrix} n_{1,1} & n_{1,2} & \cdots & n_{1,n} \\ n_{2,1} & n_{2,2} & \cdots & n_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ n_{m,1} & n_{m,2} & \cdots & n_{m,n} \end{pmatrix}$$

(see Eq. 4.17) whose generic element i, j is:

$$n_{i,j} = \frac{\phi_{i,j} - \min(\mathbf{\Phi}_i)}{\max(\mathbf{\Phi}_i) - \min(\mathbf{\Phi}_i)}$$

where $\mathbf{\Phi}_i$ is the i-th row vector of $\mathbf{\Phi}$.

Applying PCA to Entry-point Analysis

PCA, for the same reasons mentioned above (variables' scaling), is sensitive to outliers and then they have to be removed. We are not talking about the distribution of single attributes anymore but here we refer to the set of observations represented as 6-dimensions data points. The quantity we consider for outliers

9. Scale-free is a graph whose in-degree and out-degree distributions are power-laws.

removal is the norm of vectors of units' attributes. By looking at the empirical densities of the norm of vectors we observed an exponential trend reminding the characteristic shape of the power-law (some examples are shown in Fig. 4.10). This has been proved by Wilke et al. [83] who stated that the summation of several power-law random variables equals to a power-law variable again. Though the norm of a vector is not a simple summation of elements, for our own purposes we can consider it as having a power-law distribution. Power-law is clearly not symmetric around the mean value and, since we do not want to discard elements belonging to the peak, the criterion to define outliers is: $\|\mathbf{v}\| - \mu \geq c \cdot \sigma$ where μ and σ are respectively sample mean and standard deviation of the vectors' norm of a given cluster, the factor c is usually set either to 2 or 3. We prefer to privilege the recall (for the definition of recall and an explanation of the reasons behind its choice here see Sec.6.3) of FEAT so we set $c = 3$. This increases the chances of not missing relevant items though it leads to a lower precision.

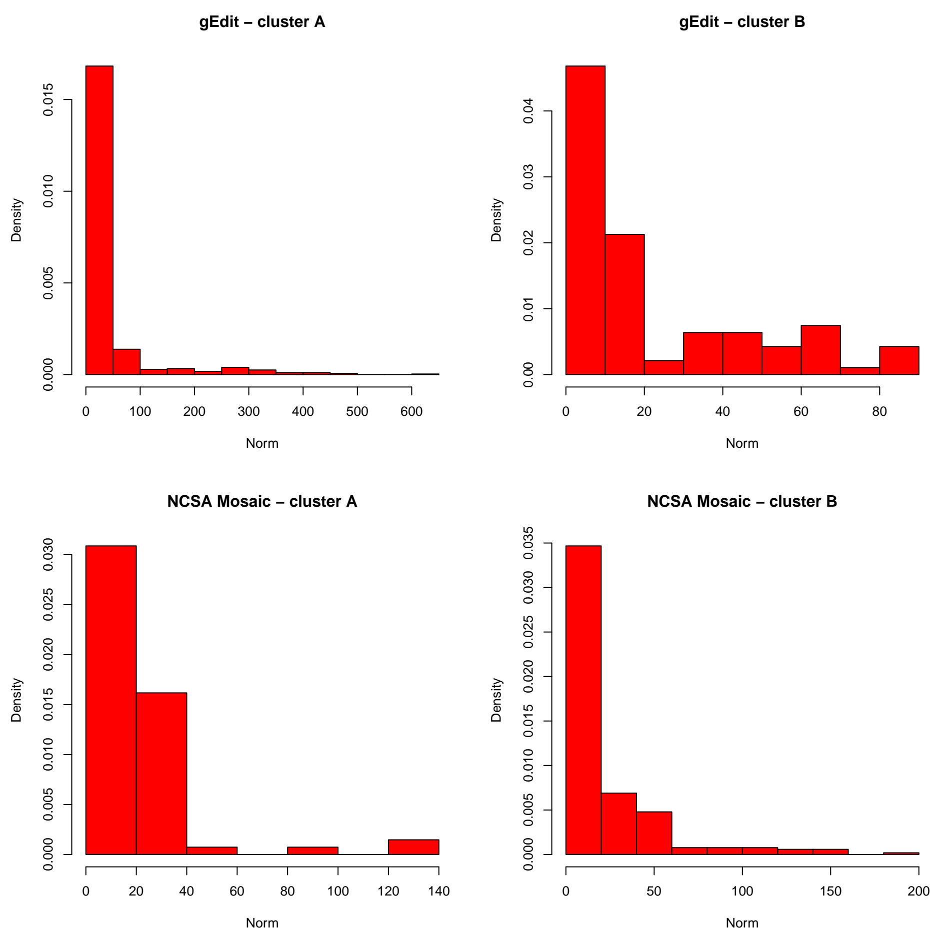

Figure 4.10 – *Histograms of the norm of attributes' vectors of two projects (*GEDIT *and* MOSAIC*, two clusters each).*

Before moving on, let us sum up what we have seen of FEAT so far:

- A whole software system is split into clusters which are highly cohesive sets of units of code.
- Clusters' elements (units) are represented as vectors of six attributes extracted from the call graph.

The next step is the identification of entry-points. We need a way to compare the units in $\mathbf{\Phi}$ to a reference in order to select entry-points. The solution we devised is to create an artificial data point (a query vector) that can represent the *ideal* entry-point in a specific cluster. The distance measured between the query vector and units (vectors in $\mathbf{\Phi}$) is the basis to create a ranking of units where the first positions are occupied by entry-points; in other words: the lower the distance the higher the ranking.

The query vector $\mathbf{q}$ follows the same structure of every other unit's vector (see Eq. 4.16) but the values of its components have to be chosen following our basic assumption for what an entry point is: a node having higher *out*-values and lower *in*-values. Hence, given a cluster $\mathcal{C} = \{u_1, u_2, \ldots, u_n\}$ the artificial query vector representing an ideal entry-point is defined as:

$$\mathbf{q} = \begin{pmatrix} \max\limits_{\forall u \in \mathcal{C}}\{\deg^+(u)\} \\ \min\limits_{\forall u \in \mathcal{C}}\{\deg^-(u)\} \\ \max\limits_{\forall u \in \mathcal{C}}\{\mathrm{RO}(u)\} \\ \min\limits_{\forall u \in \mathcal{C}}\{\mathrm{RI}(u)\} \\ \max\limits_{\forall u \in \mathcal{C}}\{\mathrm{SO}(u)\} \\ \min\limits_{\forall u \in \mathcal{C}}\{\mathrm{SI}(u)\} \end{pmatrix} \tag{4.20}$$

The application of PCA to entry-point analysis requires first the computation of the SVD factorization of the matrix $\mathbf{N} = \mathbf{U\Sigma V}^T$. In our context, $\mathbf{U}$ describes the relationship between attributes (rows) and features [10] (columns). $\mathbf{\Sigma}$ is a diagonal matrix whose elements $(\sigma_1, \sigma_2, \cdots, \sigma_n)$ are sorted in decreasing order of values and they represent the relative strength of every feature with respect to the overall space. Finally, we have $\mathbf{V}^T$ describing the relationship between features (rows) and code units (columns).

Let be $r = \mathrm{rank}(\mathbf{N})$, we can *shrink* the original space to an *adequate* number of dimensions $k < r$ while keeping a suitable amount of variance defined as a factor

10. These features must not be confused with the concept of feature used in program understanding in branches like feature location or feature extraction. We refer here to the broader sense adopted in PCA where a feature is a scaled, rotated linear combination of several input attributes. Features are the elements of the orthonormal basis obtained through SVD. See Sec.3.2 for more details.

$K \in (0,1)$. Let's call *entry-points cumulative variance* the following function:

$$\mathrm{ecv}(k) = \frac{\sum_{i=1}^{k} \sigma_i}{\mathrm{tr}(\boldsymbol{\Sigma})}$$

where $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n$. Then, the solution of the constraint:

$$\begin{aligned} \underset{k}{\text{minimize}} \quad & \mathrm{ecv}(k) \\ \text{s.t.} \quad & \mathrm{ecv}(k) \geq K, \\ & k < r. \end{aligned} \tag{4.21}$$

gives us the factor k, allowing us to define the *entry-points feature space* as follows: $\mathbf{N} \approx \mathbf{N}_k = \mathbf{U}_k \boldsymbol{\Sigma}_k \mathbf{V}_k^T$.

Finally, we can create the query vector (see Eq. 4.20), project it into the entry-points feature space and compute its distance toward every cluster's unit. The projection of the query vector is computed as follows:

$$\mathbf{q}_k = \mathbf{q}^T \mathbf{U}_k \boldsymbol{\Sigma}_k^{-1}$$

while the projections of unit code vectors are the column vectors:

$$\mathbf{V}_k^T = [\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n]$$

The Euclidean distance $\mathrm{d}(\mathbf{q}_k, \mathbf{v}) = \sqrt{\sum_{i=1}^{k}(q_i - v_i)^2}$ between the query vector $\mathbf{q}_k \in \mathbb{R}^{\mathrm{k}}$ and $\mathbf{v} \in \mathbb{R}^{\mathrm{k}}$ is the key element for ranking units defined as follows:

Definition 4.3. (Entry-Point Ranking) Let $\mathcal{C}$ be a cluster of n units and $\mathbf{q}_k$ its feature space query vector, the ranking over units is defined as:

$$\mathcal{K}_{\mathcal{C}} = \{e_1, e_2, \cdots, e_n \mid \mathrm{d}(\mathbf{q}_k, \mathbf{v}_{e_1}) \leq \mathrm{d}(\mathbf{q}_k, \mathbf{v}_{e_2}) \leq \cdots \leq \mathrm{d}(\mathbf{q}_k, \mathbf{v}_{e_n})\}$$

Example

Let's consider the induced subgraph associated with a cluster $\mathcal{C}$ extracted from an actual experiment and showed in Fig. 4.11. The all-shortest-paths matrix of the graph is:

$$S_{7,7} = \begin{array}{c} \begin{matrix} u_1 & u_2 & u_3 & u_4 & u_5 & u_6 & u_7 \end{matrix} \\ \begin{pmatrix} 0 & \infty & 1 & 1 & \infty & \infty & 2 \\ 1 & 0 & 2 & 2 & \infty & \infty & 3 \\ \infty & \infty & 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 0 & \infty & \infty & 1 \\ \infty & \infty & \infty & 1 & 0 & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty & 0 & 1 \\ \infty & \infty & \infty & \infty & \infty & \infty & 0 \end{pmatrix} \begin{matrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \end{matrix} \end{array}$$

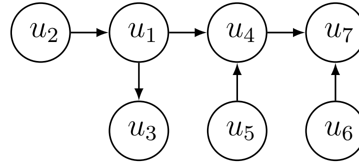

Figure 4.11 – *Graph associated to the cluster of the example on PCA applied to entry-point selection*

and the matrix of attributes, obtained through Eq(s). 4.10 to 4.15 is:

$$\mathbf{\Phi} = \begin{pmatrix} 2 & 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 2 & 0 & 0 & 2 \\ 3 & 4 & 0 & 1 & 2 & 1 & 0 \\ 1 & 0 & 2 & 3 & 0 & 0 & 5 \\ 4 & 8 & 0 & 1 & 3 & 1 & 0 \\ 1 & 0 & 3 & 4 & 0 & 0 & 9 \end{pmatrix} \begin{matrix} \text{deg}^+ \\ \text{deg}^- \\ \text{RO} \\ \text{RI} \\ \text{SO} \\ \text{SI} \end{matrix}$$

(columns: u_1, u_2, u_3, u_4, u_5, u_6, u_7)

By setting K to 0.85 and solving Eq. 4.21 we obtain $k = 2$, which means that the fulfillment of the constraint requires us to keep the first two PCA components. Hence, though reducing a six-dimensional space to a two-dimensional one, we still cover at least 85% of the entire variance contained into the original space. Below we have the $\mathbf{V}_k^T$ matrix whose columns are the projections of the units in $\mathcal{C}$ into the feature space.

$$\mathbf{V}_2^T = \begin{pmatrix} 0.29 & 0.29 & 0.27 & 0.43 & 0.12 & 0.05 & 0.74 \\ -0.38 & -0.74 & 0.14 & 0.07 & -0.31 & -0.13 & 0.41 \end{pmatrix}$$

Then, we create the artificial query vector (see Eq. 4.20) $\mathbf{q} = [2, 0, 4, 0, 8, 0]$ and project it into the feature space: $\mathbf{q}_k = [0.28, -0.72]$. Finally, by measuring the distance between units' vector and the query vector we obtain the following ranking (see Def. 4.3) $\mathcal{K}_G = \{2, 1, 5, 6, 4, 3, 7\}$. It is interesting to note the first and last element of the list. Looking at the graph in Fig. 4.11 we see that unit 2 corresponds exactly to our definition of entry point and indeed obtains the highest score while unit 7, which is just a dead end in the graph, reaches the lowest one. The graph in Fig. 4.12 shows the 2-dimensional feature space presenting also a geometric point of view of the cluster. Here, again, we can see how close to unit 2 (its vector) is the query vector (denoted by "q") while unit 7 lays far away from it.

4.8.2 Geometric Aspects of Entry-points

Fig. 4.13 shows some entry-points taken from a program topos of GEDIT. We can see two groups of units represented by the two blue dots. The interesting thing

Table 4.4 – *Ranking of the units in the example*

Vertex	Distance
2	0.026
1	0.342
5	0.441
6	0.637
4	0.808
3	0.864
7	1.216

	Unit Name	$\mathrm{d}(\mathbf{q}_k, \mathbf{v}_i)$
1	`_gedit_cmd_search_find_{`**`prev,next`**`}`	0.209
2	`_gedit_cmd_edit_{`**`delete,copy,paste, select_all,cut,redo,undo`**`}`	0.227

Table 4.5 – *Extract of the rankings of a program topos with distances between entry points and query vector (the suffixes to complete unit names are between curly braces)*

is that the units in every group belong to the same functional area: in one group we have *clipboard management* entry-points while the other contains those related with *find* capability. This means that, from the geometric standpoint, semantically similar entry-points tend to position close to each other (see Tab. 4.5 for the results). We believe that the reciprocal position of units into the feature space, in addition to their distance with the query vector, can be profitably used for the discovery of system's capabilities.

4.9 Program Topoi

Up to this point we described all basic elements needed to rank the units in a cluster. The ranking itself is not sufficient to extract entry-points, we need a criterion to determine which units are to be considered entry-points and which are not. Recall that the ranking of units in a cluster depends on the Euclidean distance between them and the query vector, then we can classify as entry-points all units whose position is *sufficiently* close to the query vector. More precisely a program topos is defined as follows:

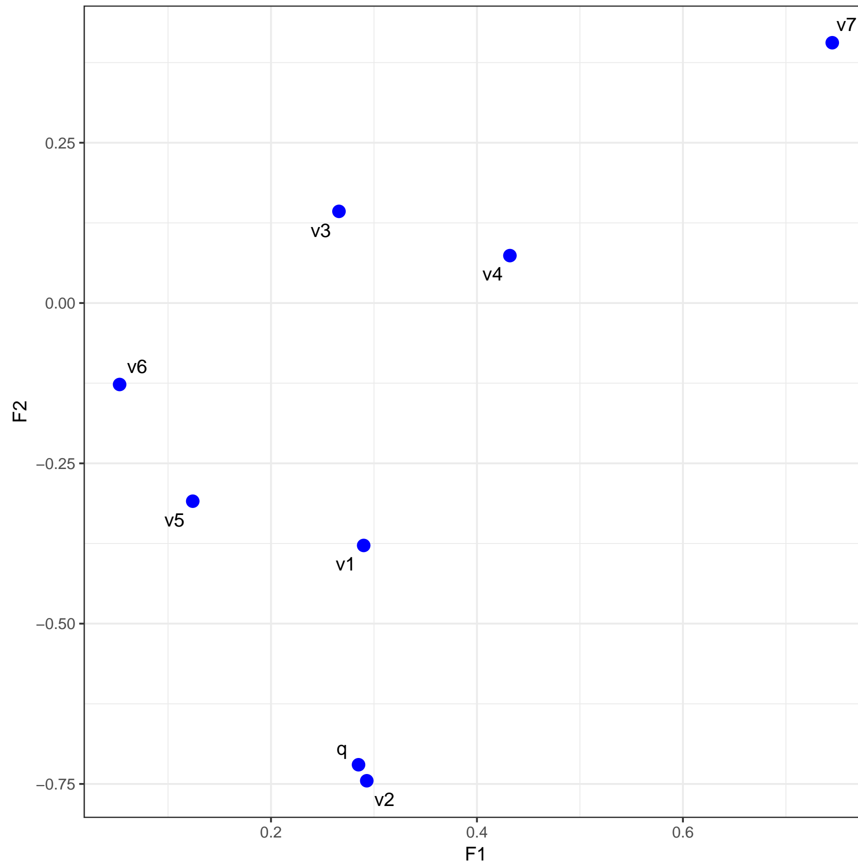

Figure 4.12 – *2D representation of the example. The vector labeled as "q" is the query vector.*

Definition 4.4. (Program Topos) Let $\Delta_{\mathcal{C}}$ be the set of distances between the units in a cluster $\mathcal{C}$ and the query vector $\mathbf{q}_k$, $\mathcal{K}_{\mathcal{C}}$ the entry point ranking and, β a parameter ranging in $(0, 1)$ then the program topos of $\mathcal{C}$ is the sub-list $\Theta_{\mathcal{C}} \subset \mathcal{K}_{\mathcal{C}}$ such that the distance between any unit of $\Theta_{\mathcal{C}}$ and $\mathbf{q}_k$ is not greater than a threshold value d_β which is the β-order percentile[a] of $\Delta_{\mathcal{C}}$.

a. The τ-order percentile of a distribution X is the value (x_τ) below which the fraction τ of elements in the ranking falls. Defined as: $P(X \leq x_\tau) = \tau$.

The units in a program topos $\Theta_{\mathcal{C}}$ are the entry points of $\mathcal{C}$ (Fig. 4.1 box 6).

4.9.1 Entry-point Dictionary

In order to enrich the description of an entry-point we equip each unit of a program topos with a textual content. This text, extracted in the first phase of the process, is the union of words coming both from the entry-point itself and from

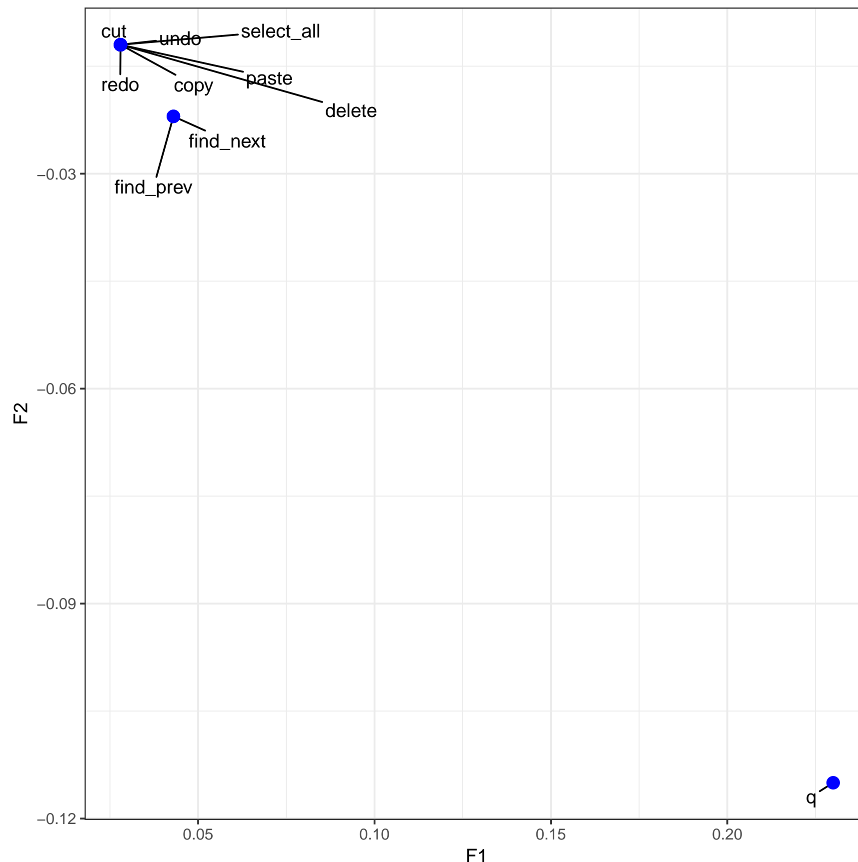

Figure 4.13 – *2D representation of some entry-points (identified by their unit names) extracted from* GEDIT*. The vector labeled as "q" is the query vector.*

its neighborhood [11]. By merging words from the entry-point and its neighbors we create a richer index of words increasing the chances of retrieving entry-points through free text queries. Table 4.6 shows some dictionaries of entry-points extracted from GEDIT.

Conclusion

This chapter presented FEAT in its entirety describing the three steps of the process: preprocessing, clustering and, entry-point selection in details. The approach evolved and became more complex taking inspiration from the experimental evidences gathered along the way to the current version.

The chapter explained how FEAT tries to automate some common practices adopted in program comprehension (Sec.2.2) like the usage of both semantic and struc-

11. The neighborhood of a vertex v in a graph G is the induced subgraph of G consisting of all vertices adjacent to v.

Unit Name	**Dictionary**
`_gedit_cmd_edit_copy`	edit, call, function, gedit, log, trace, window, clipboard, file, line, number, return, transfer, section view, cmd, debug, copy, enable, active, information
`_gedit_cmd_search_replace`	dialog, function, gedit, log, time, cmd, file, line, number, position, debug, present, search, section, inform, create, restore, replace, trace, enable, data, call
`_gedit_cmd_search_find_next`	enable, find, gedit, log, trace, file, line, number, run, cmd, inform, debug, forward, section, function, search, call
`_gedit_cmd_search_find_prev`	enable, find, gedit, log, trace, cmd, file, line, number, previous, inform, debug, function, run, section, call, search, backward

Table 4.6 – *Example of entry-points' dictionaries enriched with their neighborhood*

tural information to isolate concepts under the form of clusters. To better suit the program understanding context we deeply customized HAC's algorithm providing both a novel notion of distance (Sec.4.5) and a cutting criterion (Sec.4.7). Applying PCA to code units involved a thorough study of call graphs from the statistical standpoint (Sec.4.8.1) revealing interesting patterns. Finally, we reached the point of providing a formal definition of program topoi (Def.4.4). It is also interesting to note how entry-points showed some geometric properties that could lead to future development.

V

FEAT **Tooling Support**

This chapter presents an overview of both the tool employed in the experimental evaluation of FEAT and the platform on which the tool is based.

5.1 Crystal Platform

CRYSTAL is a software platform that we designed in the context of the Certus SFI project hosted at Simula Research Laboratory. It is based on the micro-services architectural style, and it encourages the adoption of a design strategy enhancing both self-documenting systems and share and reuse of software components. The core of the platform is an OSGi (Open Services Gateway initiative) container which hosts a set of services implementing basic functionalities.

The design of high level functionality, those directly used by users or external systems, is realized through a model, a BPMN 2.0 (Business Process Modeling and Notation) executable model which decouples the specific aspects related to the implementation of an application from the primitive, reusable services. The model, following the syntax of the BPMN visual language, provide a clear and formal specification of an application use cases implementation.

5.1.1 OSGi

The OSGi technology is a set of specifications that define a dynamic component system for `Java`. These specifications enable a development model where ap-

plications are (dynamically) composed of many different (reusable) components. The OSGi specifications enable components to hide their implementations from other components while communicating through services, which are objects that are specifically shared between components. This surprisingly simple model has far reaching effects for almost any aspect of the software development process. The OSGi component system is actually used to build highly complex applications like IDEs (Eclipse), application servers (GlassFish, IBM Websphere, Oracle/BEA Weblogic, Jonas, JBoss), application frameworks (Spring, Guice), industrial automation, residential gateways, phones, and so much more. [1]

OSGi model makes an application to emerge from putting together different reusable components that had no a-priori knowledge of each other. Applications are built by dynamically assembling a set of components. For example, you have a home server that is capable of managing your lights and appliances. A component could allow you to turn on and off the light over a web page. Another component could allow you to control the appliances via a mobile text message. The goal was to allow these other functions to be added without requiring that the developers had intricate knowledge of each other and let these components be added independently.

5.1.2 Business Process Modeling and Notation

The Business Process Modeling Notation (BPMN) is a graphical notation that depicts the steps in a business process. BPMN depicts the end to end flow of a business process. The notation has been specifically designed to coordinate the sequence of processes and the messages that flow between different process participants in a related set of activities.

BPMN is targeted at a high level for business users and at a lower level for process implementers. The business users should be able to easily read and understand a BPMN business process diagram. The process implementer should be able to integrate a business process diagram with further detail in order to represent the process in a physical implementation. BPMN is targeted at users, vendors and service providers that need to communicate business processes in a standard manner [2].

CRYSTAL makes an innovative usage of BPMN:

- Decoupling the logic, governing a use case, from its actual implementation.
- Easing the adoption of good design practices. Every component is a small

1. source www.osgi.org
2. source www.bpmn.org

service with a clear identity in terms of functionality.

- The main flow of an application is defined through a visual yet executable diagram.
- Improve collaboration among different stakeholders: testers, engineers, business users, researchers, etc.
- A BPMN diagram itself is a way of documenting a software application.

CRYSTAL is a versatile platform not targeted to any specific application and its underlying model may be exploited in many different contexts. Let us now see CRYSTAL.FEAT the tool, based on CRYSTAL, used for the whole experimental evaluation presented in this thesis.

5.2 Crystal.FEAT

5.2.1 Architecture

The overall system architecture of the platform is illustrated in Fig. 5.1, there we can see the three main blocks building the whole tool: (1) OSGi container, (2) BPMN engine and, (3) RDBMS. The whole technological stack is open source and more precisely the OSGi container is Apache Karaf v4.1.4 [3], the BPMN engine is Activiti v5.22 [4] and the RDBMS (Relational Data Base Management System) is MySQL v5.7.22 Community Edition [5]. The current version of Crystal.FEAT amounts to 45 KLOC.

CRYSTAL is based on the micro-services architectural pattern hence there is not a monolithic executable containing all features but a set of small binary components deployed into the OSGi container. The overall business logic implementing FEAT process is not embedded into any of those components but provided as a BPMN diagram. Put simply, the execution of a FEAT process does not start from a usual `main()` procedure but from loading the BPMN model, providing it to the BPMN engine and, making it run.

3. http://karaf.apache.org/
4. https://www.activiti.org/
5. https://www.mysql.com/

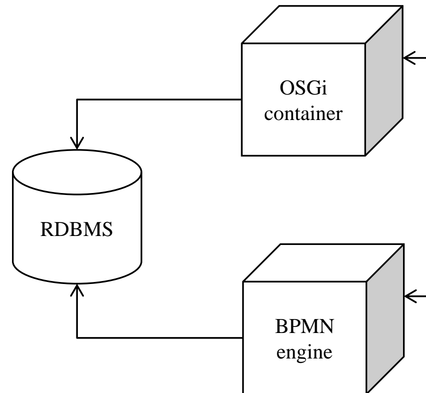

Figure 5.1 – *Main components of* CRYSTAL*'s system architecture*

5.2.2 FEAT **Process**

The BPMN workflow of FEAT is shown in Fig. 5.2. Just very few words about the BPMN notation; every box represents a task and circles are events like: start (light circle), stop (bold circle), error raise (bold circle with thunder) and, error catch (dashed circle with thunder). We will skip many of the details about the syntax of the BPMN language, nevertheless the diagram is self-explicable and its basic understanding does not require any prior background in workflow automation. Let us examine the main tasks involved in the process.

Preprocessing

The input to CRYSTAL.FEAT is a compressed (zip) file containing a software project. Even though the theoretical approach does not depend on a specific language, the current version of the tool can handle only `C` programs. The first step is then specific for handling `C` programs by preprocessing source code to expand macro definitions.

Creation of FEAT **Model**

According to Def.4.1, in the task "Create FEAT model" the process parses the `C` code generating the call graph and the set of unit-documents. Parsing is accomplished by means of a parser whose code is automatically generated from the grammar definition of the `C` language. The parser generator is ANTLR[6] which comes with many available grammars. In our case we needed to add a couple of *lexer*[7] rules to the standard grammar in order to properly handle comments in

6. www.antlr.org
7. The lexer scans the text and transforms individual characters into tokens.

source code since usually they are just thrown away from parsers. In a second phase the task resolves function calls and it creates the call graph. Beside the call graph, a "All-Pairs Shortest Path" matrix is created according to the Floyd-Warshall algorithm. It contains the shortest graph distance between each pair of vertices in the call graph.

Batch of Experiments

CRYSTAL.FEAT can generate batches of experiments (task "Create Experiment Batch") on the basis of an external configuration file containing the various options (i.e. values for α, β, cutting criterion, etc.) and for each combination of the options' values it creates an experiment instance. Each generated experiment instance is the input for the successive, iterative block (identified by the big box called "Experiment loop" in Fig. 5.2 containing several tasks).

Unit-documents Creation

The tasks "Function Documents Extraction" and "VSM Data Creation" take respectively care of generating text files, corresponding to each unit found in the source code, and pre-process them for tokenization, stop word removal, etc. (see Sec. 4.3 for more details). Finally, a LSA factorization is created and ready to be used to compute the semantic distance between units. The VSM model is persisted on disk through the *Weka* v3.8.0 framework [8] and SVD factorization is done through *oj! Algorithms* v44.0 library [9].

Clustering

FEAT model is now ready to be provided to the HAC algorithm which will use it to compute both hybrid distances (see Eq. 4.6) and cutting criterion (see Eq. 4.9). The HAC algorithm, more precisely is a priority queue HAC (see Alg. 1 for more details) has been derived from Weka's implementation and underwent through an extensive refactoring. Main changes regarded the possibility of dynamically provide a distance measure and cutting criterion whereas previously only the Euclidean distance was allowed and no cutting criterion was present.

8. https://www.cs.waikato.ac.nz/~ml/weka/
9. http://ojalgo.org/

Entry-point Analysis

This task computes on a cluster by cluster basis the similarity between units and the query vector (see Sec. 4.8.1 for an example) producing a program topoi. A copy of the all shortest path matrix is serialized on disk in order to provide the neighborhood of entry-points. Also a dictionary of words of every entry-point is stored into the database.

Report Generation

Finally, all information about process execution are stored and made available for further analysis.

5.2.3 User Interface

CRYSTAL.FEAT allows three ways of interaction: web user interface, command line and, RESTful web services. The first one is mainly intended for uploading single projects (see the button "New Project" in Fig. 5.3) and for searching program topoi (see Fig. 5.4). Searching program topoi requires the input of a free-text query text. In the example shown in Fig. 5.4 all program topoi related with the word "clipboard" are retrieved. In this version CRYSTAL.FEAT exploits the *full-text search* capabilities of MySQL [10]. The screen shot shows the entry-points found with the query; by clicking on an entry-point the system will show the graph of the entry-point neighborhood (Fig. 5.5). A running example of CRYSTAL.FEAT is available on the web [11] and contains all 610 projects used in the experimental evaluation (see Chap.6).

The second way of interacting with CRYSTAL.FEAT is by means of the Apache Karaf console. It is a command line interface which can be accessed either locally and remotely through SSH. Table 5.1 lists the more relevant implemented commands.

Finally, CRYSTAL.FEAT exposes few capabilities as RESTful Web services. REST stands for REpresentational State Transfer and is used to build Web services that are lightweight, maintainable, and scalable in nature. The underlying protocol is HTTP and REST defines an architectural style allowing a uniform access to stateless operations. RESTful web services are expected to follow the semantic of

10. For more details about the syntax of full-text search expressions in MySQL follow the link https://dev.mysql.com/doc/refman/8.0/en/fulltext-search.html
11. CRYSTAL.FEAT is available on line at the link http://ec2-18-185-125-11.eu-central-1.compute.amazonaws.com:8181/crystal/feat/

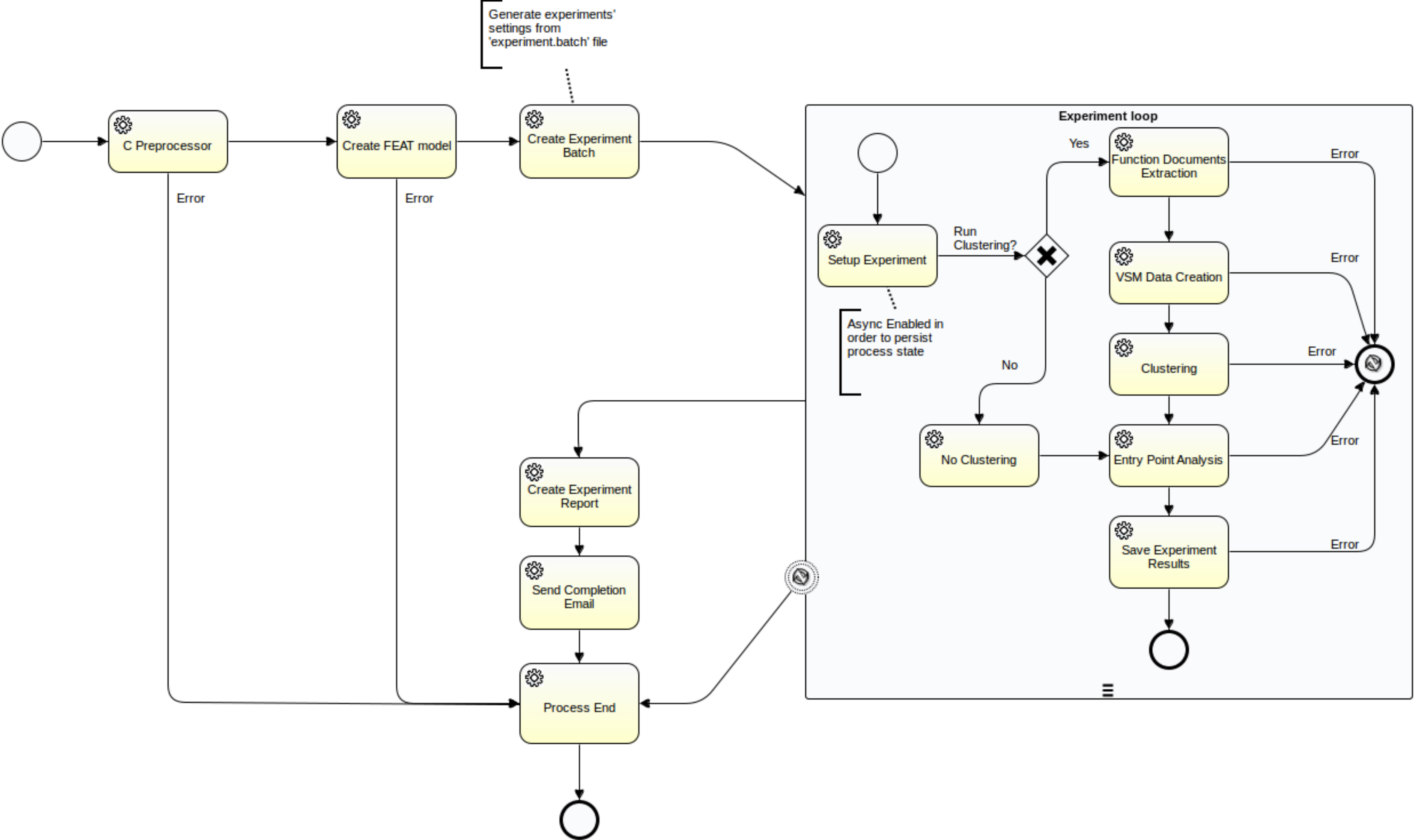

Figure 5.2 – *BPMN executable diagram for the main process of* FEAT*. Boxes represents tasks and circles are events like: start, stop, raise error, catch error, etc.*

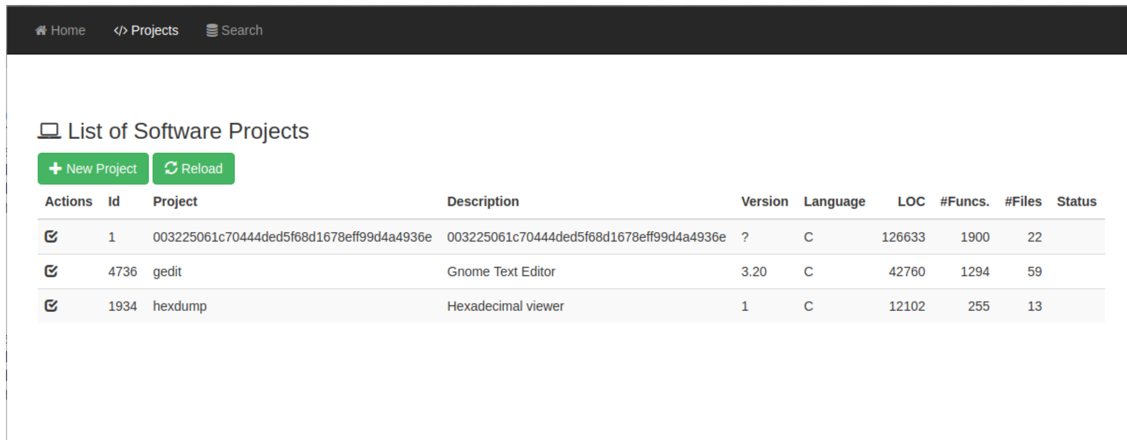

Figure 5.3 – *Screenshot of the web user interface of* FEAT*. List of currently loaded projects.*

the HTTP verbs GET, POST, PUT and DELETE.

CRYSTAL.FEAT RESTful API, accessed via HTTP, allows to manage projects, run FEAT, run queries, etc., data is exchanged as JSON text and all API endpoints are rooted at http://<web-server-url>:8101/cxf/crystal. For a list of the more relevant APIs take a look at Tab. 5.2.

Conclusion

This chapter presented the tool implementation for FEAT. FEAT is heavily based on the micro-services architectural pattern deployed on an OSGi container. To further stress its distributed nature and enhance reuse, all main functionalities are created through BPMN executable diagrams. There three ways of accessing FEAT namely through: a web user interface, a set of RESTful API and, a command-line interface.

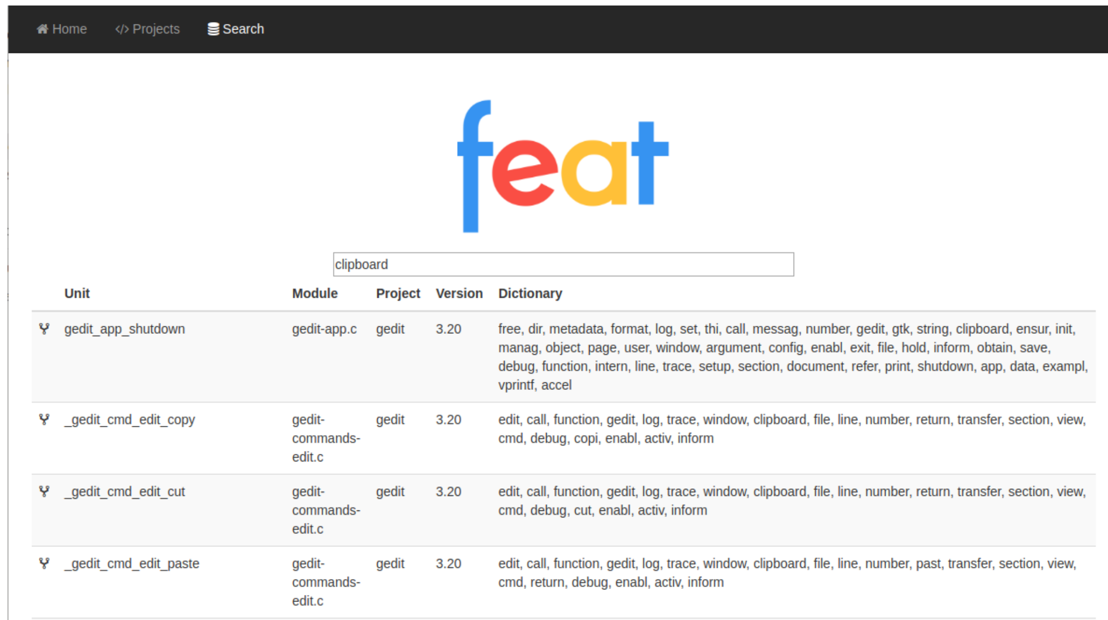

Figure 5.4 – *Screen shot of the web user interface of* FEAT. *Program topoi search.*

Figure 5.5 – *By clicking on an entry-point, found through the search capability, the system will show its neighborhood*

Command	Description
`crystal:feat`	**Run full** FEAT **process**
`crystal:bpe`	Run the Best Partition Experiment (see Sec. 6.2)
`crystal:cgstat`	Create call graph statistics (see Sec. 4.8.1)
`crystal:rndep`	Run random experiment (see Sec. 6.3.1)
`crystal:tar-get-repo`	Extract projects in tar.gz files from a local folder and creates projects
`crystal:gh-mine`	Run a GitHub miner looking for repositories matching some criteria like: language, size, etc. provided through a property file. For every dowloaded GitHub repository a project is created.
`crystal:gh-get-repo`	Download a GitHub repository and creates a project
`crystal:get-zip`	Download the *zipball* file(s) of projects created out of GitHub repositories
`crystal:list-projects`	List of the currently loaded projects

Table 5.1 – CRYSTAL.FEAT *commands' list*

Endpoint	Description
`/feat/manager/project/list`	Get the list of loaded projects
`/feat/manager/query/units`	Full text unit query
`/feat/manager/query/topos /unit/neighborhood`	Graph neighborhood of a unit
`/feat/manager/query/modules`	Full text module query
`/feat/manager/query/topos`	Full text topos query
`/feat/features/project /cluster`	Run FEAT

Table 5.2 – CRYSTAL.FEAT *RESTful API*

VI

Experimental Evaluation

This chapter presents the experiments that we designed and executed for the evaluation of the most relevant aspects of FEAT. The experiments have been executed on a software testing platform called CRYSTAL (more details in Chap. 5) hosted on an Intel dual core i7-4510U CPU with 8GB RAM. In addition, for the more demanding experiments described in Sec. 6.5.1 we used Amazon Web Services (AWS) which is a platform providing virtual servers. The server we rented has the following characteristics: 8 CPU (Intel Xeon E5-2686 2.3 GHz), 61 GB RAM and 32 GB SSD hard disk.

The experiments show comparison with some baselines, quantitative evaluation of FEAT's ability of discovering program topoi, a scalability experiment and, finally an application of FEAT to a large scale source code repository.

6.1 Experimental Subjects

The experimental subjects that we used can be divided into three groups according to the purpose of the experiments: (1) synthetic projects, (2) projects equipped with oracles and, (3) projects downloaded from public repository. We employed 610 projects coming mainly from a large public repository called Software Heritage (more on this later in Sec. 6.5.1) and have been selected without any special requirement a part from the following:

1. Written in `C` language.

2. Size of the repository [1] between 5 and 50 MBytes.

For those experiments requiring a finer-grained analysis such as assessing the accuracy of discovering capabilities, or evaluating feature location of FEAT, we used the following projects:

- HEXDUMP [2]. It is a hexadecimal viewer, it displays binary data contained in files, as a readable sequence of codes.
- GEDIT. GNU Editor v3.20 is the default text editor of the GNOME desktop environment.
- MOSAIC. NCSA Mosaic (National Center for Supercomputing Applications) is a web browser, well-known for its extended support of Internet protocols. It is used as a benchmark in research works similar to the present one (i.e. Marcus et al. and Kuhn et al. for feature location approaches [47, 37]).

Finally, for some experiments we have created synthetic projects according to the model (see Def. 4.1) used by FEAT and based on its hybrid perspective of software systems.

6.2 Goodness of FEAT in Selecting Partitions

The main objective of the cutting criterion (Eq. 4.9) is the automatic selection of a partition of clusters to be used in the entry-point analysis step. The question we want to address here is:

> **RQ1**: *How effective is the hybrid perspective of* FEAT, *based on* d_{FEAT} *and* T_{FEAT}, *about driving HAC to find optimal partitions in a controlled setting?*

In this experiment we create instances of FEAT models (see Def.4.1) characterized by the following input variables: number of units, density of the call graph and, a range of values where the length of unit-documents can span. These parameters are used to randomly generate both the call graph and the set of unit-documents. For every instance, the experiment produces all partitions of the set of units, computes modularity and coherence and, find the maximum T_{FEAT} value. Let us call this part of the experiment *brute force*. Then, we run FEAT and compare T_{FEAT} value of the chosen partition to those obtained through brute force.

1. With *repository* we identify a container including source code but also additional elements like documents, scripts, etc. hence affecting repository's size.
2. http://sourceforge.net/projects/hexdump

From combinatorics we know that the number of partitions of a set grows extremely fast. Hence, we let cardinality ($n = |\mathcal{U}|$) of sets of units $\mathcal{U}$ varies in the range $n \in \{10, 11, 12\}$. Correspondingly, the number of partitions are $B_{10} = 115,975$, $B_{11} = 678,570$ and, $B_{12} = 4,213,597$ (B_n is the Bell number of n). Call graph's random creation follows Gilbert's approach [23] and is ruled by a density value (ρ) corresponding in our context to the probability of having an edge between any pair of vertices. Density's range is $\rho \in \{0.1, 0.2, \ldots, 0.9\}$.

Starting from a fixed alphabet (11 symbols) all words of a given size (4) are generated to form a dictionary of 14,641 words. Unit-documents' lengths span at random having between 5 and 20 words. Documents' content is randomly created as well and words' distribution follow Zipf's law [74].

The generation of any instance of the model is replicated 10 times and the results, shown in Tab. 6.1, are averaged out. HAC merges units and clusters, driven by $\mathrm{d}_{\mathrm{FEAT}}$, producing partitions and, selects the partition where the maximum value for $\mathrm{T}_{\mathrm{FEAT}}$ occurs. To compare FEAT's values to brute force approach ones, we provide an indicator called T_{score}. Let us call $\mathcal{T}$ the set of $\mathrm{T}_{\mathrm{FEAT}}$ values of all partitions of $\mathcal{U}$ generated through the brute force approach, then we define the *cutting criterion score* as:

$$T_{score} = \frac{\mathrm{T}_{\mathrm{FEAT}} - \min(\mathcal{T})}{\max(\mathcal{T}) - \min(\mathcal{T})} \in [0, 1]$$

representing how *close* is the partition found from FEAT to the optimal ones.

Every line in Tab.6.1 contains the average values of 10 tests with the same input values ($n, \rho, \alpha = 0.5$).

n	ρ	T_{score}	n	ρ	T_{score}	n	ρ	T_{score}
10	0.1	0.887	11	0.1	0.880	12	0.1	0.903
	0.2	0.811		0.2	0.893		0.2	0.801
	0.3	0.759		0.3	0.822		0.3	0.848
	0.4	0.740		0.4	0.792		0.4	0.738
	0.5	0.731		0.5	0.750		0.5	0.671
	0.6	0.762		0.6	0.766		0.6	0.795
	0.7	0.780		0.7	0.817		0.7	0.778
	0.8	0.872		0.8	0.849		0.8	0.828
	0.9	0.973		0.9	0.908		0.9	0.977
		0.813			**0.831**			**0.815**

Table 6.1 – *Experimental results of the clustering step's evaluation*

There are two important restrictions that need to be taken into account when analyzing the results of this experiment. First, the number of units (between 10 and 12) is small because, as mentioned above, exploring the whole search space

made of all partitions of a set becomes rapidly infeasible when the number of elements goes above a given threshold. Second, the density of randomly generated call graphs is higher than of those extracted from real programs (see Tab.6.8); the choice for a range of values for the density is dictate by the number of nodes in the random generated graphs: with a maximum value of 12 nodes, lower values of density would lead to the creation of many disconnected graphs which would be useless for our purposes. Nevertheless the experiments show very promising performance both in terms of closeness to the optimal achievable value of the cutting criterion and time. We observe that when HAC is driven by $\mathrm{d}_{\mathrm{FEAT}}$ and $\mathrm{T}_{\mathrm{FEAT}}$ it shows an interesting performance in finding partitions with both high modularity and coherence. With a very low number of iterations, which in the worst case can be equal to $n-1$, FEAT shows an average $T_{score} > 0.8$ in all combinations that we experimented.

6.3 Program Topoi Discovery Experiments

The experiments reported in this section evaluate the ability of FEAT of retrieving relevant informations from source code. In order to provide a numeric, comparable assessment we use the following metrics which are widely adopted in machine learning classifiers' systems namely *precision*, *recall* and, *F-measure* [44]. 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) and, *(iii)* we created the *oracle* which is a text file with the list of entry-point names. This is clearly a time-consuming task requiring an experienced engineer in order to run basically a fully manual program comprehension task. Using the topoi automatically mined by FEAT and the topoi listed in the oracle, we define *true positives (tp)* the units which are correctly classified as entry points, *false positives* (fp) the units which are incorrectly classified as entry points, *true negatives (tn)* the units which are correctly not classified as entry points and finally *false negatives* (fn) the units which are incorrectly not classified as entry points. The definitions of the aforementioned metrics are as follows:

1. *Precision* is the percentage of retrieved entry-points that are relevant:

$$\mathrm{P} = \frac{tp}{tp + fp}$$

2. *Recall* is the percentage of relevant entry-points that are retrieved:

$$\mathrm{R} = \frac{tp}{tp + fn}$$

For these metrics, ranging between 0 and 1, the evaluation criterion is: the higher, the better. They assess two different, orthogonal characteristics of a system: the more the units classified as entry-points the higher the recall while for the precision happens exactly the opposite. Another measure which is commonly used in judging a machine-learning classifier is its *accuracy* that is the fraction of its classifications that are correct. Based on the same elements introduced above it is defined as: $\mathrm{A} = \frac{tp+tn}{tp+fp+fn+tn}$. Unfortunately, accuracy does not seem to be the most suitable metric for FEAT; in our context the data are really skewed; the majority of units are not relevant and just few of them are entry-points. Hence, the maximization of accuracy could be achieved by deeming *all* units as non-relevant and this is clearly useless for program comprehension purposes.

Given all these premises we believe that recall (fraction of relevant entry-points that are retrieved) is a more meaningful measure for FEAT's effectiveness than precision. Users who are looking for system capabilities always prefer to have some results and can tolerate false positives especially if they come with some additional information, like words' index and entry-point's neighborhood, helping in the evaluation. The opposite, or in other words a system with higher precision, may incur in the risk of neglecting important entry-points and this is not desirable.

A single measure that trades off precision versus recall is the *F-measure*, which is the weighted harmonic mean of precision and recall:

$$\mathrm{F}_\beta = \frac{(\beta^2+1)\mathrm{PR}}{\beta^2\mathrm{P}+\mathrm{R}}$$

When using $\beta = 1$ the formula equally weights precision and recall. Values of $\beta < 1$ emphasize precision, whereas values of $\beta > 1$ emphasize recall. Hence, since we value recall more than precision in our experiments we set $\beta = 2$ (F_2 metric) [3] [44].

Oracles for HEXDUMP and GEDIT have been manually created 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. In desktop applications, like GEDIT and HEXDUMP, finding entry-points is quite straightforward. These applications usually follow an event-driven pattern and they have a central place where all event handlers related to menus and shortcuts are declared. After having located this point, the task becomes simply the collection of all involved unit names. The list of these units constitutes the oracle. The oracle of GEDIT contains 36 entry-points while HEXDUMP's one contains 12 entry-points.

First, we present experiments whose purpose is to compare FEAT with two base-

3. Parameter β here must not be confused with the one used in the definition of program topos (see Def.4.4).

lines: random classifier and FEAT without clustering. After the baselines' experiments a comprehensive experiment on FEAT will be presented. Evaluations are made on the basis of F_2, P and, R metrics computed against the two oracles.

6.3.1 Random Baseline Comparison

When we deal with a *smart* classifier the first question that needs to be addressed is how it compares with a blind, random approach:

> **RQ2**: *How does* FEAT *compare with a classifier selecting entry-points at random?*

The random classifier simply draws a given number of units from the set of units of a software system. GEDIT and HEXDUMP are the subjects of this experiment and the number of elements for each draw can range between 5% and 10% of the total number of units. These numbers are close to the average size of clusters that we observed in several experiments with the projects. The experiment took place as follows:

1. Run 1,000 draws for both HEXDUMP and GEDIT. For each draw select at random the number of units that will be drawn. Let be n the total number of units then draw's size s is $s \in [0.05n, 0.1n]$.
2. Draw s units at random, check them with the oracle, compute the metrics (A, F_2, P, and R) for every draw, and finally average them out.

Experiment's results are shown in Tab. 6.2. Accuracy values are surprisingly high but it would be misleading if they were not evaluated considering precision and recall as well. Despite the high accuracy, these two metrics show a poor performance of the random classifier with $\mathrm{F}_2 < 0.07$ in both cases. This experiment offers us the chance to stress the point about the meaning of accuracy related to our domain. FEAT's aim is to classify some, very few units as entry-points from a larger number of units, then, using the terminology introduced before, we can say that we look for few *true positives* in a large set of irrelevant items. In other words, if the total number of units is n and $tn \approx n$, then just by rejecting all units would lead to an accuracy close to 1. The conclusion is that, in program comprehension, accuracy is not the right metric when it comes to evaluate machine learning classifier and then we decided to adopt F_2.

6.3.2 No-clustering Experiment

FEAT is a two steps process requiring first clustering and afterwards entry-point

Project	A	F_2	P	R	Units	Oracle size	Draw size
HEXDUMP	0.887	0.068	0.048	0.076	253	12	$[13, 25]$
GEDIT	0.902	0.054	0.027	0.072	1294	36	$[65, 129]$

Table 6.2 – *Results of the experiment on random selection of entry-points*

selection. However, in order to run the second step, clustering is not strictly necessary and one may wonder whether this computationally demanding step can be skipped altogether considering the whole call graph as a unique, big cluster of units. Hence, the next experiment addresses the question:

> **RQ3**: *How does* FEAT *perform running entry-point selection directly on the whole call graph?*

In practice the experiment, after the creation of the call graph, simply runs the entry-point selection step on a fictitious cluster containing all units of the project. The results, which for recall are quite interesting, are shown in Tab. 6.3. The main issue with this approach is about usability; having a single large program topos does not allow to benefit from a more selective view on the system like the one produced through the clustering step. Entry-points in this approach have a larger, less selective index of words and neighborhood that make the evaluation and usage of program topoi harder from a user standpoint.

Project	A	F_2	P	R	Units	Oracle size
HEXDUMP	0.775	0.273	0.097	0.500	253	12
GEDIT	0.772	0.305	0.089	0.778	1294	36

Table 6.3 – *Results of the experiment running* FEAT *without the clustering step*

6.3.3 Impact of α, β Parameters

The subject of the next experiments is the whole FEAT approach and focuses on the choice of parameters α, β [4] and, the combination of textual elements available in source code according to the following predefined subsets:

- *Code*: unit names, variable names and, literals.
- *Comments*: remarks preceding and contained in unit blocks.

4. This is the parameter related to program topos definition mentioned in Def.4.4.

- *All*: union of code and comments.

All results shown in this section come from experiments ran on HEXDUMP and GEDIT which are the projects with an associated oracle.

Recall that α parameter sets the relevance of the two elements of the hybrid distance (see Eq.4.6) whereas β determines how many units will be classified as entry-points and then affecting the size of program topoi (see Def.4.4). The following experiment aims at evaluating how various combinations of α and β influence FEAT's results.

> **RQ4**: *How does the parameter pair $\langle \alpha, \beta \rangle$ affect* FEAT*'s performance?*

For both HEXDUMP and GEDIT we let α vary between 0 and 1 (increments of 0.1) and β vary between 0.05 and 0.5 [5] (increments of 0.05) amounting to 220 tests. Textual elements correspond to the *Comment* subset (We show here just one subset of source code elements for the sake of simplicity). In Fig.6.1 we see the graphs showing the results of the experiment on the two projects. In order to pinpoint tests reaching the best performance we created a ranking, shown through colors in Fig.6.1, according to the following criterion: *best* tests are those whose F_2 value is within the 95th percentile of the observed set of F_2 values (blue dots). Red dots are maximum values whereas gray are the remaining ones. The experiment shows that a balanced value of α gives better performance and for β we see that the highest values are reached with the range $\beta \in [0.15, 0.35]$.

The next research question is:

> **RQ5**: *Given a fixed β value, what is the impact of α and of the selection of textual elements (identifier, comments, etc.) on* FEAT*'s performance ?*

We evaluate F-measure, precision and, recall while α goes from 0 to 1 with increments of 0.1 and $\beta = 0.3$ (See previous experiment for explanation on the choice of this value). At the same time we use several textual elements extracted from source code The experiment generated 66 tests whose results are summarized in Fig.6.2. Regarding **RQ5** we notice that, regardless the combination of textual elements, the highest values for F_2 are reached with a balanced value of parameter $\alpha \approx 0.5$ with promising values for recall which in both cases is $R \approx 0.8$. More textual elements, as in the *All* combination, does not always imply better performance, actually the red line in Fig.6.2 related to *Comments* shows similar performance. This might indicate that, semantically speaking, the most relevant content is usually conveyed by comments and not by source code elements such

5. We considered values of β higher than the 50th percentile not interesting because they would create too large program topoi to be useful.

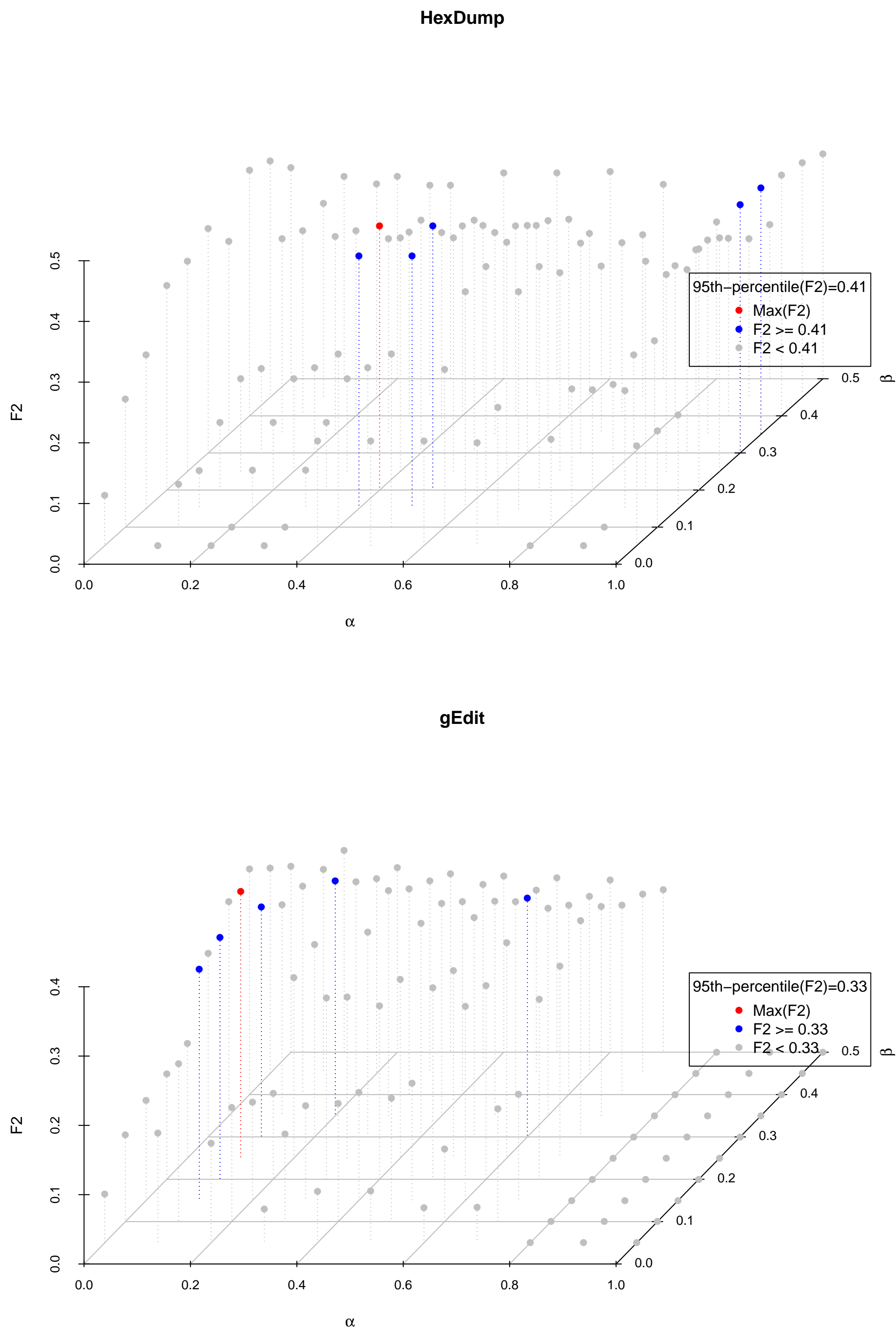

Figure 6.1 – *Experiments showing* FEAT*'s performance wrt. the combination of* α *and* β *values. Parameter* α *varies between* 0 *and* 1 *(with steps of* 0.1*) and* β *between* 0.05 *and* 0.5 *(steps* 0.05*)*

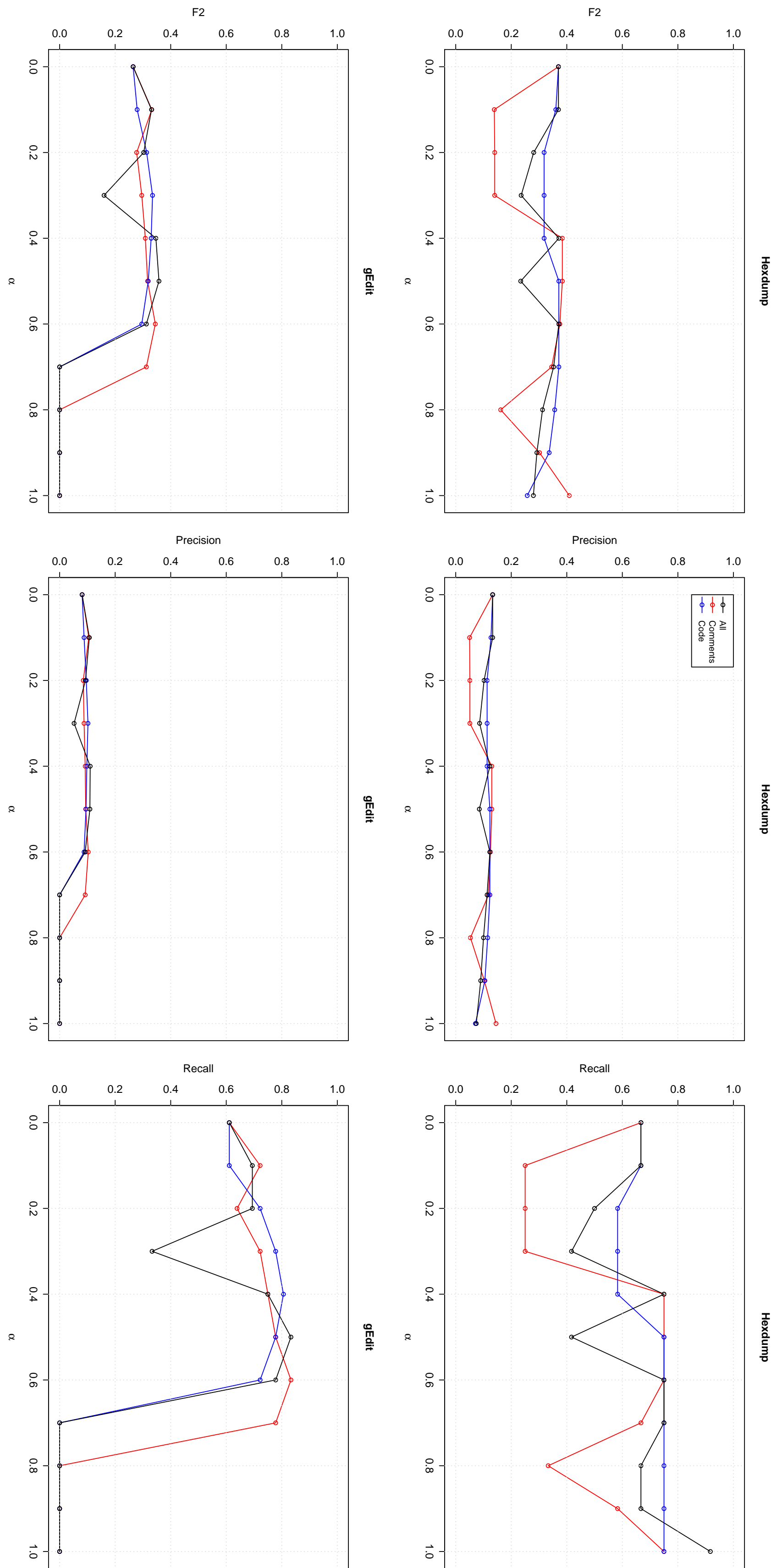

Figure 6.2 – *Plots of* F_2*, precision, and recall of projects* HEXDUMP *and* GEDIT*.* α *varies between* 0 *and* 1 *(with increments of* 0.1*). The colors identify the different combinations of source code elements (see the legend in the top-centered plot).*

as identifiers, literals, etc.

Although it is hard to draw any definite conclusion regarding the selection of α with only a few number of analyzed projects, we can extract some rules of thumb for this purpose. Before applying FEAT, we suggest to run a brief inspection on the code and assess the quality of coding style, especially in terms of meaningfulness of comments and naming conventions. In case of good quality, we recommend to give a higher weight to the lexical analysis part of FEAT by selecting $\alpha \in [0.50, 0.65]$. Otherwise, we suggest to use lower values of α so that less weight is given to the lexical part. Typically, one can start with a value for $\alpha \in [0.35, 0.50]$. For instance, in GEDIT we found a strong, consistent naming convention accompanied by short and precise comments. In this case, our experiments have shown that $\alpha = 0.65$ gave a better result than $\alpha = 0.50$. The choice of a value for β looks easier and in this case we suggest to start with $\beta = 0.3$.

6.3.4 Applicability of Program Topoi

The objective of this experiment is described by the following research question:

RQ7: *How helpful program topoi are when dealing with feature discovery and feature location tasks?*

Here we mimic applications of FEAT to a hypothetical setting. Let us assume that we have been provided with the project GEDIT and that we want to discover which features are implemented and where precisely they are located in the source code. First, we set the two parameters according to the previous experiment's findings that is $\alpha = 0.5$ and $\beta = 0.3$ and we select all textual elements from source code. Second, we run FEAT and go through the generated program topoi. An excerpt of them is shown in Tab.6.4 that lists four entry points. Each line of the table reports the name of the entry-point and part of its dictionary. In addition to that, FEAT provides also the neighborhood of each entry-point as it is shown in Figs.6.3-6.6. Though the names of the units are already self-explanatory we added a brief description of the related capability under each unit. It is evident that the indexes of words and neighborhood graphs are helpful supports in finding capabilities of an unknown system. It is also important to stress that the discovery of capabilities does not require to examine a flat, long list of units but it benefits from the program topoi representation where every program topos is a sorted by relevance short list of entry-points which is easier to consult. In the case of GEDIT without FEAT, we would have an unsorted list of 1,294 units whereas the largest program topos has 97 ranked units and all the others have between 3 and 12 units.

Unit Name (**Feature**)	Index of Words
`_gedit_cmd_file_open` (File open)	chooser, command, dialog, document, file, folder, open, ...
`_gedit_cmd_file_print` (File print)	command, dialog, file, filename, find, folder, preview, print, ...
`_gedit_cmd_search_find` (Find text)	command, document, dialog, find, goto, previous, search, syntax, text, ...
`_gedit_cmd_edit_copy` (Copy to clipboard)	clipboard, command, copy, edit, text, ...

Table 6.4 – *Part of a program topos obtained from the analysis of* GEDIT

	Unit Name	Module
1	`ensure_user_config_dir`	gedit-app.c
2	`save_accels`	gedit-app.c
3	`gedit_app_shutdown`	gedit-app.c
4	**`_gedit_cmd_edit_copy`**	**gedit-commands-edit.c**
5	**`_gedit_cmd_edit_cut`**	**gedit-commands-edit.c**
6	**`_gedit_cmd_edit_paste`**	**gedit-commands-edit.c**
7	`gedit_tab_get_view`	gedit-tab.c
8	`tab_switched`	gedit-window.c
9	`setup_side_panel`	gedit-window.c
10	`bottom_panel_item_added`	gedit-window.c

Table 6.5 – *Searching clipboard capabilities in* GEDIT

The second part of the experiment relates to finding the location (i.e. unit name and module) where a capability is implemented. The only available information is our generic notion of the capability we want to find. On the basis of the same program topoi generated in the first part of this experiment we run a free text query. Let us assume that we want to search where the clipboard management capabilities are located in GEDIT. Then we provide the query "clipboard" and the system provides us the ranking shown in Tab. 6.5. The table shows the first ten results and in the positions 4, 5 and, 6 we have the entry-points corresponding to the usual copy/cut/paste features of the clipboard management and the module containing them. Again, we believe that this is a valuable asset to tackle a feature location task.

Just a technical remarks regarding the last experiment. To accomplish the actual search we use the full-text indexing and searching capabilities of MySQL (program topoi are persisted in a MySQL RDBMS. More architectural details in Chap. 5) we believe that a more tailored approach to free-text search using all the informations gathered by FEAT would provide even better performance.

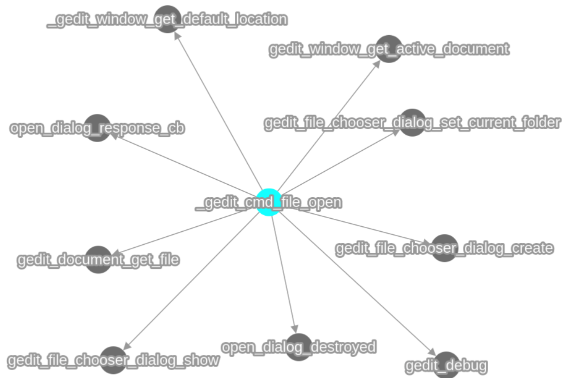

Figure 6.3 – *File open*

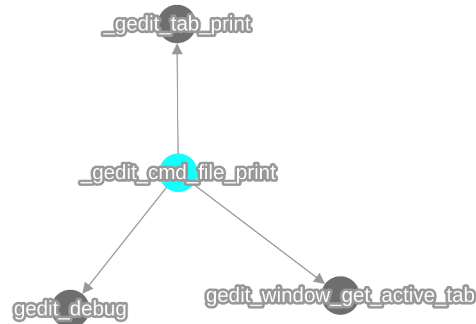

Figure 6.4 – *File print*

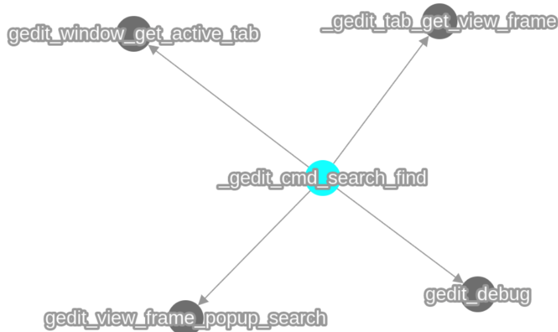

Figure 6.5 – *Find text*

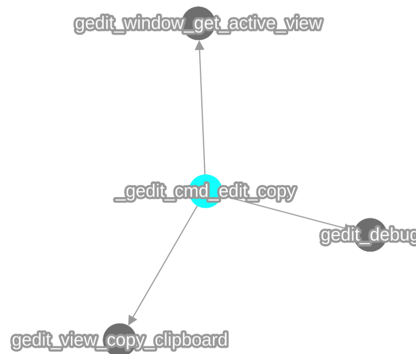

Figure 6.6 – *Copy to clipboard*

This experiment has shown through a concrete example not only that FEAT can be useful but also its duality; being able to deal with both feature extraction and location tasks.

6.4 Benefits of FEAT's Hybrid Distance

At the very core of FEAT there is the dual perspective explained in Chap. 4 equipped by the hybrid distance described in Eq. 4.6. This constitutes also the main difference between FEAT and similar approaches. The last experiment evaluates whether the adoption of a hybrid distance in FEAT brings any added value in comparisons with other existing, similar approaches.

> **RQ8**: *In locating software features, are there any benefits in combining structural and semantic elements of source code?*

Marcus *et al.* [47] applied LSI (latent semantic indexing) [6] to map concepts ex-

6. LSI is usually considered a synonym for LSA even though some tend to use the acronym

Unit/Data	File
`wrapFont()`	gui-menubar.c
`mo_set_fonts()`	gui-menubar.c
`mo_get_font_size_from_res()`	gui.c
`XtResources resources[]`	HTML.c
`PSFont()`	HTML-PSformat.c

Table 6.6 – *Oracle used in the feature location experiment with* MOSAIC

pressed in natural language to relevant parts of the source code. In this approach, users formulate queries and evaluate the results returned by the system. The subject used in this work is MOSAIC web browser. The case study aims at locating features related to "font"-handling and, for this purpose, the authors created an oracle (see Tab.6.6) made of four functions and one data type. They supplied several queries to the system having an increasing complexity: starting from simpler queries such that *"font"* and, successively adding more words. The query: *"font style bold italics large small regular"* found fifteen functions involved in font management including all the functions included in the oracle, while the query using only the word *"font"*, did not find any relevant element. In order to compare FEAT with this approach, we analyzed MOSAIC's source code and using the query language of the graph database (neo4j), we ran the first query (*"font"*) to search all clusters having "font" as part of the entry-point index. FEAT, just with this simple query, managed to identify all the functions of the oracle. The results of this test are given in Tab.6.7. It is worth noticing that our approach with $\alpha = 0.5$, even with the simplest query, performed always better than the one based on LSI. LSI never reaches a precision higher than 0.33.

But the most important point to highlight is that FEAT can automatically reveal more than LSI about how units are related. Given a free text query, LSI cannot detect the functional dependencies among units. It is only able to reveal similarities among units on the basis of the usage of natural language, the drawback is that its outcome is a flat list of unit names. Unlike LSI, FEAT splits *font*-related units into two clusters as shown in Fig. 6.7 and Fig. 6.8. These figures display the output produced by querying the system with query "font", where orange nodes represent clusters and blue nodes are units. By looking at the source code of MOSAIC, we discovered that the *font*-related unit in the oracle (see Tab.6.6) called `PSfont()`, shown in Fig.6.7, is part of the postscript printing feature. The three remaining units of the oracle (`wrapFont()`, `mo_set_fonts()` and, `mo_get_font_size_from_res()`), shown in Fig.6.8, are part of the user interface feature (window creation). So, even though all functions in the oracle have a high lexical similarity w.r.t. to the query *"font"*, they are not similar at all when looking at them from the functional point of view. FEAT, unlike LSI, taking into account both semantic and structural elements through its hybrid distance metric

LSI when LSA is applied to information retrieval applications.

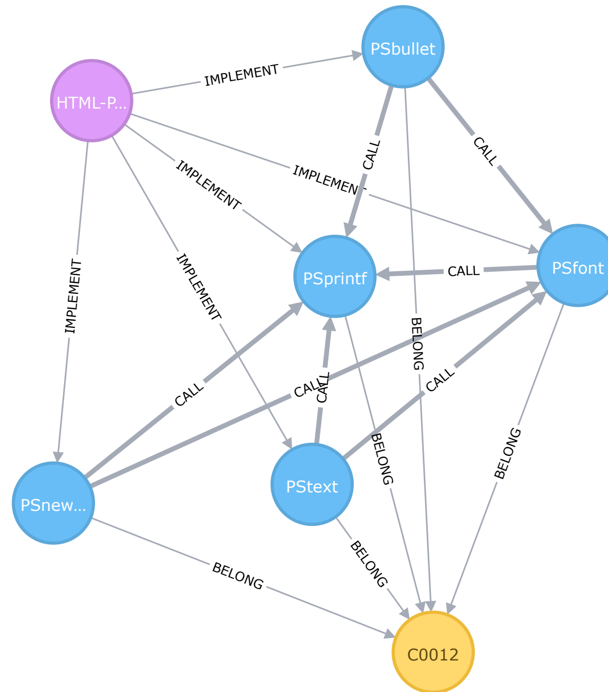

Figure 6.7 – *NCSA Mosaic web browser. Clustering of postscript printing feature.*

obtains a more accurate representation of a software system's features.

FEAT		LSI	
P	R	P	R
0.44	1.00	0.00	0.00

Table 6.7 – *Comparison of FEAT with a LSI-based approach, while running query:* "font".

6.5 Scalability Evaluation

The adoption of a new methodology is determined also by its scalability, then in this section we address the following research question:

> **RQ9**: *What is the correlation between* FEAT*'s running time and memory usage w.r.t. software projects characteristics such as: number of units, size of the dictionary, LOC and, call graph density?*

The results of this experiment, involving the projects listed in Tab.6.8, are reported in Fig.6.9. On the left hand side we have graphs reporting the running

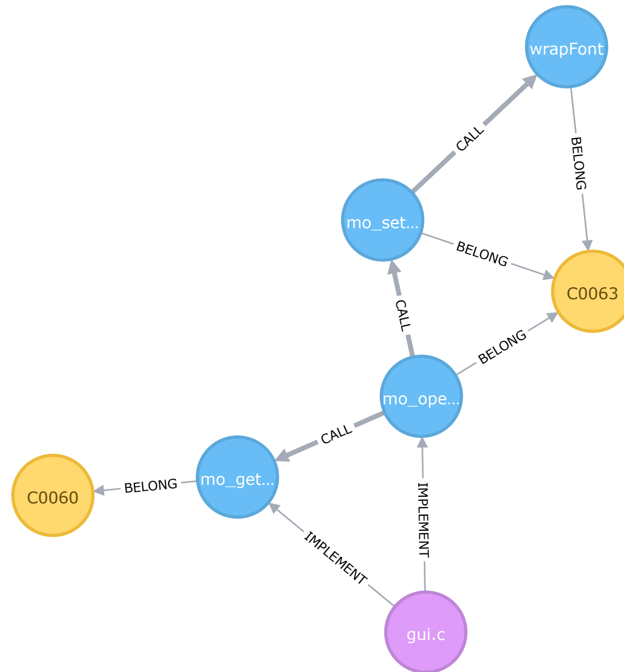

Figure 6.8 – *NCSA Mosaic web browser. Clustering of browser window creation feature.*

time ($RT\,[\mathrm{s}]$) of the experiment while α varies in $\{0, 0.5, 1\}$. For every value of α we have two curves: the total time employed by FEAT (solid line) and the clustering time (dashed line). The difference between the two curves is equal to the time needed by the preprocessing step. In all cases the time of the entry point selection step can be neglected.

The right hand side of Fig.6.9 contains the graphs about memory usage. In this case we plot only one graph per characteristic because no meaningful differences have been observed among the various values of α. Let us focus on the running time graphs. The analysis of running time leads to two main observations. First, for larger projects performance is negatively affected by values of $\alpha > 0$ which means that the computation of the textual distance (Eq.4.3) and the coherence criterion (Eq.4.8) are demanding. Indeed the graph KUnit/Time shows a steep increase for $\alpha = 1.0$ and number of units higher than $2\,\mathrm{KUnit}$. A similar behavior can be observed in the KWords/Running Time graph where Running time grows faster for values higher than $\approx 2.3\,\mathrm{KWord}$. Second, the clustering step, whose complexity is $\Theta(n^2 \log(n))$, is the most costly one when $\alpha > 0$ while for $\alpha = 0$ running time is dominated by the preprocessing step. Density shows a negative correlation w.r.t. both time and memory. Regarding the memory usage no clear patterns emerged, nevertheless we observe a fast growth in relationship with $KLOC \geq 70$.

In conclusion, FEAT shows acceptable performance with projects counting up to $1.3\,\mathrm{KUnit}$ and dictionaries with size up to $\approx 2\,\mathrm{KWords}$ producing results in

	Project	LOC	#Unit	#File	Dict.	ρ
1	Linux FS EXT2	8,445	180	14	748	0.0201
2	Hexadec. Viewer	12,053	254	13	764	0.0091
3	GNU bc Calculator 1.06	12,851	215	20	723	0.0204
4	Intel Ethernet Drivers and Util.	30,499	581	16	1,479	0.0062
5	Ultradefrag v7.0	34,637	1,112	74	1,874	0.0054
6	Zint Barcode Generator v2.3.0	38,095	345	43	1,275	0.0134
7	GNU Editor v3.2	42,718	1,370	59	1,048	0.0021
8	bash v1.0	70,955	1,477	128	2,216	0.0027
9	Linux IPv4	84,606	2,216	127	3,211	0.0011
10	x3270 Terminal Emulator v3.5	91,449	1,881	136	3,008	0.0025

Table 6.8 – *Open Source software projects used in the scalability evaluation experiment*

about 20 min, for greater values of these characteristics the full hybrid approach becomes rapidly too costly.

6.5.1 FEAT at Software Heritage

Software Heritage[7](SH) is an initiative owned by Inria whose aim is to collect, preserve and, share all publicly available source code. It is more than just providing a central point for accessing source code, its main objective is the preservation of an important artifact of human ingenuity: software.

At the moment, SH holds almost 84 million projects amounting to more than 4,4 billion files. It is evident how challenging it is to provide search capabilities over such an enormous repository. Last year we began a collaboration with SH whose objective is the adoption of FEAT as a searching tool.

The final experiment presented in this chapter is about a subset of projects downloaded from SH and processed with FEAT. We selected 600 projects coming from various domains, written in `C` language whose (compressed) size is between 50 Mbytes and 150 Bytes. SH's repository is organized as a Merkle tree and the projects, which are nodes in this tree, are identified through their SHA1 code.

Thirty-two projects were discarded because they are too large to be processed, this limitation does not come from FEAT but from the hosting platform CRYSTAL which interprets long running tasks (i.e. taking several hours) as system

7. www.softwareheritage.org

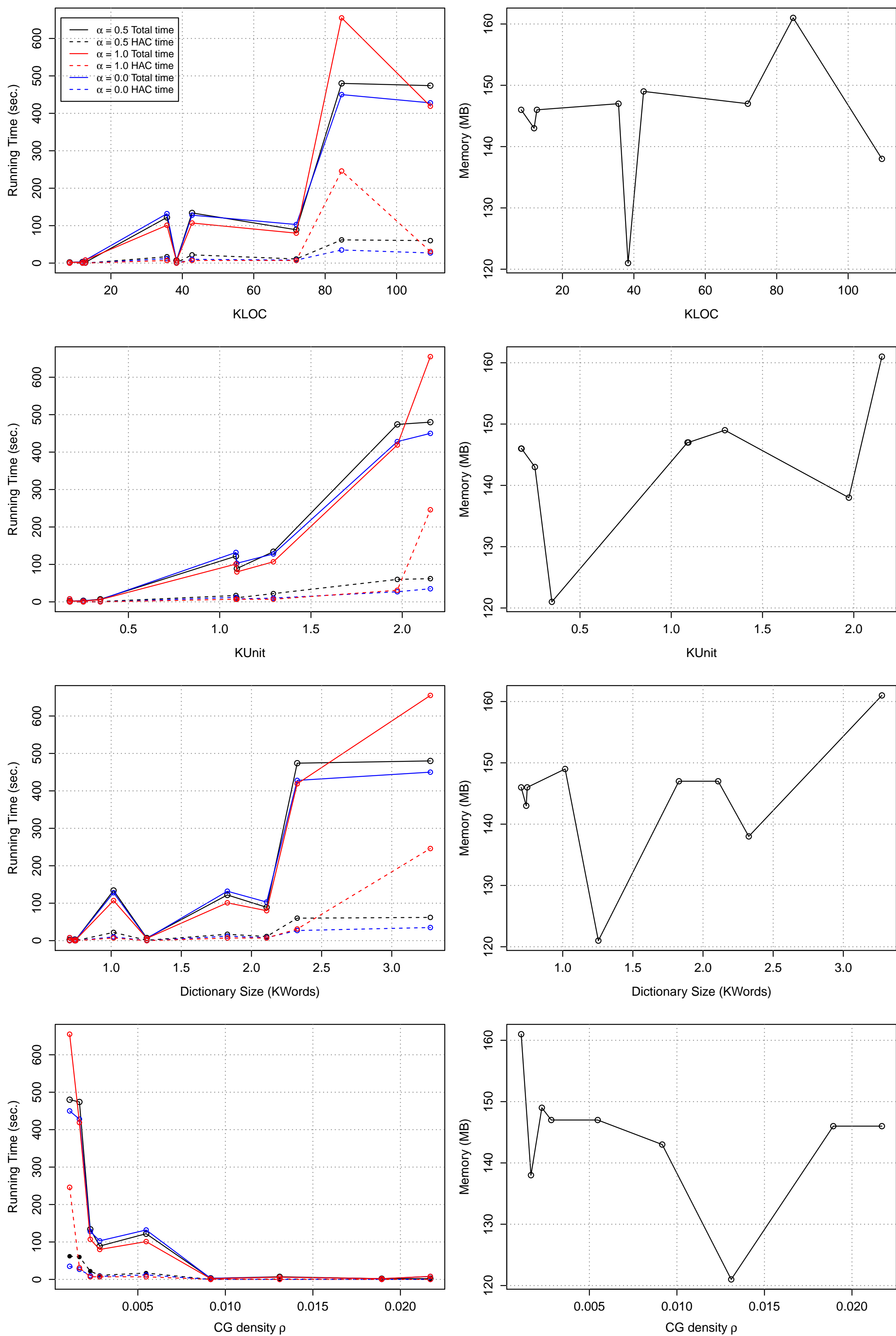

Figure 6.9 – *Correlation between running time and memory usage w.r.t. LOC, number of units, size of the dictionary and, density of* CG

		[min,max]
Projects	448	
Files	48,187	
Units	428,480	[3, 4630]
LOC	25,594,995	[37, 1611756]
Dictionary		[4, 5980]

Table 6.9 – *Summary of* FEAT*'s experiment at Software Heritage*

hangs. This needs a refactoring of the platform for such peculiar activities. Other 120 projects have been automatically discarded by FEAT because they are too small to be analyzed (i.e. very few lines of code). Hence, the results we are going to present come from 448 projects. Some key figures, summarizing the experiment, are shown in Tab. 6.9.

The plot in Fig. 6.10 shows the same measures seen in the other scalability experiment; running time and memory usage wrt. units, lines of code, dictionary size and, density of the call graph. Having a larger sample of projects made emerge how most of these measures are too chaotic to be used as an estimator for FEAT's performance. The only exception is for the number of units vs. running time here we can recognize a polynomial trend at least until a threshold which can be identified around 4,500 units. Indeed, in Fig. 6.11 we determined a third order polynomial fitting ($RT(u) \approx au + bu^2 + cu^3 + d$) shown in red in the graph. After that threshold, running time seems to grow much faster but, unfortunately, with the current version of the tool it is not possible to run experiments on projects with a higher amount of units.

It is clear that processing such huge amount of data is a major undertaken and it requires a lot of work for improving the efficiency of the approach, nevertheless this experiment has shown that FEAT, through program topoi, is a viable approach for the creation of smart, efficient indexes making semantic and structural information available even on large code repository.

6.6 Threats to Validity

This section discusses some elements that can be considered a threat to the 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.

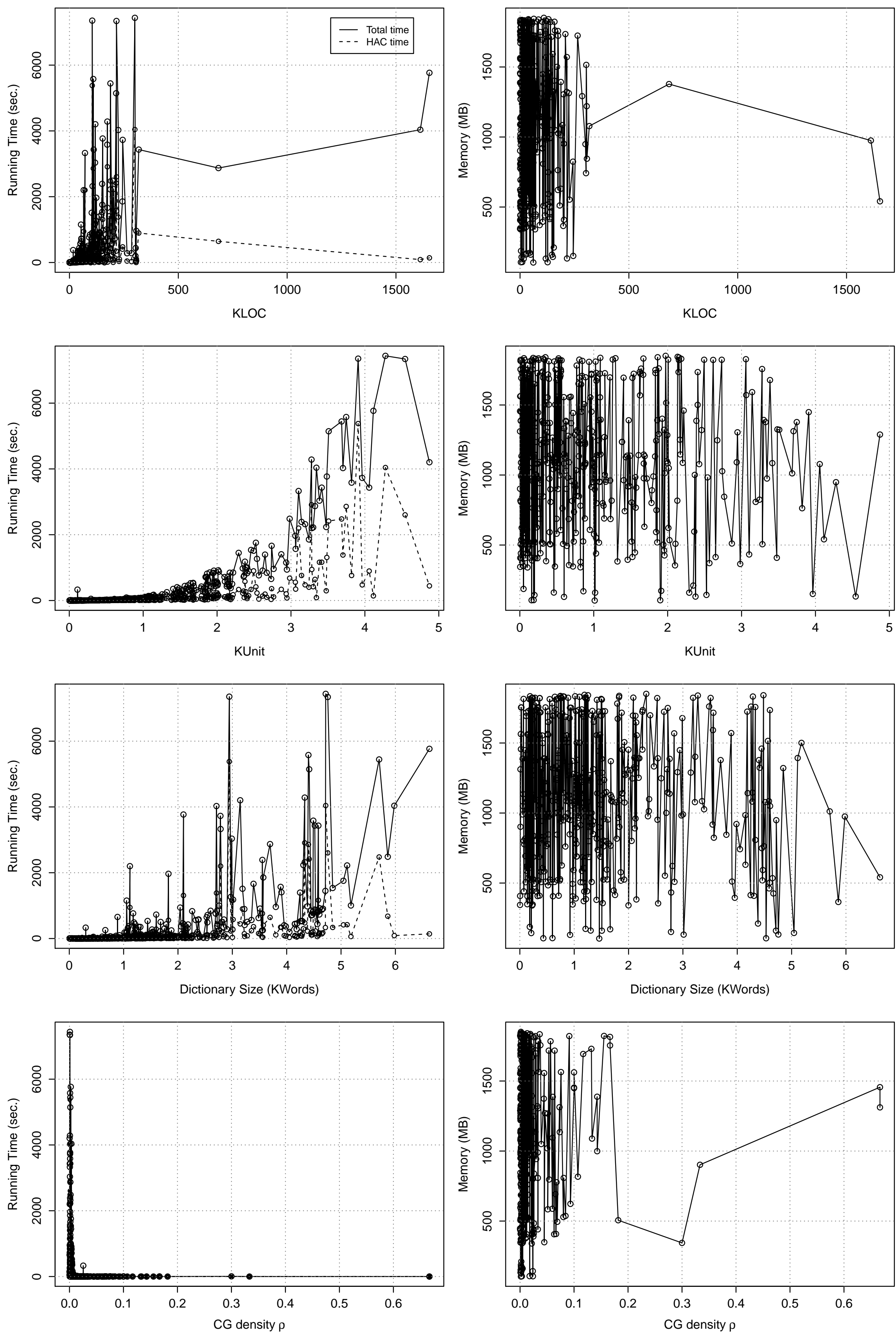

Figure 6.10 – *Correlation between running time and memory usage w.r.t. LOC, number of units, size of the dictionary and, density of* CG *with* FEAT

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 create an oracle for each additional project.

- 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 mitigate 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.

Conclusion

In this chapter we presented the several experiments that we used to: compare FEAT with some baselines, evaluate it from a quantitative standpoints, obtain indications on its α and β parameters setting, create practical application in real settings as Software Heritage.

Obviously, as we have already said, FEAT needs to be improved especially when dealing with large repositories, nevertheless the results we obtained both in terms of program topoi discovering and performance are encouraging and allow us to conclude that FEAT could be deployed and applied in real settings.

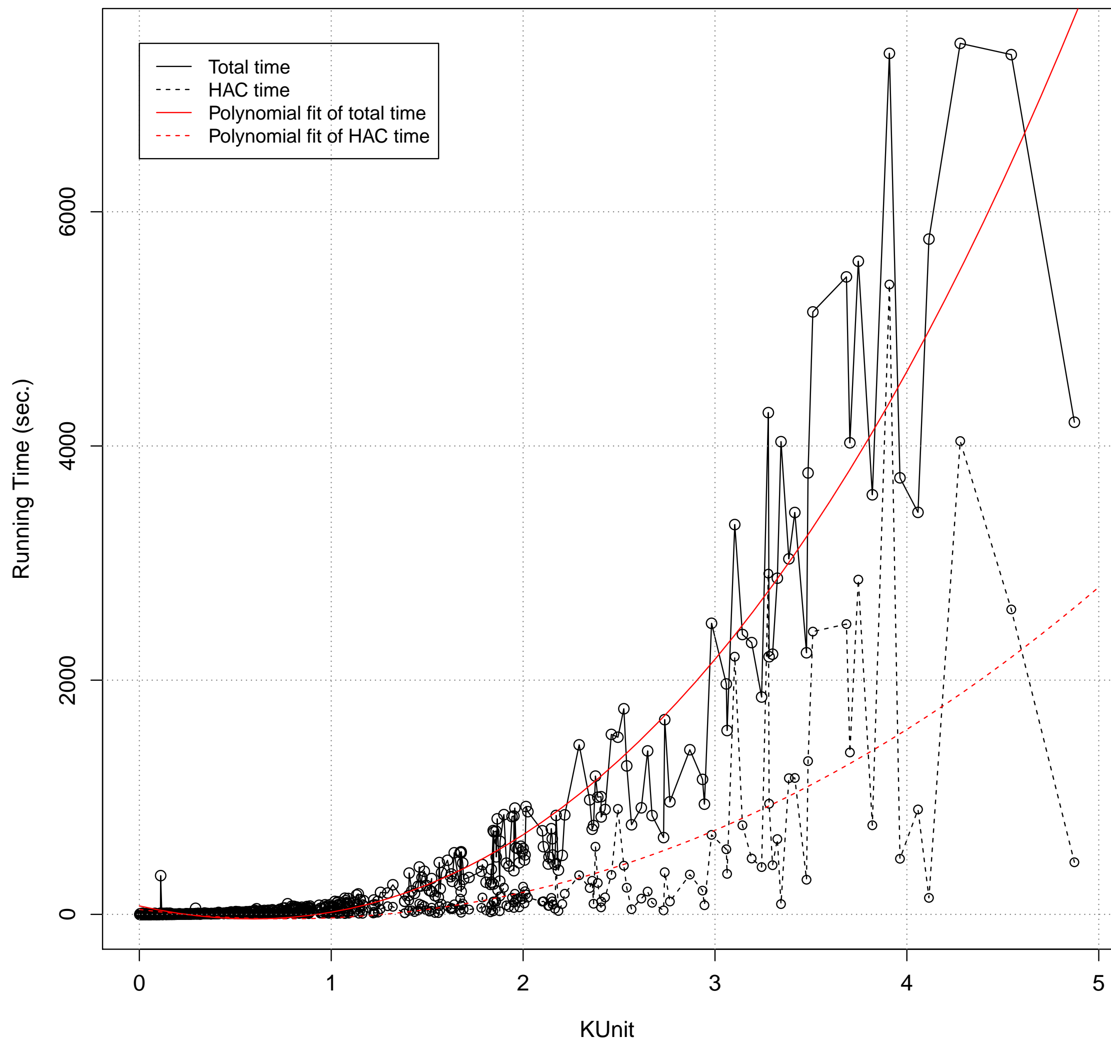

Figure 6.11 – *Running time vs. number of units. The red lines represent the polynomial fitted models*

VII

Conclusions

Program topoi are concrete and useful representations of software systems' features. When a software system has evolved over a long period of time, topoi extraction provides stakeholders an updated view on the system features which is a valuable asset to get more maintainable, reliable and, better documented systems. To address the challenge of topoi extraction, in this thesis we presented FEAT a three-steps method based on the creation of a model of the system, hierarchical agglomerative clustering (HAC) and, entry-points selection. In FEAT, HAC exploits a novel hybrid distance combining semantic 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 and textual coherence 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 more than 600 open-source projects, amounting to more than 25M LOC in total, and by applying it to Software Heritage we showed that FEAT is a feasible approach for automatically discovering program topoi directly from source code. This thesis showed that FEAT can deal with medium-sized software projects (more than 1,000,000 LOC) in a reasonable amount of time.

Perspectives

This thesis has shown how an automated approach to program comprehension can benefit from a model based on both semantic and structural information con-

tained in source code. The aim is to exploit some of the elements that experienced programmers use when they look for concepts implementation in source code. Clearly, program comprehension involves more than just the two elements of our hybrid perspective; as we presented in Sec.2.2, experienced programmers think in term of *concepts* and they project concepts on actual implementation when looking for features in source code. FEAT models a software system as a combination of semantics and structure information. Namely, it evaluates two units as similar if they share both a common usage of the natural language and they are close to each other in the caller-callee relationship represented in a call graph. Semantics and structure contained in source code are not the only available elements when approaching a more accurate representation of concepts in program comprehension. But there are also other meaningful representations of software systems like *data dependence graph* which might be considered to enrich FEAT's model. The introduction of a third element to FEAT's perspective would require a suitable way to combine them; Could still a linear combination of the three elements work to this aim? Would multidimensional arrays (tensors) provide a better representation of a software system's model? These are interesting questions deserving to be analyzed in the future.

In the course of this thesis we gathered several aspects that we would like to deeper investigate in the future in order to improve FEAT. Some of them lie at the ground of the approach, whereas others can be considered more technical improvements aimed at enlarging the audience of potential users of FEAT. Let us start with the more theoretical aspects.

- Entry-points are provided with a graph neighborhood and a list of alphabetically sorted words. Sometimes this list can be quite long and hard to interpret. The problem with the text embedded in bodies of units is that it does not follow any document structure. Nevertheless, applying summarization techniques on comments can be helpful to address this issue and provide a more understandable text to users.

- Program topoi can be exploited to automatically uncover links between source code and test scripts by means of program topoi's index of words. Several approaches have been proposed to perform data mining on program and test-execution traces but they usually require either the instrumentation of source code or the evaluation of execution traces. FEAT could provide a static, fully automated approach to the creation of traceability map between source code and test cases.

- In the line of work on the traceability map between source code and test cases using FEAT, it would be interesting to model and to take into account a set of preferences on tests given a set a features. The idea here is to be able to start by testing the most important features w.r.t., user's preferences. The preferences can be also automatically elicited from test cases where a test

case can dominate a set of test cases in terms of coverage testing criteria.

More on the technical side of FEAT, we foresee the following future works:

- Dealing with larger projects. In this thesis, with the prototypical version of CRYSTAL.FEAT, we found a limitation in handling projects of size higher than 4.5K units. Our approach could be improved by: (i) task parallelization (ii) incremental build of the model in order to deal with new versions of already processed projects.

- Incremental creation of FEAT's model in order to take into account small variations in already processed projects without requiring a new elaboration from scratch.

Bibliography

[1] C. Aaron, M. Newman, and C. Moore. Finding community structure in very large networks. *Physical Reviews E.*, 70, 2004. 50

[2] S. L. Abebe and P. Tonella. Extraction of domain concepts from the source code. *Sci. Comput. Program.*, 98:680–706, 2015. ix, 20, 21

[3] C. C. Aggarwal, A. Hinneburg, and D. A. Keim. On the surprising behavior of distance metrics in high dimensional spaces. In *Database Theory - ICDT 2001, 8th International Conference, London, UK, January 4-6, 2001, Proceedings.*, pages 420–434, 2001. 7

[4] R. Al-Msie'deen, A. Seriai, M. Huchard, C. Urtado, S. Vauttier, and H. E. Salman. Feature location in a collection of software product variants using formal concept analysis. In *Safe and Secure Software Reuse - 13th International Conference on Software Reuse, ICSR 2013, Pisa, Italy, June 18-20. Proceedings*, pages 302–307, 2013. 22

[5] N. S. Altman. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician*, 46(3):175–185, 1992. 11

[6] P. Andritsos and V. Tzerpos. Information-theoretic soft. clustering. *IEEE Trans. Software Eng.*, 31(2):150–165, 2005. x, 30

[7] R. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1 edition, 1957. 6, 7

[8] P. V. Biron. Backpropagation: Theory, architectures, and applications, edited by yves chauvin and david e. rumelhart. *JASIS*, 48(1):88–89, 1997. 10

[9] R. E. Brooks. Towards a theory of the cognitive processes in computer programming. *Int. J. Hum.-Comput. Stud.*, 51(2):197–211, 1999. 17

[10] I. M. Chakravarty, R. G. Laha, and J. D. Roy. *Handbook of methods of applied statistics*. McGraw-Hill, New York, NY, 1967. 58

[11] K. Chen and V. Rajlich. Case study of feature location using dependence graph. In *8th International Workshop on Program Comprehension (IWPC 2000), 10-11 June 2000, Limerick, Ireland*, pages 241–247, 2000. viii

[12] A. Christl, R. Koschke, and M. D. Storey. Automated clustering to support the reflexion method. *Information & Software Technology*, 49(3):255–274, 2007. 26

[13] A. Clauset, C. R. Shalizi, and M. E. J. Newman. Power-law distributions in empirical data. *SIAM Review*, 51(4):661–703, 2009. 58

[14] R. Damasevicius, P. Paskevicius, E. Karciauskas, and R. Marcinkevicius. Automatic extraction of features and generation of feature models from java programs. *ITC*, 41(4):376–384, 2012. 22

[15] M. M. Deza and E. Deza. *Encyclopedia of Distances*. Springer Berlin Heidelberg, 2009. 40

[16] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013. 20

[17] L. Donetti and M. A. Muñoz. Detecting network communities: a new systematic and efficient algorithm. *Journal of Statistical Mechanics: Theory and Experiment*, 10:P10012, 2004. viii, 2, 50

[18] 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 *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 181–190, 2011. ix, 21

[19] V. Estivill-Castro and A. T. Murray. Discovering associations in spatial data - an efficient medoid based approach. In *Research and Development in Knowledge Discovery and Data Mining, Second Pacific-Asia Conference, PAKDD-98, Melbourne, Australia, April 15-17, 1998, Proceedings*, pages 110–121, 1998. 29

[20] J. R. Firth. Applications of general linguistics. *Transactions of the Philological Society*, 56(1):1–14, 1957. 32

[21] P. W. Foltz, W. Kintsch, and T. K. Landauer. The measurement of textual coherence with latent semantic analysis. *Discourse Processes*, 25(2-3):285–307, 1998. viii, 2, 50

[22] G. W. Furnas, S. C. Deerwester, S. T. Dumais, T. K. Landauer, R. A. Harshman, L. A. Streeter, and K. E. Lochbaum. Information retrieval using a singular value decomposition model of latent semantic structure. In *SIGIR'88, Proceedings of the 11th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Grenoble, France, June 13-15, 1988*, pages 465–480, 1988. 32, 33

[23] E. N. Gilbert. Random graphs. *Ann. Math. Statist.*, 30(4):1141–1144, 12 1959. 83

[24] G. H. Golub and C. F. V. Loan. *Matrix computations (3. ed.)*. Johns Hopkins University Press, 1996. 31

[25] S. Grant, J. R. Cordy, and D. B. Skillicorn. Using heuristics to estimate an appropriate number of latent topics in source code analysis. *Sci. Comput. Program.*, 78(9):1663–1678, 2013. 21

[26] D. J. Hand and K. Yu. Idiot's bayes—not so stupid after all? *International Statistical Review*, 69(3):385–398, 2001. 10

[27] J. Hartigan and M. Wong. Algorithm AS 136: A K-means clustering algorithm. *Applied Statistics*, pages 100–108, 1979. 14

[28] G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006. 16

[29] T. K. Ho. Random decision forests. In *Third International Conference on Document Analysis and Recognition, ICDAR 1995, August 14 - 15, 1995, Montreal, Canada. Volume I*, pages 278–282, 1995. 9

[30] C. Ieva, A. Gotlieb, S. Kaci, and N. Lazaar. Discovering program topoi through clustering. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, February 2-7, 2018*, 2018. 3

[31] C. Ieva, A. Gotlieb, S. Kaci, and N. Lazaar. Discovering program topoi via hierarchical agglomerative clustering. *IEEE Trans. Reliability*, 67(3):758–770, 2018. 3

[32] A. K. Jain. Data clustering: 50 years beyond k-means. In *Machine Learning and Knowledge Discovery in Databases, European Conference, ECML/PKDD 2008, Antwerp, Belgium, September 15-19, 2008, Proceedings, Part I*, pages 3–4, 2008. 25

[33] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley, 1990. x, 30

[34] J. Koenemann and S. P. Robertson. Expert problem solving strategies for program comprehension. In *Conference on Human Factors in Computing Systems, CHI 1991, New Orleans, LA, USA, April 27 - May 2, 1991, Proceedings*, pages 125–130, 1991. 18

[35] T. Kohonen. Neurocomputing: Foundations of research. chapter Self-organized Formation of Topologically Correct Feature Maps, pages 509–521. MIT Press, Cambridge, MA, USA, 1988. 17

[36] A. Kuhn, S. Ducasse, and T. Gîrba. Enriching reverse engineering with semantic clustering. In *12th Working Conference on Reverse Engineering, WCRE 2005, Pittsburgh, PA, USA, November 7-11, 2005*, pages 133–142, 2005. x, 26, 30

[37] A. Kuhn, S. Ducasse, and T. Gîrba. Semantic clustering: Identifying topics in source code. *Information & Software Technology*, 49(3):230–243, 2007. ix, 22, 82

[38] T. K. Landauer and S. T. Dutnais. A solution to plato's problem: The latent semantic analysis theory of acquisition, induction, and representation of knowledge. *PSYCHOLOGICAL REVIEW*, 104(2):211–240, 1997. 33

[39] B. Lemaire and G. Denhière. Effects of high-order co-occurrences on word semantic similarities. *CoRR*, abs/0804.0143, 2008. 33

[40] S. Letovsky. Cognitive processes in program comprehension. *Journal of Systems and Software*, 7(4):325–339, 1987. 18

[41] E. Linstead, P. Rigor, S. K. Bajracharya, C. V. Lopes, and P. Baldi. Mining concepts from code with probabilistic topic models. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, pages 461–464, 2007. ix, 21

[42] B. Liu. *Web Data Mining: Exploring Hyperlinks, Contents, and Usage Data*. Data-Centric Systems and Applications. Springer, 2007. 14

[43] J. Macqueen. Some methods for classification and analysis of multivariate observations. In *In 5-th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1967. 25

[44] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to information retrieval*. Cambridge University Press, 2008. 6, 11, 12, 14, 15, 26, 39, 42, 49, 84, 85

[45] A. Marcus and S. Haiduc. Text retrieval approaches for concept location in source code. In *Software Engineering - International Summer Schools, ISSSE 2009-2011, Salerno, Italy. Revised Tutorial Lectures*, pages 126–158, 2011. viii

[46] A. Marcus and S. Haiduc. Text retrieval approaches for concept location in source code. In *Software Engineering - International Summer Schools, ISSSE 2009-2011, Salerno, Italy. Revised Tutorial Lectures*, pages 126–158, 2011. 19

[47] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *11th Working Conference on Reverse Engineering, WCRE 2004, Delft, The Netherlands, November 8-12, 2004*, pages 214–223, 2004. 26, 82, 93

[48] P. W. McBurney, C. Liu, and C. McMillan. Automated feature discovery via sentence selection and source code summarization. *Journal of Software: Evolution and Process*, 28(2):120–145, 2016. ix, 21, 26

[49] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, Dec 1943. 10

[50] C. McMillan, N. Hariri, D. Poshyvanyk, J. Cleland-Huang, and B. Mobasher. Recommending source code for use in rapid software prototypes. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 848–858, 2012. ix, 21

[51] B. S. Mitchell. A heuristic approach to solving the software clustering problem. In *19th International Conference on Software Maintenance,2003*, pages 285–288, 2003. x, 29

[52] T. M. Mitchell. *Machine learning*. McGraw Hill series in computer science. McGraw-Hill, 1997. 8, 9, 10

[53] L. Moreno. Summarization of complex software artifacts. In *36th International Conference on Software Engineering, ICSE '14, Companion Proceedings, Hyderabad, India, May 31 - June 07, 2014*, pages 654–657, 2014. 22

[54] L. Moreno and A. Marcus. Jstereocode: automatically identifying method and class stereotypes in java code. In *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*, pages 358–361, 2012. 22

[55] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *SIGSOFT '95, Proceedings of the Third ACM SIGSOFT Symposium on Foundations of Software Engineering, Washington, DC, USA, October 10-13, 1995*, pages 18–28, 1995. 26

[56] F. Murtagh and P. Contreras. Methods of hierarchical clustering. *CoRR*, abs/1105.0121, 2011. 27

[57] C. R. Myers. Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. *Phys. Rev. E*, 68:046116, Oct 2003. 62

[58] M. E. J. Newman. Power laws, pareto distributions and zipf's law. *Contemporary physics*, 46(5):323–351, 2005. 59

[59] M. E. J. Newman. Modularity and community structure in networks. In *Proc. of the Nat. Academy of Sciences of USA*, volume 103, pages 8577–8582, 2006. 49

[60] A. Y. Ng and M. I. Jordan. On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes. In *Advances in Neural Information Processing Systems 14 [Neural Information Processing Systems: Natural and Synthetic, NIPS 2001, December 3-8, 2001, Vancouver, British Columbia, Canada]*, pages 841–848, 2001. 11

[61] J. Novak. Learning, creating, and using knowledge : Concept maps as facilitative tools in schools and corporations / j.d. novak. 6, 01 1998. 18

[62] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986. 9

[63] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993. 9

[64] V. Rajlich. Intensions are a key to program comprehension. In *The 17th IEEE International Conference on Program Comprehension, ICPC 2009, Vancouver, British Columbia, Canada, May 17-19, 2009*, pages 1–9, 2009. 18

[65] V. Rajlich and N. Wilde. The role of concepts in program comprehension. In *10th International Workshop on Program Comprehension (IWPC 2002), 27-29 June 2002, Paris, France*, pages 271–278, 2002. 18

[66] M. A. Ranzato, C. Poultney, S. Chopra, and Y. L. Cun. Efficient learning of sparse representations with an energy-based model. In B. Schölkopf, J. C. Platt, and T. Hoffman, editors, *Advances in Neural Information Processing Systems 19*, pages 1137–1144. MIT Press, 2007. 16

[67] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej. How do professional developers comprehend software? In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 255–265, 2012. 20

[68] J. Rubin and M. Chechik. A survey of feature location techniques. In *Domain Engineering, Product Lines, Languages, and Conceptual Models*, pages 29–58. 2013. viii

[69] G. Salton, A. Wong, and C. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620, 1975. 32

[70] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, 1959. 5

[71] B. Schölkopf, R. C. Williamson, A. J. Smola, J. Shawe-Taylor, and J. C. Platt. Support vector method for novelty detection. In *Advances in Neural Information Processing Systems 12, [NIPS Conference, Denver, Colorado, USA, November 29 - December 4, 1999]*, pages 582–588, 1999. 16

[72] J. Shawe-Taylor and N. Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, 2004. 12

[73] M. Shtern and V. Tzerpos. Clustering methodologies for software engineering. *Adv. Software Engineering*, 2012:792024:1–792024:18, 2012. x, xii, 26, 29, 36

[74] A. Siddharthan. Christopher d. manning and hinrich schutze. *Foundations of Statistical Natural Language Processing*. MIT press, 2000. ISBN 0-262-13360-1, 620 pp. $64.95/£44.95 (cloth). *Natural Language Engineering*, 8(1):91–92, 2002. 83

[75] M. D. Storey. Theories, tools and research methods in program comprehen-

sion: past, present and future. *Software Quality Journal*, 14(3):187–208, 2006. 17

[76] É. D. Taillard. Heuristic methods for large centroid clustering problems. *J. Heuristics*, 9(1):51–73, 2003. 29

[77] P. D. Turney and P. Pantel. From frequency to meaning: Vector space models of semantics. *J. Artif. Intell. Res.*, 37:141–188, 2010. 31, 32

[78] V. Tzerpos and R. C. Holt. ACDC: an algorithm for comprehension-driven clustering. In *Proceedings of the Seventh Working Conference on Reverse Engineering, 2000*, pages 258–267, 2000. x, 30

[79] V. Vapnik and A. Lerner. Pattern Recognition using Generalized Portrait Method. *Automation and Remote Control*, 24, 1963. 12

[80] S. Wiedenbeck. Beacons in computer program comprehension. *International Journal of Man-Machine Studies*, 25(6):697–709, 1986. 18

[81] K. E. Wiegers. *Software Requirements*. Microsoft Press, Redmond, WA, USA, 2 edition, 2003. 1, 19

[82] T. A. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *4th Working Conference on Reverse Engineering, 1997*, pages 33–43, 1997. xi, 30

[83] C. Wilke, S. Altmeyer, and T. Martinetz. Large-scale evolution and extinction in a hierarchically structured environment. In *Proceedings of the 6th International Conference on Artificial Life*, pages 266–274. MIT Press, Cambridge, 1998. 63

[84] X. Xu, C. Lung, M. Zaman, and A. Srinivasan. Program restructuring through clustering techniques. In *4th IEEE International Workshop on Source Code Analysis and Manipulation, 2004*, pages 75–84, 2004. x, 30

[85] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang. SNIAFL: towards a static non-interactive approach to feature location. In *26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom*, pages 293–303, 2004. 22