CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Migrating and Evolving Software Product Lines:

An Industrial Case Study of Feature Location and Visualization Techniques

Master's thesis in Computer science and engineering

BERIMA KWEKU ANDAM

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2018

Master's thesis 2018

Migrating and Evolving Software Product Lines:

An Industrial Case Study of Feature Location and Visualization Techniques

BERIMA ANDAM

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Division of Software Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2018

Migrating and Evolving Software Product Lines:
An Industrial Case Study of Feature Location and Visualization Techniques
BERIMA ANDAM

Supervisors: Prof. Thorsten Berger, Prof. Michel R. V. Chaudron,
Software Engineering Department
Department of Computer Science and Engineering
Division of Software Engineering
Chalmers University of Technology and University of Gothenburg

Supervisor: Dr. Andreas Burger
ABB Corporate Research
Ladenburg, Germany

Examiner: Prof. Regina Hebig
Chalmers University of Technology and University of Gothenburg

Master's Thesis 2018
Department of Computer Science and Engineering
Division of Software Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Migrating and Evolving Software Product Lines:
An Industrial Case Study of Feature Location and Visualization Techniques
BERIMA ANDAM
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Many software development tasks revolve around features of a system, for example adding or removing a feature from a system. As a first step to performing these tasks, we need to know which source artifacts implement the feature(s).
Knowledge of these so-called feature traces are often stored in feature-source trace documentations. In reality however, feature-source trace documentations are often either outdated or entirely unavailable. The main reason why this is often the case is that feature implementing source artifacts change so quickly that it is hard to keep its documentation up-to-date.
In previous work, approaches aiming at reducing the amount of work needed to keep documentation up-to-date have been proposed. One such approach embeds feature-source trace documentation directly in the source artifacts using annotations [27]. It was found in the study that such annotations are cheap to create and maintain whiles its benefits far out-weight its costs as they naturally co-evolve with the artifacts they annotate.
In this thesis, we adapt this approach and propose tool support for creating, maintaining and exploiting such annotations. The goal of the approach is two-folds: first to reduce the amount of manual work required to create and maintain these annotations in order to encourage developers to use them. Secondly, to provide visualizations and metrics of the embedded documentation to enable developers to understand the documented system from its feature perspective.
Sometimes experts who can embed feature trace knowledge are not available. Therefore, in the second part of the thesis we propose an approach for recovering feature-source trace documentation from source artifacts. It is based on a machine learning approach to predict feature traces.
The approach was evaluated through a case study at ABB Corporate Research where it was tested on a product family. The results of a preliminary study with developers shows that they found the visualizations and metrics provided by the tool useful for comprehending the features in the system and its properties. The results of the experiments show that the proposed machine learning approach for feature location produces accurate feature trace predictions over time.

Keywords: Features, Feature Location, Software Metrics, Visualization, Tool Support, Machine Learning.

Acknowledgements

First, I would like to express my gratitude to my academic supervisors Prof. Thorsten Berger and Prof. Michel Chaudron, for their patience, support and guidance throughout the period of this thesis.
I would also like to thank the Software Engineering Group at ABB's Corporate Research Center in Ladenburg for hosting and supporting the research project. I would especially like to thank my industrial supervisor, Dr. Andreas Burger for his continuous support and for tirelessly organizing all the resources needed to undertake the research. I could not have asked for better mentors.
Finally, I would also like to thank my wife and best friend Sophia Amenyah and parents Mr. and Mrs. Andam for their love and support.

Berima Kweku Andam, Gothenburg, 12 2018

Published Parts of Thesis

Section 5, parts of Section 1 and Section 6 of this thesis document has been published in the paper [2] at the 11th International Workshop on Variability Modelling of Software-intensive Systems, held in Eindhoven, Netherlands on the 1 – 3 of February.

Contents

List of Figures

List of Tables

Acronyms

AST Abstract Syntax Tree. 16

BR Binary Relevance. 35, 36

CIDE Colored Integrated Development Environment. 16
CTSM Cosine Text Similarity Metrics. 31–33

DT Decision Tree. 35, 43–45, 52

FLA Feature Location Approaches. 29, 30
FLOrIDA Feature LOcatIon DAshboard. 3–5, 22–28, 32, 52
FPC Feature Presence Condition. 32

IDE Integrated Development Environment. 16
IR Information Retrieval. 15, 16, 29, 31, 32

kNN K-Nearest Neighbor. 35, 43, 46, 48

LoC Lines of Code. 16, 35, 38, 39, 41–43, 45, 46, 52

MLA Machine Learning Algorithm. 12, 13, 38, 46
MLP Multi-Layer Perceptron. 35

NAEFA Number of Already Existing Feature Annotations. xi, 31, 32, 42–46, 52
NoAu Number of Authors. 17, 23

SCLD Source Code Location Distance. xi, 31, 34, 42–44, 46
SCP Source Code Property. 38, 39, 42, 46
SPL Software Product Line. 7, 8
SVM Support Vector Machine. 35, 41, 43–46, 52

TSM Text Similarity Metrics. xi, 31, 42–44, 46, 52

WEKA Waikato Environment for Knowledge Analysis. 35, 44

1
Introduction

The notion of *feature* is commonly used when engineering software systems. A feature abstracts over concrete software artifacts, such as code, requirements or models. Developers use features for communicating and reasoning about a system, as well as keeping a comprehensive understanding of it. Using features is especially helpful when many variants of a system exist, as features provide an intuitive way of distinguishing individual variants [12, 9, 13].
Many software-engineering activities are centered around features [38], such as extending or removing a feature, propagating features across variants, or consolidating cloned features. All these activities require developers to understand the features that exist in a system, as well as their exact locations in the artifacts. Thus, explicitly recording and maintaining features and their locations can support many such feature-related activities [38].

1.1 Problem Statement

In most software systems however, features are not treated as first-level entities. The trace between features of the system and their implementing source artifacts are not clearly defined. In systems where features are treated as first level entities, and traces between them and their implementing artifacts are kept, it is still a difficult task to keep such trace documentation up-to-date.
The reason is that maintaining feature locations can be a daunting task. Implementing source artifacts change so quickly that externally kept feature location information may quickly become inaccurate when it is not continuously updated—a laborious and error-prone task [14]. Developers must choose between constantly updating documentation of already implemented features or implementing new feature request from customers (the main business). Usually the choice is to focus on implementing feature request from customers over updating documentation. As a consequence, up-to-date feature-source documentation often only exist in the minds of the experts of the system. When these experts are no longer available, the documentation is thus also not available.
The resulting missing trace between features and their implementing artifacts make software development tasks that require feature location knowledge rather unnecessarily difficult, time and resource intensive.
Several approaches have been proposed in previous work to support feature - source traceability documentation. A notable approach that shows significant prospects proposes embedding feature annotations in the source artifacts themselves, instead

of keeping them separately in external storage mediums such as documents or databases [27]. It has been shown that embedding feature trace documentation in the artifacts themselves can significantly reduce the amount of work needed to keep the documents up-to-date. Embedding feature traces prevents documented feature traces from getting out of date when source code is moved around, as the documentation is also moved along with it. Adding such annotations to artifacts is cheap, while the maintenance effort is low, as they naturally co-evolve with their artifacts (i.e. as opposed to externally kept feature locations), which significantly reduces manual updates [27]. Embedded annotations have also been shown as beneficial for engineering software with many variants [27, 38], especially when variants are not realized as a software product line [7, 18, 4] with an integrated platform, but using clone-and-own.

Problem 1: In traditional documentation approaches, knowledge of a software's feature implementation is kept in artifacts such as class or package diagrams. These artifacts typically store knowledge at different levels of abstraction. Thus, when a developer is interested in the details of the systems implementation on a certain level of abstraction, they would refer to the artifact corresponding to that level of abstraction. Compared to this, embedded annotations are on a fairly low level. After documenting using embedded annotations, finding code related to a feature can then be done by using simple tools such as grep [37] to search for the name of the feature in the source code. When working on small systems, the number of artifacts that may be returned may not be large, thus such tools may suffice. However, in large systems, the number of embedded annotations may be huge, and trying to understand a feature's implementation by investigating the returned source files may still require a lot of work. The cognitive effort required to understand a features implementation on this level might be quite high as the abstractions and simplifications embedded in traditional approaches are no longer there.

In this way, even though embedded annotations solve the problem of keeping documentation up-to-date, making it practically useful requires answering some questions which are currently open such as; What are the best approaches to exploiting knowledge embedded in annotations? How can we create aggregate views on several abstraction levels from embedded annotations? How can we calculate metrics about the features from annotations? Can embedded annotations provide knowledge that are typically stored in more traditional documentation? What types of documentation artifacts (i.e. views and metrics) can be extracted from them in that case? Are the views and metrics as useful for comprehending the system as traditional documentation? Do they suffice on their own or do they serve as a complimentary source to more traditional ones? etc.

To make embedded annotations useful for practical applications, these and other questions need to be investigated further.

Problem 2: A second problem is that, even though embedding feature annotations have been shown to significantly reduce the amount of work required to keep feature - source trace documentation updated (compared to traditional approaches), in large systems, the number of annotations that will be required to fully document features can be considerably large. Thus, a significant amount of work is still required to create and maintain even embedded documentation for such systems. A way to

reduce this effort required, could be to use automatic feature location approaches to find the feature locations in the first place and then tool support to then document them using embedded annotations.
A number of such feature location and documentation approaches have been proposed in previous work. These feature location approaches provide a useful starting point by finding relevant pieces of source code that may implement a feature of interest. However, these approaches have short-comings that prevent their adoption in practice [48]. Current feature location approaches can be divided into two broad groups; Static and Dynamic. Dynamic approaches are quite accurate but usually require input that are hard to get in real life. Static approaches on the other hand are often less accurate or are programming language dependent. Hybrid feature location approaches try to combine the strengths of dynamic and static approaches but often require substantial effort for their application, still leaving the majority of the work to the developers, a situation that can be laborious and expensive when working on large systems. Furthermore, current automatic feature location approaches identify feature implementing code on a relatively coarse-grained level; on the file or function/method level, although features are often more fine-grained, comprising just one or few lines of code [48].
In order to be feasible for use in real life situations, a feature location approach must provide an acceptable level of accuracy while using input that is readily available (or relatively inexpensive to come-by). They must also be as programming language independent as possible (so that they can be applicable for many system types). Finally, the granularity of the returned artifacts must be flexible as feature implementations usually span multiple levels of abstraction (It could be a line of code, a method or a whole package, or a mix of these). None of the currently existing feature location approaches meets this criterion. [48, 56]

1.2 Purpose of Thesis

The aim of this thesis is to investigate the hypothesis that: It is possible to create aggregate views from embedded annotations and that these aggregate views can reduce the cognitive effort required to understand feature locations documented as embedded annotations. Secondly, that a machine learning approach to feature location that relies on expert knowledge of feature locations documented using embedded annotations can be used to accurately recover feature location for undocumented parts of the same system.
The thesis contains several contributions:

1. An approach for generating some aggregate traditional software views from embedded annotations
2. A tool Feature LOcatIon DAshboard (FLOrIDA) that implements the approach
3. A machine learning approach for predicting new feature locations based on existing embedded annotations
4. An experiment that evaluates the performance of the proposed machine learning algorithm for feature location

The tool supported approach is expected to be useful for two main software development use cases involving features; as summarized in Figure 1.1. The primary one is to support continuous addition of traceability information (i.e., features and feature annotations) during development. The secondary—but sometimes necessary—use case is retroactive feature-location recovery of legacy software. Figure 1.1 summarizes these two main use cases. For both, a developer starts the tool and selects the root directory of the project's codebase (i.e. large, gray folder on the top-left of Figure 1.1) that is or should be annotated. It then builds an internal model containing the files in the project, the declared feature model, and then associates the files to the features based on the embedded annotations. Using this model, the tool FLOrIDA can then generates graphical views and metrics (i.e. arrow ❶ in Figure 1.1), which can support developers who observe (i.e. arrow ❿ in Figure 1.1) these for reasoning about the system and keeping an overview understanding of the features. In addition, using the proposed feature location based on existing annotations, FLOrIDA then can provide a feature-location recovery to suggest feature locations in legacy (i.e., not annotated) code.

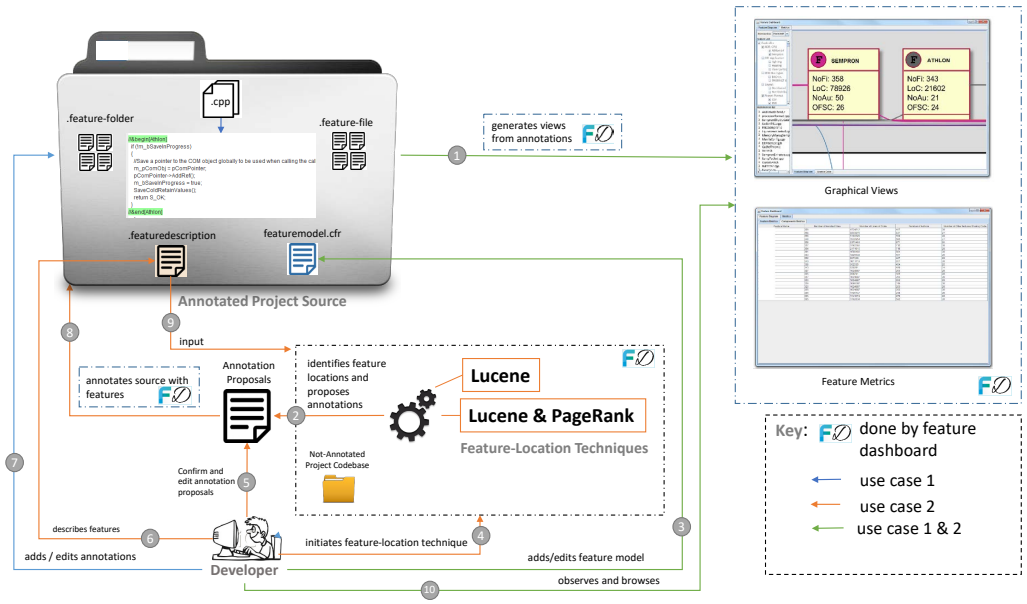


Figure 1.1: Annotating and displaying feature traces

To achieve these objectives, the thesis tries to answer specifically these research questions:

- **RQ1:** How can we exploit information documented in embedded annotations for source code comprehension?
  - **RQ1.1** What types of traditional source code views can we extract from embedded annotations
  - **RQ1.2** What types of metrics can we extract from embedded annotations?
- **RQ2:** How can we use machine learning to accurately predict feature locations in source code?
  - **RQ2.1**: What source code properties are best predictors of feature locations?

- **RQ2.2**: At what granularity of source code, is feature location using machine learning most accurate?
- **RQ2.3**: What machine learning algorithm(s) provide the most accurate predictions of feature locations?
- **RQ2.4**: How many example feature locations must already exist for the best configuration to give the best prediction?
- **RQ2.5**: How often must a machine be retrained to get good predictions?
- **RQ2.6**: How does the performance of the machine learning algorithm(s) differ when tested on a sample containing both annotated and unannotated code?

The rest of the thesis document is organized as follows: Section 2 describes work that this thesis builds on as well as technologies used which may be needed to understand the rest of the thesis. Section 3 describes previous work in the areas of feature location, concern visualization and how this work relates to them. Section 4 describes the approach for generating aggregate views and its implementations in the FLOrIDA tool. It also introduces the features of the FLOrIDA tool, how it is built and feedback from developers. Then section Section 5 describes the semi-automated hybrid approach to feature location, the setup of the experiment to evaluate these configurations, and the results of the experiments. Finally the results of the work is summarized in Section 6.

2

Background

2.1 Software Product Lines

Developing software from scratch is an expensive and error prone activity. One way to reduce the cost and improve the quality of software is to reuse already existing software assets [42]. Typically, a software system's evolution is driven by customer demands for new product features. An effective way of reducing time-to-market and to improve productivity is to reuse already existing assets. When a customer requests an already existing product, but with some addition or removal of features, a common way to build the requested products is to use the ad hoc re-use strategy, clone-and-own [42].

However, when the number of product variants grow, the time required to perform maintenance and evolution tasks on the system also grows exponentially. Eventually, a more systematic reuse and development strategy such as migrating the system to an Software Product Line (SPL) is needed to manage the complexity [42].

Motivated by the need to avoid this problem of overlaps and to encourage long-term systematic reuse of similar functionality, companies sometimes try to migrate these products to an SPL [8]. In an SPL, features from all the products are pooled together and using variability management mechanisms, and individual products are built by selecting specific features from the pool. This arrangement provides several benefits to the organization, such as the ability to meet customized customer needs faster and cheaper and to increase the product quality significantly over time as the feature components in the pool mature [8].

A set of software systems is considered as an SPL if the products share a set of common features, which are managed collectively in a single pool. The features in the pool are designed to meet a set of predefined needs of a certain domain [18]. According to Krueger [30], there are three paths to establishing an SPL; The first and most preferred path(i.e. the proactive approach) is to perform a complete analysis of the domain that the SPL is to be established for, then set in place variability management mechanisms and further, to design feature implementations to solve problems in the domain. Subsequently, single products are built for a specific customer by selecting features from the pool which correspond to the customer's needs. A second approach is to build an SPL from the beginning. Then build and modify new products separately for each new customer and then periodically integrate common occurring modifications into the SPL. Finally, a last approach is to build products independently without an SPL creation goal in mind, but only regenerating the separate products into an SPL when it becomes necessary to do

so. The last-mentioned approach is the most commonly used path even though it appears less intuitive. The reason is that, at start, most products are created to satisfy a customer's needs without an SPL creation intention in mind. As described earlier, new products are added to the product family as new customer needs emerge. A further motivation especially for using the clone-and-own methodology is that it is effective for quickly introducing new products to market before re-engineering is later considered.

2.2 Features and Feature Oriented Development

Many software-engineering activities besides migrating legacy software to an SPL are also centered around features [38], such as extending or removing a feature, propagating a feature across variants, or consolidating cloned features into an SPL [7, 18, 4].
It is common however, to find that such documentation is entirely unavailable or outdated. Therefore, accurate feature-source trace documentation is often only in the minds of experts (i.e. the developers who implemented these features). Thus, when these experts are no longer available, the documentation is also not available.

2.3 Feature Location Recovery

Several studies in the past have proposed approaches for recovering feature-source traces. These feature location approaches use different kinds of information from the system being analyzed to locate the source code of a feature. The approaches can be classified based on whether the source of the data used for location is gained by running the system being analyzed or not. There are three categories of feature location approaches according to this categorization; dynamic, static and hybrid feature location approaches [48, 20, 23].
Dynamic approaches identify features by collecting and analyzing dynamic data while executing the feature to be located. They are often very precise as they collect precise information about elements of the program that are activated as a feature is executed [6, 57, 58, 21, 54]. Static feature location approaches on the other hand extract and analyze static information from the systems source code or related documentation such as dependencies between program elements, or relatedness of vocabulary used in the documents [1, 14, 45, 49, 59]. A third group of approaches; Hybrid feature location approaches combine both dynamic and static data for locating features [43, 22].
The aim of all feature location approaches is to find only relevant feature traces (i.e. high precision) while trying to recover all relevant traces (i.e. high recall) [48]. According to Rubin et al, [48] neither dynamic nor static approaches are able to achieve both, as each category of approaches has its limitations.
Dynamic feature location approaches are able to achieve high precision as they are conservative in nature [6]. The reason is that, execution traces can contain a lot of noise. Therefore, multiple traces are needed to identify feature relevant code. The traces are obtained by running two sets of test cases for each feature to be

identified. One set which executes the feature of interest exclusively and another set which executes all other features but the feature of interest. This is something that is often unavailable in real life and expensive to obtain. Dynamic approaches however often have low recall as the quality of test cases selected by the user directly affects the quality of outputs given by the approaches. According to Simon et al, [50], "selecting test cases that exercise too much or too little of the system can cause problems". Static feature location approaches on the other hand, often use input which are often readily available or inexpensive to obtain. They achieve a high level of recall, but suffer a low level of precision. Hybrid feature location approaches try to combine the strengths of these approaches while reducing its draw backs.
Some of the approaches (i.e. dynamic location approaches) require input that are hard to acquire in real industrial cases [20]. Other approaches (i.e. Static approaches) require input that are readily available, but on the other hand often require a lot of time investment from experts to refine their outputs, as they may return a lot of false positives [48]. Furthermore, most of the approaches do not make use of expert knowledge of features and their location in the recovery process but rather attempt automatic location which so far does not work. Feature locations returned by expert driven approaches may also be defective even for experts of the system. This happens because, typically, a lot of time would have passed between developing the feature and when feature location needs to be done [48, 20].
We believe that features and their locations should be recorded early, when they are implemented, to avoid high costs for retroactively identifying features and their locations. Instead of migrating clone-based products in a costly and risky process, we strive to establish a truly incremental product-line adoption process—a.k.a., the virtual-platform approach [4].This process should support using only a subset of product-line concepts (e.g., features, traceability, configurator, pre-processor or build system), allowing a truly incremental adoption of a product line, where an incremental investment (e.g., introducing features) provides an incremental benefit (e.g., keeping an overview understanding of features across cloned products).

2.4 Embedded Feature Annotations

To record feature locations, two storage strategies can be used. Locations can be stored outside the source artifacts as is done traditionally in external storage, such as in a traceability database or in plain documents. Alternatively, they can be stored directly within the artifacts with embedded annotations. Typically, feature-source trace knowledge is kept in documents or in trace databases. However, external storage of feature locations is brittle and requires continuous effort to maintain. To make feature-trace documentation more robust, several approaches have been proposed in past research. A notable approach by Wenbin et al [27], proposes embedding feature traces documentation directly in software artifacts implementing the features. The advantage of this approach over external approaches is that embedded documentation eliminates one of the reasons why documentation becomes outdated. Namely, Embedding the feature-location information facilitates keeping it updated when the codebase evolves. The study also showed that many of the annotations naturally co-evolved with the artifacts (e.g., when code is moved), not requiring active an-

notation maintenance. Feature-source trace is still intact even when source code is moved from one location in the source structure to another, which would not be the case for externally kept trace knowledge.
A case study shows that the cost of creating and maintaining the annotations is low to negligible, while the benefit for feature-related activities is substantial [27]. Specifically, 18 % of the recorded feature locations as annotations saved 90 % of feature-location costs needed for typical feature-related activities. sBesides this, embedding feature-source traces gives the advantage that, similar copied features between two variants can later be located and merged when needed.
Exploitation of embedded annotations however remains untapped. Embedded annotations in industrial scaled systems can be significantly large. The sheer number of annotated artifacts for a single feature can be overwhelming for a developer who wants to understand the source code, if they are retrieved in plain format. Besides this, the artifacts in raw format alone does not provide a lot of information about the feature properties that may be useful for reasoning about it, or for performing feature oriented tasks.

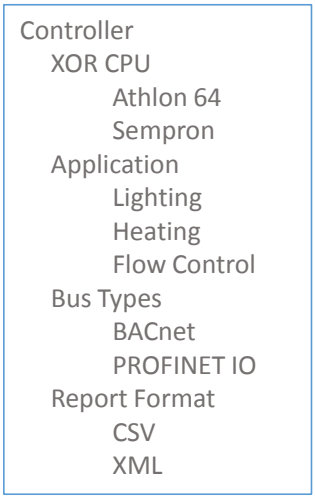


Figure 2.1: Example feature model in Clafer syntax

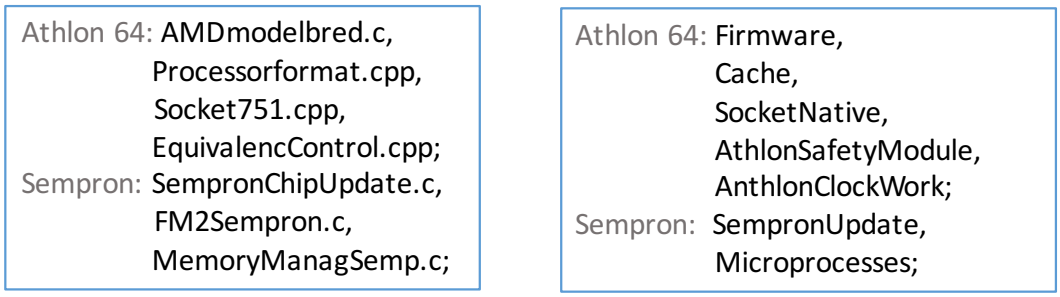
2.4.1 Feature Model

Documenting feature locations using embedded annotations can be done using the Clafer syntax [3]. To begin the documentation process, the developer first creates a simple, textual feature model, that lists out the features that the system implements (i.e. as shown in Figure 2.1 and then saves it as featuremodel.cfr. This file is then stored at the project's root folder. Features are added to the model: one feature per line, with the feature hierarchy being expressed by indentation—a Clafer convention. The top feature should be the name of the project. Any simple text editor suffices to edit the model.
Although the Clafer language is much richer, to document feature traces, only using the simple hierarchical feature suffices. Yet, developers could also add domain-specific feature dependencies into the model (e.g., feature groups, such as the XOR group CPU in Figure 2.1), which could be exploited later when propagating features across cloned variants or merging variants.

To relate parts of an artifact to features, embedded annotations (i.e. escaped as comments) are used. To relate a whole artifact (e.g., source file) or a whole folder to one or more features, textual mapping files are added to the folder structure. These traces relate artifacts to features, which were earlier defined in the feature model.

2.4.2 File and Folder Annotations

To associate whole source artifacts with features in the model, the developer creates special mapping files. Specifically, to associate files in a folder with a feature, a simple text file .feature-file has to be created within the folder. Each line in the file contains a mapping between one or multiple features and a file using the syntax `featureName: fileName(,fileName)*`, as shown in Figure 2.2a. Mapping whole folders to features is similar. The developer creates a .feature-folder file in the parent folder. Each line in the file maps a feature to one or multiple folders using the syntax `featureName: folderName(,folderName)*`, as shown in Figure 2.2b.



(a) .feature-files **(b)** .feature-folders

Figure 2.2: Examples of mapping features to files and folders

2.4.3 In-File Annotations

Annotating parts of a file with features is slightly different. To annotate multiple lines, the developer simply surrounds them with a beginning tag `//&begin[featureName]` and an ending tag `//&end[featureName]`, see line 4 and 10 in Listing 2.1. If only one line should be associated to a feature, the developer can use a single line annotation with the syntax: `//&line[featureName]` on top of the line of code as shown for example in line 1 of Listing 2.1. Note that in our examples, the comments (`//`) are C/C++ specific. For other languages, the tags should be used within the respective commenting characters.

2.4.4 Feature References for Ambiguous Feature Names

In feature annotations, features are referenced using their least-partially-qualified (LPQ) names. These are usually just the feature names if they are unique within the feature model (i.e. which is the case for all features in our example). However, if a name is not unique, then it must be qualified partially—just enough to make the reference unique.

```
1   ////&line[System Monitor]
2    void HEAPUTILModuleOp(tModOp ModOp)
3    {
4      //&begin[State Visualizer]
5      if (ModOp == CloseModOp)
6      {
7        size = file.tellg();
8        memblock = new char [size];
9        file.seekg (0, ios::beg);
10       file.read (memblock, size);
11       file.close();
12
13      cout << "the entire file content is in memory";
14
15      delete[] memblock;
16     }
17     //&end[State Visualizer]
18   }
19  ////&line[Report Maker]
```

Listing 2.1: Annotated source code

For example, the feature `Sempron` has a unique name in the model and can be simply referred to by its name. If this name occurred twice in the model (e.g., if another feature also named `Sempron` occurred under the feature `Application`), then both must be qualified to uniquely identify them from each. Using their LPQ name, features could be referenced as `application::sempron` and `cpu::sempron`. While fully qualified names could also be used (e.g., `::controller::cpu::sempron`), they are much longer and more brittle as compared to the LPQ names when the feature model evolves.

2.5 Machine Learning

Machine learning algorithms identify patterns in large amounts of data describing a problem that is presented to it, and then uses the learned pattern to detect future occurrences in similar data. They are often used to improve the performance of a program on a task by learning from past data.

2.5.1 Concepts

In this study, Machine Learning Algorithm (MLA) will be used to map *instances* of data (i.e. source code) to different *classes* (i.e. features of the subject system). Each piece of source code is described by a set of extracted source code properties and associated source code property values. A source code property is a derived descriptive statistic, calculated from the source code it represents - for example the cosine similarity between the words used in the source code and the words used in a sample already known source code implementing the feature of interest. All pieces of source code are described with the same set of source code properties, but each source code exhibits different source code property values, depending on which feature it is implementing. There are two alternative groups of MLAs that can be used for classifying source code to features, *supervised* or *unsupervised* groups of MLAs. Unsupervised (clustering) algorithms are used to group source code into different clusters (i.e. in our case features). The number of clusters to be created

are not know before hand, the clustering algorithm groups the source code based on the similarities in the source code's property values. It decides the number and statistical nature of the clusters.
The features of a system are however typically known before hand (as many software engineering activities revolve around them). The supervised group of algorithms are thus more suitable.

2.5.2 Multilable classification

Traditional single label classification deals with assigning a label to an instance to be classified. However, as pieces of source code (which are the instances in our case) can belong to multiple features (labels) at the same time, a variation of classifiers known as multilabel classifiers are more suitable for the task. Several multilabel classification algorithms are available in literature. These can be grouped into two main categories, problem transformation algorithms and algorithm adaptation. Problem transformation algorithms, transform a multilabel tasks into one or more single-label classification, regression or ranking tasks. Algorithm adaptation methods on the other hand extend specific single-label learning algorithms to handle a multi-label learning problem [55].

2.5.3 Feature reduction

Several descriptive source code properties can be calculated from each piece of code to be classified, which can be used for training a classifier. Examples are size of the source code, author, date of creation, many different text similarity metrics, etc.
In practice, the best subset of these properties must be deduced and used, as it is not feasible to use the maximum set of possible source code properties. This is because it is computationally intensive to create and use all possible properties and will most likely result in a slow performing MLA. Besides this, the representativeness of the set of source code properties can have a great impact on the predictive performance of a MLA. It is not always the best option to train a classier using the maximum set of source code properties attainable. The reason is that redundant source code properties can negatively affect the predictive performance of a classifier.
The process of selecting the number and types of features for training the machine can be done manually through trial and error. However, this can be tedious if the number of features to consider are large. Some automatic approaches are available for automatic feature selection. These are grouped into filter or wrapper selection models. Filter algorithms calculate a metric to rate properties and then to select a subset of properties based on this rating. Wrapper methods on the other hand evaluate the features in combination with the algorithm to be used algorithms to be used. It results in a tailored set of properties for each algorithm [19].

3

Related Work

3.1 Feature Location

Feature location has been studied extensively. However, there are still problems with current feature location approaches that prevent their use in practice. As mentioned in previous chapters, feature location approaches can be grouped into two main categories; static and dynamic. Static approaches are the oldest and most intuitive approach used by programmers to find feature locations in source code. An example is the use of pattern matching supported by popular tools like grep, fgrep, set, awk that perform pattern matching on strings. Although these approaches are very intuitive, fast to use and allow searching on a low level of granularity, they miss out on feature location knowledge stored in structural dependencies between source code elements [39].

To address this problem, several studies propose approaches for searching the source code, where the system is represented as a graph. Entities in the system are represented as nodes in the graph while relationships among these entities are represented as edges between them. Concrete examples of tools based on this approach are FEAT[46], Ripples [17], Scan [39] and Rigi [36]. FEAT uses the built graph to facilitate the separation of different feature code from one another whiles Scan uses the information to support the user to find related source code during incremental change [44]. The advantage that graph-based approaches provide over search based approaches such as grep, is their ability to retain structural dependency information among program elements as additional input for feature location. Examples of structural information that may be useful for feature location are for example data-flow, inheritance, inclusion and control flow relationships. This information is vital during the stages of searching and analyses of relevant feature locations. According to Singer [51], during software comprehension, software engineers use this relationship information to traverse the graph for the location of concepts. We also believe that structural information is important for locating features. Thus, similar to the above-mentioned approaches we also analyze the structural (i.e. in our case location based) relationships between known locations and a source code to be classified and use this information for training the classifier.

Another group of feature location approaches, the Information Retrieval (IR) based approaches rely on making meanings out of semantic information embedded in the comments and identifier names used in the source code of the system and surrounding documentation. IR based feature location approaches differ in a number of ways including how they store intermediate results, their pre-processing needs and the

granularity of source code they return as results of the feature location process [34]. IR systems are widely used in web search engines, libraries etc. where are used for storing, managing and retrieving unstructured data. Some IR based feature location approaches use identifiers in source codes alone for the feature location task [5]. Other approaches use the text in source artifacts such as source code comments and external documentation surrounding the source code [35]. Some IR feature location approaches identify source artifacts on the function or on the class level while others work on a component level [5, 35]. Our approach, just as with the approaches mentioned above also makes use of embedded semantic knowledge scored in identifier names in the source code. However, unlike other approaches such as that of Marcus et al [35] who also make use of source code comment as input for feature location, the approach proposed in this thesis does not use source code comments as part of the identification process . As another difference, the approach presented in this work identifies source code on the Lines of Code (LoC) granularity level as opposed to the method, class or component level.
This study investigates the possibility of combining the strengths of the approaches mentioned above whiles at the same time improving accuracy. The approach uses easy to find static and IR-based metrics to encode expert knowledge and then automate feature location thereafter.
As of the time of the writing of this thesis, no approaches have been identified that have tried to apply the machine learning to the task of predicting feature locations. However, similarly studies where machine learning has been applied to a number of related tasks such as requirement - source tracing exist [52].

3.2 Product-Line Engineering Tools

There are many available tools that supports different sets of phases of Feature-Oriented Software Development (FOSD) and Feature-Oriented Product-Line Development (FOPLD). FeatureIDE [29] for example, is an Eclipse based Integrated Development Environment (IDE) for feature oriented development that provides tool support for all phases of FOSD, namely domain analysis, requirements analysis, domain implementation, and software generation.
Pure::variants and Gears are commercial tools that also provide similar features for managing different phases of FOSD and FOPLD. They both contain tools for mapping features to software assets so that customized software system can be generated given a selection of these features.
Another available tool is Colored Integrated Development Environment (CIDE) [28], an open-source, IDE that helps developers to select specific pieces of code that implement optional features in legacy Java software systems. Each optional feature is assigned a color which is then used to color the pieces of code that are selected by the developer as implementing that feature. Behind the scenes CIDE uses this information to select parts of the Abstract Syntax Tree (AST) representation of the source code to help compile different products from these colored sources.
All the tools mentioned above are full blown IDE (i.e. heavy weight tools) that provide tools for creating variability in the target system. They are tailored for the purpose of creating variability in the target system. Annotation support is pro-

vided as way of distinguishing implementations of optional features as opposed to distinguishing implementations of all types of features. They focus more on providing support for the actual development of variable assets. Thus, their support for annotating and visualizing feature traces is oriented towards this purpose. Annotating feature implementations in source code is also usually done manually by the developer through the tool.
Unlike these however, FLOrIDA is designed as a lightweight tool whose purpose is to support feature-source traceability as a first step towards establishing a software product line (i.e., before all other, later stage SPL infrastructure, such as a configurator, preprocessor or build system etc. which these other tools consist of are needed) or for gaining a feature perspective of a system while performing purely feature oriented activities such as adding, removing or extending of features.
FLOrIDA provides complementary views that not only shows views as defined in the annotations but also different levels of abstract views and metrics on this relationship (more details about the views are provided in Section 4.3). In contrast to FeatureIDE, FLOrIDA, as explained, also incorporates approaches for semi-automatic location of features in legacy systems. Beyond this, FLOrIDA provides proactive support for documenting newly added source code with features.
Pleuss et al. [41] interactively visualize variability expressed in feature models of a product line. Similar to us, they present several abstracted views and filters of the features (i.e. which are configuration options) in the model, such as to illustrate cross-tree constraints, and to present consequences (i.e. with explanations) of selecting particular features.

3.3 Concern and Topic Visualization

Concerns and topics can be seen as similar, if not more general, concepts. Concern location has been studied intensively, and various concern visualization approaches exist specially to support program understanding and feature location.
FEAT [47] provides the developer a mechanism for creating structural views of related program elements by adding them to a concern graph. Through a querying mechanism, the user can then associate the graph elements to pieces of source code.
TraceGraph [32] incorporates a simple visualization of program traces which allows changes in the program execution traces to be easily identified. The tool aims at supporting concept location tasks prior to feature oriented development tasks such as bug-fixing or feature extension with particular focus on long running interactive software. Furthermore, the notion of topics is typically used to characterize developer discussions or comments. Likewise, topic visualization approaches exist. For instance, Izquierdo et al. [26, 47] provide graphical visualizations and metrics for Github issues annotated with labels representing issue topics. Features could be seen as topics. Their visualizations are network diagrams illustrating the label usage, label timelines, and user involvements. Our metrics and views also cover some of their information, such as the Number of Authors (NoAu); yet, we provide additional metrics, such as feature scattering and tangling degrees.

4

Abstract View Generation Approach and Implementation (FLOrIDA)

4.1 View Generation Approach

As described in Section 1, raw feature annotations (just like source code) contain too much details and may be overwhelming for a developer who wants to get an overview understanding of a systems implementation. Traditionally, high level diagrams such as a package diagram may provide such a perspective. In this section, we propose an approach for generating abstract views but in this case of the feature-to-source relationship. To generate these views from embedded annotations, the following algorithms are proposed.

Processing files in project

To start off, all source files and annotations in the project must be read and annotations associated with the source artifacts to which they correspond. The following procedure can be used for this purpose:

```
1: L ← Line in File
2: F ← File in Folder
3: Fo ← Folder in project
4: FoAnF ← Folder Annotation File
5: FiAnF ← File Annotation File
6: BegEndAn ← Begin-End Annotation
7: LiAn ← Line Annotation
8: for Fo ∈ Project do
9:   if exists (FaAnF || FiAnf) then
10:    goto processAnnotation.
11:  end if
12:  for F ∈ Fo do
13:    create a file-node for F.
14:    for L ∈ F do
15:      create a line-node for L.
16:      if L contains a LiAn or BegEndAn then
17:        add annotation to list of project LiAns/BegEndAn respectively.
```

```
18:             goto processLineAnnotations.
19:           end if
20:       end for
21:     end for
22: end for
```

Processing Folder Annotations

The following algorithm can then be used to process feature-folder annotations and thus to associate the features found in them to the folder that they annotate:

```
1: L ← Line in File
2: F ← File in Folder
3: Feat ← Feature in project
4: Fo ← Folder in project
5: FoAnF ← Folder Annotation File
6: PrFoAnFs ← All Project Folder Annotation Files
7: add FoAnF to list of project PrFaAnFs
8: for Feat ∈ FoAnF do
9:    add Fo to Feat list of folders.
10:   add Feat to Fo list of features.
11: end for
```

Processing file Annotations

For each found feature-file annotation, the following algorithm can be used to process and associate the features that it mentions to the relevant files:

```
1: L ← Line in File
2: F ← File in Folder
3: Feat ← Feature in project
4: Fo ← Folder in project
5: FiAnF ← Folder Annotation File
6: PrFiAnFs ← All Project Folder Annotation Files
7: add FiAnF to list of project PrFiAnFs
8: for Feat ∈ FiAnF do
9:    add F to Feat list of files.
10:   add Feat to F list of features.
11: end for
```

Processing Line Annotations

Further more, to process file annotations the following algorithm can be used:

```
1: LiAn ← Line Annotation
2: Feat ← Feature in project
3: PrLiAns ← All Project Line Annotation Files
4: for Feat ∈ FiAnF do
```

```
    add L to Feat list of Lines.
    add Feat to L list of features.
end for
add LiAn to project PrLiAns
```

Processing Begin-End Annotations

Finally, begin-end annotations can be processed using this algorithm:

```
LiAn ← Line Annotation
Feat ← Feature in project
BegEndAn ← Begin-End Annotation
PrBegEndAns ← All Project Line Annotation Files
for Feat ∈ BegEndAn do
    add L to Feat list of Lines.
    add Feat to L list of features.
end for
Add BegEnd annotation to project BegEnd Annotations
```

Feature-Folder abstract view

After reading into memory the project files and associating them with found embedded annotations, abstract views can be created from them using the following different algorithms. For example, the feature-folder view can be created using the following algorithm.

```
Fo ← Folder in project
Feat ← Feature
for selected Feat do
    create a feature-view-node for feature.
    for Fo ∈ Project do
        if Feat in Fo contains selected Feat then
            create folder-view-node for folder
            draw an edge between folder-view-node and feature-view-node
        end if
    end for
end for
```

Feature-File abstract view

Again to answer the question what files in the project implement a feature, a corresponding view can be generated using the the algorithm:

```
F ← File in Project
Feat ← Feature
for selected Feat do
    create a feature-view-node for feature.
    for F ∈ Project do
        if Feat in F contains selected Feat then
```

```
7:          create file-view-node for F
8:          draw an edge between file-view-node and feature-view-node
9:        end if
10:     end for
11: end for
```

Metrics Generation Algorithms

Embedded annotations also encapsulate knowledge about the relationship between the features of the system and their implementing source code. To further improve user understanding of the properties of a feature's implementation, a quantitative measure of this relationship is calculated using the well known metrics described in Table Figure 4.1 below.

Table 4.1: Feature, folder, and project metrics

Metric	Description
	Feature Metrics
SD	Scattering Degree: total number of all annotations directly referencing the feature (i.e., in-file, folder, and file annotations referencing it)
NoFiA	Number of File Annotations: total number of file annotations directly referencing the feature
NoFoA	Number of Folder Annotations: total number of folder annotations directly referencing the feature.
TD	Tangling Degree: number of other features that share the same artifacts (or parts of such) with the feature. Two features share (parts of) artifacts when the latter is annotated with both features.
LoFC	Lines of Feature Code: lines of code *belonging* to artifacts, either directly annotated, or indirectly (when a folder is annotated, all descendants are taken into account)
ND	Nesting depths of annotations: Maximum (**MaxND**), Minimum (**MinND**), and Average (**AvgND**) nesting depth the annotations directly referencing the feature. The project's root folder has depth 0 (and so has any file contained in it). Each sub-folder increases the depth by one, a file inherits the depth of its containing folder. The depth of a (top-level, i.e., non-nested) in-file annotation is the depth of the file increased by one. Since in-file annotations can be nested, each nesting increases the depth by one. All nesting-depth metrics are calculated relative to the project root folder.
NoAu	Number of Authors who contributed to a feature's artifact. Author information is automatically extracted from author tags (format: "Author: firstname lastname") in comments wrapped by "/**" and "*/" in the source code if they exist.
	Folder Metrics
NoF	Number of Features: total number of features directly referenced in annotations (folder, file, in-file) of the folder and any of its descendants
LoFC	Lines of Folder Code: total lines in any descendant file of the folder
NoFi	Number of Files: number of all descendant files of the folder
	Project Metrics
NoF	Number of features in project
Total LoFC	Total Lines of Feature Code: sum of LoFC (all features)
Avg. LoFC	Average Feature Lines of Code: sum of LoFC (all features) / NoF
Avg. ND	Average Feature Nesting Depth: sum of ND (all features) / NoF
Avg. SD	Average Feature Scattering Degree: sum of SD (all features) / NoF

4.2 Implementation

The tool FLOrIDA implements the above mentioned algorithms. It is designed as a lightweight tool that comes within one binary without requiring any installation procedure. It is implemented as a stand-alone Java program, which allows running it on any Java-supported platform. Finally, the embedded annotations are independent of the target programming language, so any kind of artifact can be annotated as belonging to a feature.

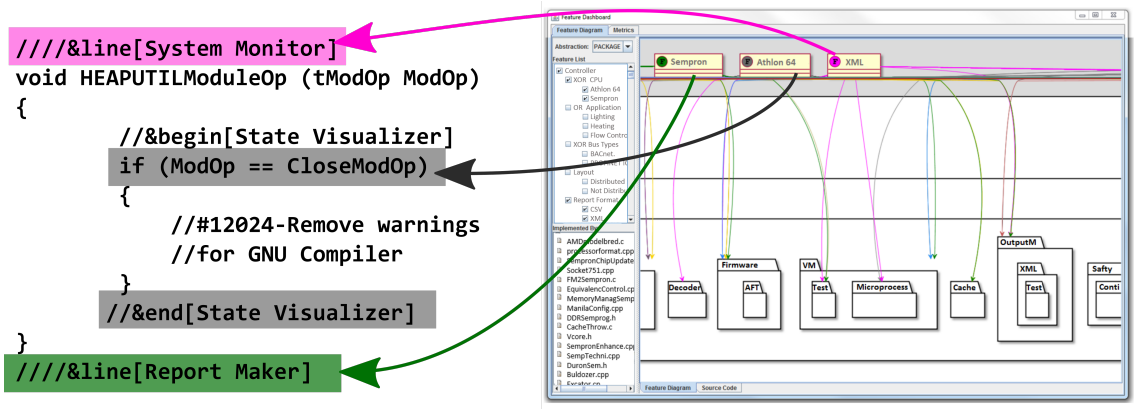


Figure 4.1: Demarcation of feature locations in source code

The implementation of FLOrIDA is divided into four parts. The first part, the annotations extractor, recursively traverses files and folders of the codebase to gather embedded annotations. The parser creates an internal model of the project that contains nodes for the source artifacts (i.e. files and folders) and the features, which are associated based on the annotations. For each file, it also checks for embedded (i.e. in-file) annotations (e.g., `//&line[System Monitor]`). If an annotation is found, the file is associated with the respective feature(s), and the specific lines are stored in the internal model.
The second part, the metrics calculator, derives metrics based on the feature model and the annotations. No programming-language- or project-specific information is taken into account, except for the metric **NoAu** (i.e. see Table 4.1), which is extracted from author information embedded in comments.
The third part, the visualization module, is responsible for rendering the graphical views. The graphical visualizations are done with the help of PLANTUML[1], which internally uses the DOT [24] graphics library.
The fourth part, the feature-location module, relies on the algorithms PageRank and Vector Space Model implemented by the Lucene[2] search engine. These algorithms have been used in previous work for feature location [10].

4.3 Feature-Oriented Views

We now present the graphical views and metrics that FLOrIDA provides for developers.

4.3.1 Browse Feature View

A developer can select a feature to show the artifacts annotated with that feature. One can then select a source file in order to explore and to analyze the source code. Any embedded annotations within the source file are highlighted with the assigned color of the feature. This option helps to clearly demarcate to the user where the

[1] `http://plantuml.sourceforge.net/index.html`
[2] `https://lucene.apache.org`

t

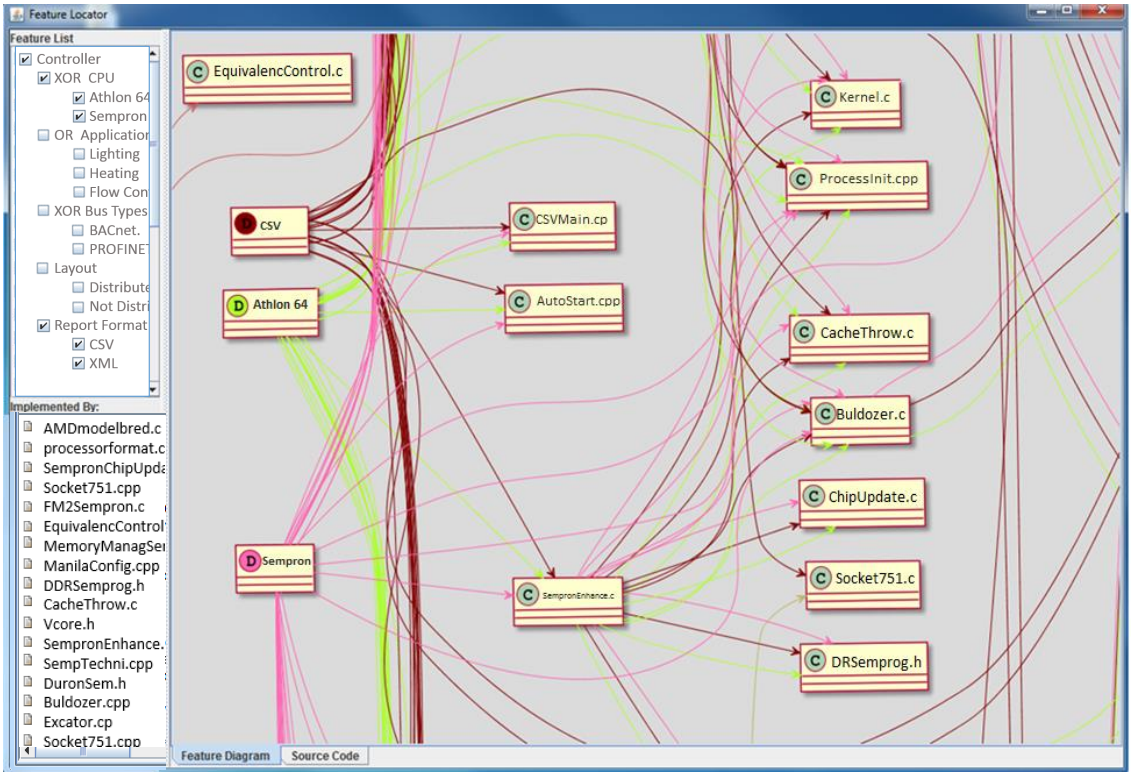


Figure 4.2: Feature-file trace view

annotated implementation of a feature begins and ends. Figure 4.1 shows such a demarcation.

4.3.2 Trace Views

A developer can select one or multiple features from FLOrIDA's displayed feature tree. Then, it displays the implementing files of the feature(s) and a graphic visualization of this relationship. When more than one feature is selected, the visualization also shows the interaction between features in terms of shared implementing source artifacts. The visualizations can be on the file level or folder level, as shown in Figures 4.2 and 4.3.
FLOrIDA's file-level-visualization (Figure 4.2) shows the relationship between a selected feature(s) and its implementing artifacts. When a feature is selected by a user, a feature node is created by FLOrIDA for that feature, and a color is assigned. For each of its implementing artifacts, a source node is created and an edge between the source and the feature is drawn, using the same color as assigned to the feature. The colors help to highlight the interactions between features when they are implemented by the identical source artifacts.
Developers can also explore a feature implementation on the folder level (Figure 4.3). When such a request is made, FLOrIDA creates a feature node just as in the case of the file-level visualization. Thereafter, for every folder that is annotated as implementing the selected feature(s) (in .feature-folder), or that contains a file implementing the feature (in .feature-files), a folder node is created. Then just as in the

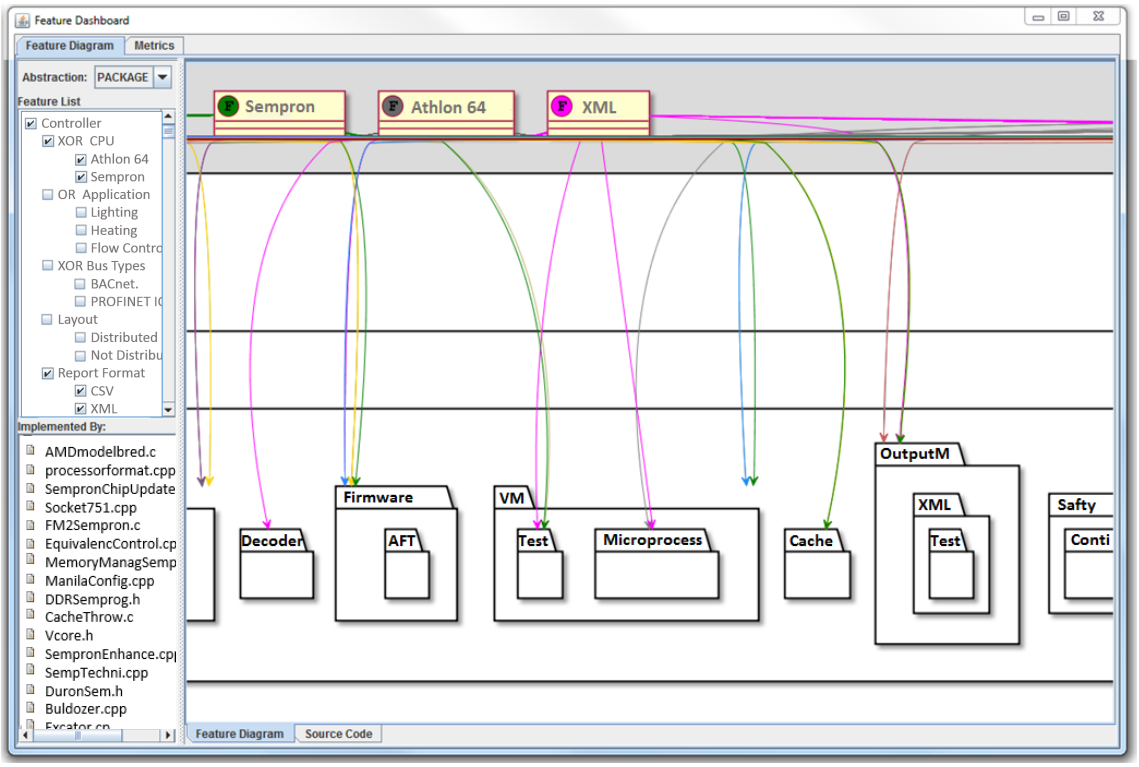


Figure 4.3: Feature-folder trace view

case of the file-level visualization, an edge is created between each feature and its associated folders.
In the case of in-file annotations (i.e., annotations on a lines of code level), FLOrIDA creates an in-file annotation node for the annotated code. An edge is then drawn from the feature to the created node using the feature's assigned color (Figure 4.4).

4.3.3 Metrics Views

To further enhance the understanding of feature's properties, feature, folder and project metrics are also provided. All currently supported metrics are provided in Table 4.1.
A user can view metrics related to a feature(s) of interest. The user does this by selecting the said feature(s) and then selecting the metrics tab. FLOrIDA then displays several metrics describing the feature and its implementing artifacts and their relationship.
Feature metrics describe each feature's relationship with its implementing artifacts, as shown in Figure 4.6. Some of the metrics are well-known feature-related metrics, such as those provided by Liebig et al. [31] and Berger et al. [11]. We use the established terms, but even though we write "code" as in LoFC (lines of feature code), we actually count the related lines in any kind of non-binary artifact. Feature metrics are also shown directly on each feature node in the trace views, as shown in Figure 4.6.
Finally, the folder metrics describe each folder and its relationship to the features in the system, and the project metrics provide some aggregate numbers about all

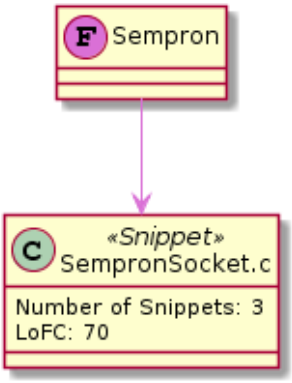
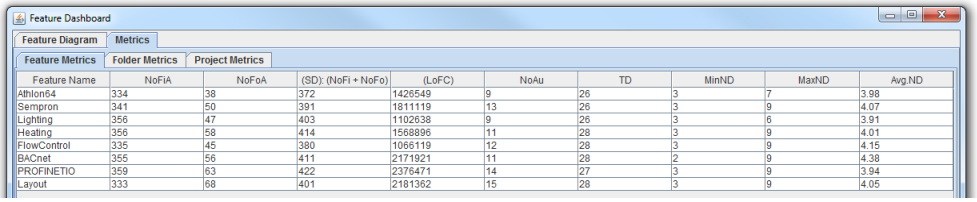


Figure 4.4: Visualization of an in-file annotation



Feature Name	NoFiA	NoFoA	(SD): (NoFi + NoFo)	(LoFC)	NoAu	TD	MinND	MaxND	Avg.ND
Athlon64	334	38	372	1426549	9	26	3	7	3.98
Sempron	341	50	391	1811119	13	26	3	9	4.07
Lighting	356	47	403	1102638	9	26	3	6	3.91
Heating	356	58	414	1568896	11	28	3	9	4.01
FlowControl	335	45	380	1066119	12	28	3	9	4.15
BACnet	355	56	411	2171921	11	28	2	9	4.38
PROFINETIO	359	63	422	2376471	14	27	3	9	3.94
Layout	333	68	401	2181362	15	28	3	9	4.05

Figure 4.5: Feature metrics view

features that exist in the whole project.

4.3.4 Feature-Location Recovery

Feature locations can also be retroactively recovered using FLOrIDA's built-in feature-location approach. To use this functionality, the developer has to, besides defining a feature model (featuremodel.cfr) of the system, provide a description of each feature defined in the feature model. To do this, the developer must create a description file .feature-description in the project's root folder. Inside this file, a description for each feature is specified using the syntax *featureName*∼*featureDescription*. When FLOrIDA's automated feature-location option is selected, it will process the project, locate, and automatically create annotations in the code-base. To distinguish them from manually created ones, a flag [auto] is appended to each line, for instance `featureName:fileName [auto]`.
Two choices of feature-location algorithms are provided from which the developer can select: Lucene and Lucene combined with PageRank. The choice allows exploring the accuracy of two different algorithms.
The Lucene algorithm uses the provided feature description information to automatically retrieve the most semantically similar source artifacts. Lucene implements the Vector Space Model algorithm used to calculate similarity of a body of text to another. Every document (i.e. in our case files; support for the method level is planned) is represented as a vector where the contents are the words in the document. Similarity between the feature's description and each document is then calculated by comparing how many unique words in the feature's description appear in the document. Unique words are obtained by using the Term Frequency/Inverse Document Frequency algorithm. Words that appear in all documents are weighted less than words that appear in a few documents. A code-file's similarity to a feature is, thus,

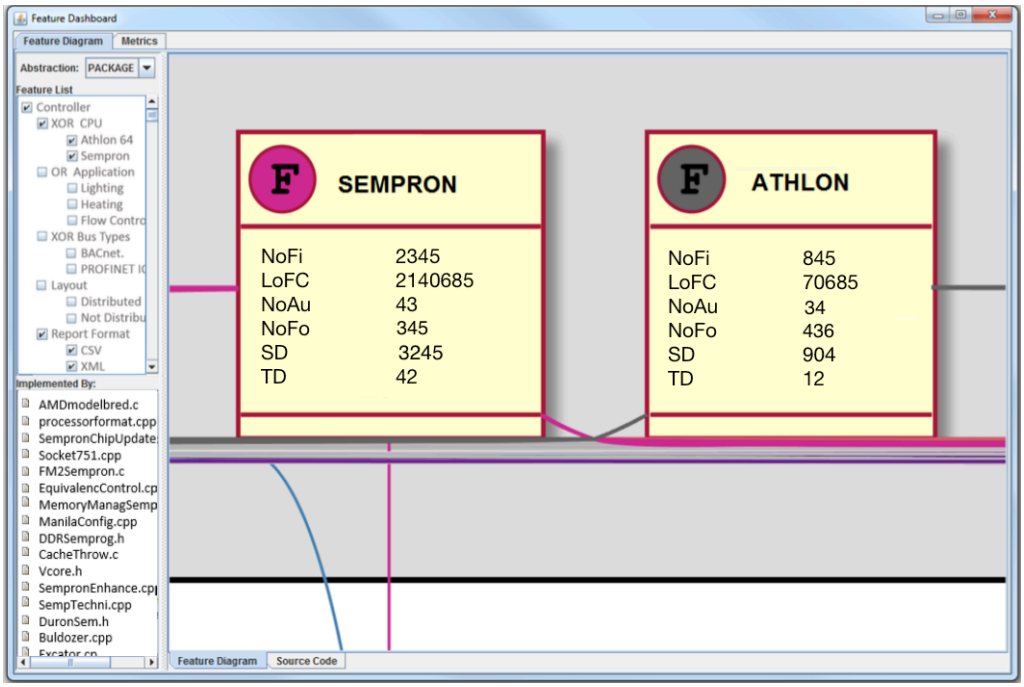


Figure 4.6: Metrics shown directly in a trace view

the combined score of each unique word in the code file that also appears in the features description.
The Lucene with PageRank algorithm first uses Lucene to calculate each file's similarity with the feature of interest, but then further refines this ranked list of similar artifacts using PageRank. The latter assesses the originality or importance of a document by calculating an importance metric based on how many documents reference a document against how many documents the document itself references. A higher score is given to documents that are referenced by others, but that do not reference others.
The developer can subsequently modify the annotations done by FLOrIDA by removing some or adding additional annotations. If a developer accepts one of the suggestions, FLOrIDA then automatically annotates the newly added code to reduce the effort required by the developer.

4.4 Evaluation and Feedback

To answer the question of the scalablity of the approach when applied on real cases preliminary evaluation were conducted on industrial automation system with 3.2 MLOC.
As there were no existing feature models of the system, one was created from analyzing existing user manuals, sales brochures, and system documentation. A total of about 82 features was extracted from the documentation. Feature descriptions were then written in natural language for each feature.
Even though the system is considerably large, it took only about four minutes to run the feature location algorithm Lucene+PageRank and to annotate the source with the proposed annotations. Then, it took another 25 seconds to extract all 6110

embedded annotations and to generate the views and calculate the metrics.
We also conducted an interview with two experts who participated in the development of the system. The first expert, currently an architect, has worked on the system as a developer since its inception (decades ago). The second expert, also an architect, has worked on the system for more than ten years. They expressed their opinions about the tool and how it is suited for the particular system after a demonstration of 1.5h.
The experts were positive about the robustness that the embedded annotation approach could give to the documentation of feature locations, which means that a large amount of documentation time could be saved as opposed to keeping documentation externally. They believe this will give a certain amount of robustness to the documentation which they do not have currently.
They also stated that the visualization and navigation that the tool provides is necessary to benefit from the stored knowledge, which could increase exponentially for very large systems, such as the case study. They believe that the additional metrics provided by the tool helps to measure properties of the system that are useful for making future decisions about the features, such as refining the feature.
Finally, the experts were confident that the approach will work well with their currently used agile development method. A thorough and systematic evaluation together with the experts, studying the exact usage, benefits, and costs of using the tool FLOrIDA and the embedded annotations could be the focus of an extension study.

5

An ML Algorithm Based Recommendation System For Feature Location

As mentioned in the introduction, feature locations documented as embedded annotations still require significant effort to create and maintain. This section presents an experiment, meant to answer the question of how to transfer feature location knowledge from an expert to a classifier.

5.1 Approach

In all Feature Location Approaches (FLA)s, the expert undertaking the feature location task, provides some form of encoded knowledge to the FLA. This encoded knowledge contains hints to the approach about how to locate the source artifacts of the feature the expert is trying to locate. In dynamic approaches, the expert provides test cases which exercise the feature. The test cases serve as a rule set mapping all source artifacts exercised by them to the feature that they test. In static and IR based approaches, the user also provides some encoded knowledge to the FLA. In this case however, the knowledge is in the form of a string query with key words. The rule here is: all source artifacts matching this query implement the feature described by the query. These approaches described above and indeed a large majority of existing FLAs use indirect ways of providing feature location knowledge to the FLA.When such test cases or query strings exist in real industrial cases, they may serve as good starting point for locating features. However, as described in previous chapters, inputs required by dynamic approaches are often unavailable and expensive to create. Besides this, there are also inherent problems with the approach used by previous FLAs, that make them inaccurate. For example, test cases are often not designed with feature location in mind. They may therefore exercise more than one feature at a time or may not execute all possible paths in a feature's use case. Thus, using such test cases for feature location may results in inaccurate or incomplete results. Information retrieval approaches based on using queries have also been found to return many false positives. This is because, software assets that implement different features may share a common vocabulary. As a results using this input solely for feature location may provide faulty results.
The approach proposed in this work, presents a more direct way of transferring feature location knowledge to the FLA. In this approach, as depicted by Figure 5.1,

the user directly selects a sample of source artifacts that they already know are feature locations for a feature of interest ❶. The FLA then analyzes the values of the properties of this supplied sample for patterns that connect them together ❷. When such a pattern is found, it learns it ❸ and then uses it to find and propose other artifacts that have similar properties ❹. In the approach, finding source artifacts with similar properties is done with the help of a classifier. The feature location task is formulated as a classification problem. The sample feature locations supplied by the expert are transformed into a training data set for training the classifier. The trained classifier is then used to predict feature associations for new source code, that a developer adds.
The classifier is retrained at a decided interval as the development proceeds. It is expected that the accuracy of the classifier will increase over time as the size of the training data set increases until a point where the accuracy is sufficient.

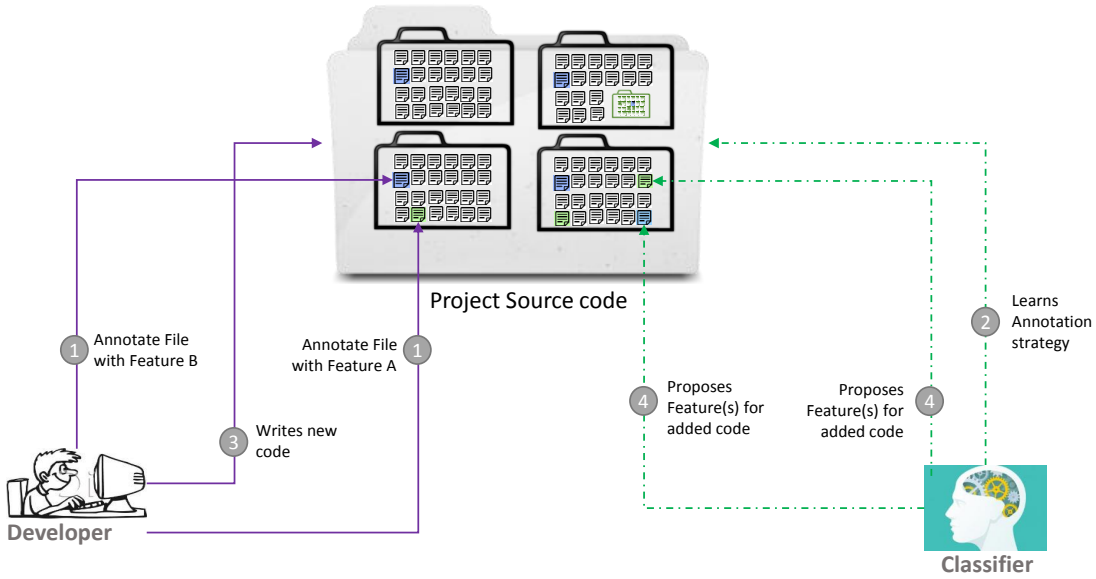


Figure 5.1: Feature-file trace view

5.2 Methodology

To explore the feasibility of using machine learning for the task of feature location, an evaluation of different factors affecting the predictive performance of such a machine is needed. These factors may include for example, which machine learning algorithms provide the most precise predictions? What source code properties give the best description of source artifacts for feature location? How many source artifacts must already exist for a the best machine to predict accurately? Experiments were setup to find out the answers to these questions. Each of these experiments simulate a software development scenario where a developer has added new source code to a system's repository and then asked a trained machine to predict feature locations for the newly added code. The predictive accuracy resulting from using different

alternatives of the factors are then recorded to find out which of the alternatives provide the most accurate results.
Together, the set of experiments over all commits of the system, simulate the performance-evolution of the alternatives over time as the system evolves (i.e. from the beginning of development when the system has only a few lines of code, to the end, when it is a fully developed system with millions of lines of code).

5.2.1 Research Questions

Concretely, this part of the thesis tries to answer the following sub-questions of RQ2 (presented in the introductory section).

RQ2.1: What source code properties are best predictors of feature locations?
RQ2.2: At what granularity of source code, is feature location using machine learning most accurate?
RQ2.3: What machine learning algorithm(s) provide the most accurate predictions of feature locations?
RQ2.4: How many example feature locations must already exist for the best configuration to give the best predictions?
RQ2.5: How often must a machine be retrained to get good predictions?
RQ2.6: How accurate is a classifier when predicting feature associations for code that do not directly implement any features?

The setups used to investigate these questions are described in more details below.

5.2.2 Exepriment Setup: RQ2.1 Best source code properties for feature location

To train a classifier to recognize a feature's implementation, properties of the feature's current implementing code must be exposed to the classifier in a machine readable format (i.e. as shown in Listing 5.1), from which it can learn to identify similar source code that implements the same feature. new locations by recognizing new source code with similar property values. To create this descriptive property set for representing a feature location, several IR and static source code properties were evaluated in the experiments.

SCP Sets		
SCP Set 1	**SCP Set 2**	**SCP Set 3**
SCLD, NAEFA, TSM	NAEFA, TSM	TSM

Table 5.1: Source code property sets used in the experiments

These properties include:

1. Number of Already Existing Annotations (NAEFA
2. Cosine Text Similarity Metrics (CTSM): A Set of text similarity scores between a piece of code and the vocabulary used in all of the known locations for that feature.
3. Distance score between location of piece of source code and the location of all other feature locations for a feature (SCLD).

These are described in more details in the following subsections. The motivation for selecting IR and static source code metrics as described in previous sections is that static and IR properties are easier to obtain and readily available in real life compared to dynamic data.
The existing annotated feature implementing code for each feature in the subject system were then extracted from the source code using the FLOrIDA tool. Then, for each annotated feature location, a data vector is created to represent it (i.e. as shown in Listing 5.1, with a concrete example in Listing 5.2). The data vector contains the metric values of the properties of a feature location mentioned above. The combined vectors for each feature are then used as data points to train the classifier to recognize feature locations for that particular feature.
Different well known text similarity metrics were evaluated in the initial experiments however the CTSM metric gave the most promising results. Details of these metrics are described in the subsections below:

```
1  [FPC:{F1_(0/1), F2_(0/1), F3_(0/1), ... ,F(n)_(0/1),}
2  CTSM_metric1:{ctms_score_F1, ctms_score_F2, ...ctms_score_F(n)},
3  SCLD_metric:{scld_score_F1, scld_score_F2, ...scld_score_F(n)},
4  NEAF_metric:{neaf_score_F1, neaf_score_F2, ...neaf_score_F(n)},]
```

Listing 5.1: Feature Data Vector Encoding

```
1  [1, 0, 0, 1, 0,0.64, 0.32, 0.15, 0.00, 0.11,0.75, 0.01, 0.03, 0.75, 0.00,2, 0, 0, 1, 0]
```

Listing 5.2: Feature Data Vector Encoding Example

Feature Presence Condition (FPC)

As shown in the blue line of Listing 5.1 and with an example in the blue line of Listing 5.2, the set of known features in the system are represented by a set of equally numbered set of binary numbers. In the training data set, if a feature is annotated on a piece of code, then the value for that feature in the feature data vector representing that piece of code will be 1 otherwise it will be 0. In the test data set, the value of the FPC are zero, representing that no features are currently associated with the piece of code. The classifier sets the values of the associated detected features to one, and the remaining features remain 0.

Number of Already Existing Annotations (NAEFA) Metric

Another metric used to predict feature locations for newly added code is the number of locations each feature has at the point when a new code is to be classified. For example, if Feature A by the third commit is associated with 2 annotations and Feature B is not associated with any annotations, the score of Feature A and B on this metric is 2 and 0 respectively. In Listing 5.1, NAEFA is represented by the read line with a concrete example on the red line of Listing 5.2.

Text Similarity Metrics

Several well-known text similarity metrics were evaluated in the study. The metrics were used to try to quantify in numeric terms how close the vocabulary used in a

piece of source code to be classified is to the vocabulary used in the source code of the already known feature locations for each feature in the system. The hypothesis here is that, the similarity score between the vocabulary in a piece of code and that is also a feature location for a feature should be close to the vocabulary used in the code of already existing feature locations of that feature than for any other feature in the system.
There are several already existing similarity metrics algorithms to choose from. To make a choice of the best similarity metric, a few of these were selected and evaluated in our experiments. These are described below. However, after some initial experiments, the CTSM gave the best output and thus was used in the encoding. In Listing 5.1 the CTSM metrics is represented by the green line of numbers, with a concrete example on the corresponding green line of Listing 5.2

Cosine Text Similarity Metric (CTSM): The cosine similarity between two documents represented as vectors, is a measure that calculates the cosine of the angle between them. It is thus a judgment of orientation and not magnitude: two vectors with the same orientation have a cosine similarity of 1, two vectors at 90° have a similarity of 0, and two vectors diametrically opposed have a similarity of -1, independent of their magnitude. In our experiment, the source code extracted from an annotation is considered as one document and the description of a feature as the other document [2].

Jaccard Similarity Metric: This is another token based vector space similarity measure like the cosine distance and the matching coefficient. Jaccard Similarity uses word sets from the comparison instances to evaluate similarity. The Jaccard similarity penalizes a small number of shared entries (as a portion of all non-zero entries) more than the Dice coefficient [1].

Levenshtein distance: The Levenshtein distance is the basic edit distance function whereby the distance is given simply as the minimum edit distance which transforms string1 into string2 [2]. Edit Operations are listed as follows:

1. Copy character from string1 over to string2 (cost 0).
2. Delete a character in string1 (cost 1).
3. Insert a character in string2 (cost 1).
4. Substitute one character for another (cost 1).

The distance measure is calculated by summing up the cost of the copy, insert and delete operations which are calculated with the formulas 5.1, 5.2 and 5.3 respectively.

$$D(i-1, j-1) + d(si, tj)//subst/copy \tag{5.1}$$

$$D(i, j) = minD(i-1, j) + 1//insert \tag{5.2}$$

$$D(i, j-1) + 1//delete \tag{5.3}$$

$$d(i, j)isafunctionwherebyd(c, d) = 0if c = d, 1else \tag{5.4}$$

Needleman-Wunch distance or Sellers Algorithm Metric: This approach is known by various names, Needleman-Wunch, Needleman-Wunch-Sellers, Sellers and the Improving Sellers algorithm [2]. It is similar to the basic Levenshtein distance. It adds a variable cost adjustment to the cost of a gap, i.e. insert/deletion, in the distance metric. So, the Levenshtein distance can simply be seen as the Needleman-Wunch distance with G=1. Distance measure is calculated by summing up the cost of edit actions needed to convert one one string to another. The cost of the copy, insert and delete actions are calculated using the formulas 5.5, 5.6, 5.7 respectively.

$$D(i-1, j-1) + d(si, tj)//copy \tag{5.5}$$

$$D(i, j) = minD(i-1, j) + G//insert \tag{5.6}$$

$$D(i, j-1) + G//delete \tag{5.7}$$

Matching Coefficient Metric: The Matching Coefficient Metric is a simple vector based approach which simply counts the number of similar terms, (dimensions), on which both vectors are non-zero. So, for vector set X and set Y the matching coefficient is |X & Y|. This can be seen as the vector based count of co-referent terms. This is similar to the vector version of the simple hamming distance although position is not taken into account. [2]

Block Distance Metric: It computes the distance that would be traveled to get from one data point to the other if a grid-like path is followed. The Block distance between two items is the sum of the differences of their corresponding components. [2]

Euclidean Distance Metric: This is calculated as the square root of the sum of squared differences between corresponding elements of the two vectors. [2]

Source Code Location Distance Metric (SCLD)

Another property of source code that was used for predicting feature locations for newly added code is the relative distance of that piece of code to the location of known feature locations in the project structure.
This metric tries to calculate a depth measure to express how far a piece of code to be classified is to the already known locations of a feature in the project structure. For example, a file *compress.c* can be 50% close to file *send.c* because file *compress.c* shares 50% of its file path with file send.c. File *delete.c* on the other hand can be 0% similar to file *send.c* because it is in a completely different file path. This distance measure is calculated for all known feature locations for each feature. Then an average is calculated to get a measure of how far the piece of code is to the feature. SCLD is represented by the brown line in Listing 5.1, and 5.2).

5.2.3 Exepriment Setup: RQ2.2: Best source code granularity for feature location

Beside this, Source code committed to a repository could be a few lines of code, whole files, a folder or a mixture of these. Thus, another open question the study aims to answer is at what source code granularity feature location must be done, to get the most accurate predictions? To answer this question, experiments were run, where different granularity levels were evaluated.
The granularity options that were tested in the experiment were file, folder and LoC level granularity. Lines of code added to a file are aggregated into a single data-point, when predicting on the file level, each single file addition is treated as a data-point on its own and multiple files added in a folder are separated per file, and each file is treated as an independent data-point. When predicting on the LoC level, each new line added to the source code is treated as a data-point on its own. If a whole file is added to the source code, each line in the file is extracted and treated as a separate data-point.

Source Code Granularities Tested
LoC (Line of Code)
File
Folder

Table 5.2: Source Code Granularities tested in the experiments

5.2.4 Exepriment Setup: RQ2.3: Best Machine learning algorithm for feature location

This section describes the classifiers evaluated in the study and the reason for selecting them. The goal was to select a classifier that gives the best prediction for feature locations. The best classifier must be able to support multilabel classification as the problem to be solved is of a multilabel nature. Besides this, other requirements are that the classifier must also be able to support categorical and numerical data, handle incomplete data and be accurate while working with small samples of data. The last requirement is particularly important as this would be a good incentive for developers looking to adopt the approach (i.e. they do not have to have a large sample annotated code set before the classifier is accurate enough).
A popular problem transformation algorithm, Binary Relevance (BR) algorithm was selected for use in the study. The BR algorithm was selected for use particularly as it provides the freedom of using different kinds of underlying binary classifiers. This made it possible to evaluate several popular classifiers which were evaluated in the study. The underlying binary classifiers selected for evaluation were the Support Vector Machine (SVM), Decision Tree (DT), K-Nearest Neighbor (kNN) and Multi-Layer Perceptron (MLP).
Implementations of these classifiers found in the Waikato Environment for Knowledge Analysis (WEKA) [25] machine learning library were used in the study. Each of the algorithms have tuning parameters which in the ideal case can be tuned to optimize its predictive performance for the problem being modeled. However, there is

also often a possibility to over tune the algorithm and thus to influence its predictive performance for the particular case study (i.e. in effect reducing the transferability of the predictive performance to other systems). To avoid this tuning bias, the defaults values of these parameters were used. The algorithms are described briefly below.

Binary Relevance(BR) BR is the most commonly used problem transformation algorithms for multilabel classification. Using BR, a multilabel data set is transformed into a set of single label data sets. Then, BR learns a binary classifier for each label in a multilabel problem [53].

Support Vector Machine (SVM) The classifier classifies data-sets into different labels by calculating a maximal margin hyper-plane that separates the classes of data.
To learn non-linearly separable functions, the data being classified is mapped to a higher dimensional space where a separating hyper-plane is found using a kernel function. New samples are classified according to the side of the hyper-plane they belong to [33].

Decision Tree (DT) Decision tree classifiers label data-set into different classes by grouping them based on the value of their properties. The classification mechanism of a decision tree is a tree-like structure where each node in the tree is a property of the data-set and each branch represents values or value ranges the data instances can take. Individual data instances are then classified by sorting them based on value of their properties, starting from the root of the tree until they end up in a branch of the tree (i.e. their class)[40].
The property of the data-set that best divides the instance of the data-set are usually selected as the root of the tree. Several approaches exist for finding this best property including the gini index[16] and information gain [15]. The information gain approach was used in this study.

Machine Learning Algorithms Used	
kNN	Naive Bayes
SVM	Bootstrap Aggregation
DT	Stacked Aggregation
Random Forest	

Table 5.3: Machine Learning Algorithms tested in the experiments

5.2.5 Exepriment Setup: RQ2.4: How many example feature locations must already exist for the best configuration to give the best predictions?

To find out how the treatments performed at different stages of software development, the subject system's evolution history was divided into stages. These stages

correspond to changes made to the source code as the system is being developed (i.e. as found in its version control history). Each change of the source code (i.e. commit) is considered as one stage in the systems evolution. Thus, to predict features for the source code in the change set of commit *(n+1)*, feature locations found in the source code up to commit n are used to train the machine. To predict feature locations for the source code in the change set in commit *(n+2)*, feature location in the source code up to commit *(n+1)* are used to train the machine and so on.

5.2.6 Exepriment Setup: RQ2.5: Best Training Interval

Additionally, it is useful to know how often a machine must be retrained after new source has been added in order to get the best predictions. To achieve this, experiments were also run where the wait time before retraining is done were varied. Since software systems can be quite large, it may not be feasible to retrain the machine every time a new commit is made. Especially when the system is being developed by multiple people concurrently. To answer the question of how the retraining intervals affect a machine's accuracy, we set up experiments where the length of retraining time was varied. We evaluated three different retraining lengths:

1. Retrain after each commit
2. Retrain after every tenth commit
3. Retrain after every twentieth commit.

5.2.7 Experiment Setup: RQ2.6: How accurate is a classifier when predicting feature associations for code that do not directly implement any features

In the subject system used in the experiments, some source code were annotated and others were not. Non of the experts who did the original annotations were around to provide a reason why these source code were not annotated. Several plausible alternative reasons why these may have not been annotated can be imagined. These reasons may include:

1. Those pieces of source code are not directly associated with any features
2. They are associated with all features of the system.
3. They are associated with a feature(s) but have not been annotated yet.

To be certain of the accuracy score that can be attributed to the classifiers being evaluated, ground truth with certainty is needed. In the study, this ground truth was obtained from annotations(i.e. if a piece of source code is annotated with feature A, a classifier is wrong if it associates it with feature B, or with no features at all). However, since it was not clear why a large part of the code was unannotated (As there were no experts to explain why they were not annotated). Thus, if a classifier associates a piece of unannotated source code with feature A, we cannot be sure if it is right or wrong. It would be right if we assume reason 3 to be true but wrong if we assume reason 1 or 2 to be true. The safest choice was thus to leave all such pieces of code out of the experiments. Hence, all such source code were ignored in the experiments to test the hypothesis for R.Q 2.1 to 2.5.

However, if we assume reason 1 to be true, we can attempt to test the hypothesis of how accurate a classifier would be, when classifying source code that are not directly associated with any feature of the system. Thus, to answer R.Q 2.6, an experiment was set up where a classifier is trained with annotated source code and tested on a mix of annotated and unannotated source code. The expectation is that, the classifier would associate the annotated source code with their correct features and will not associate the unannotated features with any features at all.
Further more, since a large majority of the source code in the subject system were unannotated, it took a long time to run the experiments for each version of the source code (training data:commit n, test data: commit n+1) if all unannotated source code were included in the test set. Thus instead of using the entire unannotated source code in each test version, a sample of the unannotated code, corresponding to 40% of the size of the annotated test set for that version is randomly selected. Together, the annotated test set and the selected 40% are then used to test the classifier. To ensure random selection of the unannotated test set. The LoCs are places inserted into a list at random and then the first n elements corresponding to 40% of the size of the annotated test set are then selected.
The reason why a fixed percentage of the unannotated part of the test set was fixed at 40% of the size of the annotated test set, was so that, a rough numeric estimate can be calculated of the size of the effect the added unannotated source code would have on the accuracy of the classifier, as compared to the classifier's accuracy score recorded in RQ 2.3. It should be noted however that, the number 40 in itself was not chosen for any special reason. The size could as well have been 20, 50 or 60% and still serve the same purpose.

5.2.8 Steps in an Experiment

In conducting each of the experiments described above, the following steps were followed:

1. Source code (i.e. containing some feature annotations) in an nth commit is pulled from a source code repository (i.e. as would be the case when a developer has written an initial set of source code and annotated it with features).
2. The Source Code Property (SCP) set (i.e. from Table 5.1 and explained further below) being evaluated in the current experiment run, are then extracted from the existing set of annotations for each feature in the pulled source and converted into a data vectors for training the machine.
3. The MLA (i.e. from Table 5.3) being evaluated in the test run is then trained with the data vectors.
4. To test the accuracy of the just trained machine, source code from the *(n+1)*th commit is pulled (i.e. as would be the case when a developer has added new code and would like to get feature predictors for it)
5. For each annotated LoC and un-annotated LoC(i.e in the case of RQ2.6 testing the behavior of the machine when tested on un-annotated source code) source in the change set, the same SCPs as in the training data are extracted and a data vector is built to represent the LoC.
6. For each of the data vectors in the test set, the machine then predicts its

feature associations.

7. The prediction is then compared to the actual annotations in the pulled change-set, to calculate the treatment's accuracy for the experiment run.

The above process is then repeated with all combinations of SCPs and algorithms in each stage of evolution to test each treatments accuracy for that stage.
One algorithm will be tried at each test run. But each algorithm will be run on all sets of SCPs. A stage in the source code evolution includes a set of two commits from the source codes version control history. An nth commit (i.e. used for training the ml algorithm) and an *(n+1)*th commit used for testing the algorithm.
Each of these factors has several alternatives.

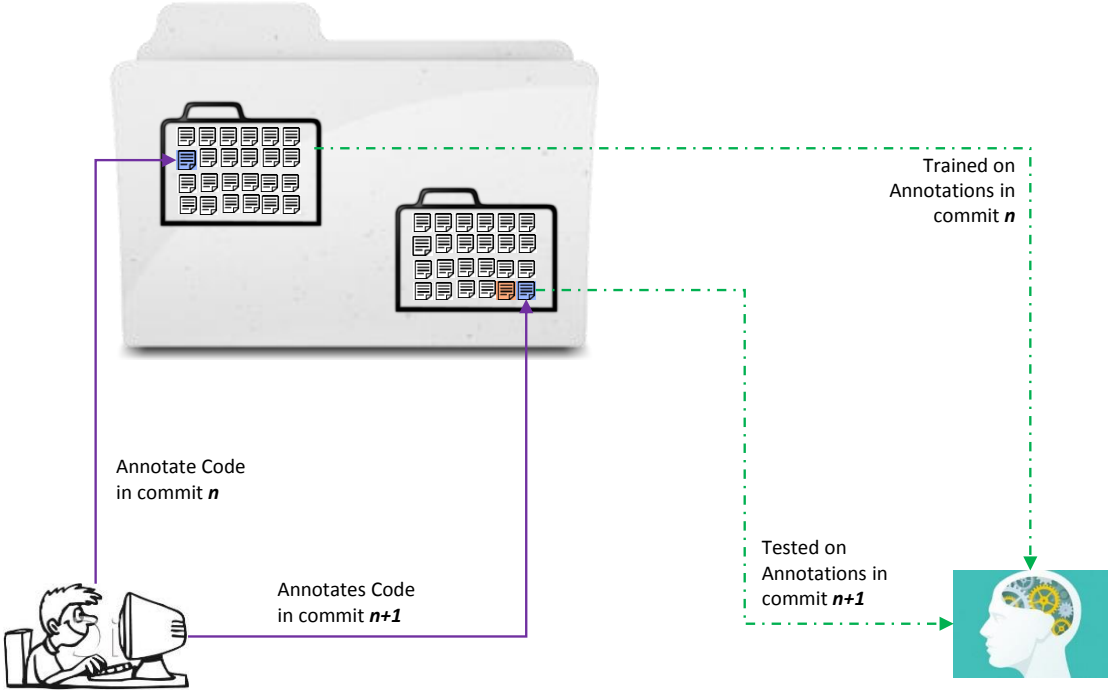


Figure 5.2: Training and Test Data Combinations

5.2.9 Evaluation

To test the performance of each algorithm during each experiment the the well known classification metrics precision, recall and the f-measure as described in the formulas below are used.

Relevant Features (i.e. regards a LoC(x): All features in the set of features x is implementing according to the training data

Non Relevant Features (i.e. regards a LoC(x): All features **not** in the set of features x is implementing according to the training data

$$\textbf{f-measure} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \tag{5.8}$$

$$\textbf{precision} = \frac{\text{relevant features} \cap \text{retrieved features}}{\text{retrieved features}} \tag{5.9}$$

$$\textbf{recall} = \frac{\text{relevant features} \cap \text{retrieved features}}{\text{relevant features}} \tag{5.10}$$

$$\textbf{false positive} = \text{retrieved features} - \text{correctly labelled features} \tag{5.11}$$

$$\textbf{false positive} = \text{retrieved features} - \text{correctly labelled features} \tag{5.12}$$

$$\textbf{true positive} = \text{retrieved features} - \text{incorrectly labelled features} \tag{5.13}$$

$$\textbf{true negative} = \text{retrieved features} - \text{correctly labelled features} \tag{5.14}$$

For each stage in the evolution history, we record the optimum set of code attributes that gives the highest accuracy, and each algorithms performance with this set. Over the entire evolution history of the project we track the overall performance of each *treatment* and calculate an average performance score. From the analysis of this gathered data we can then try to answer each of our research questions.

5.2.10 Subject System

To run the described experiment and effect to answer our research questions, we need a subject system with already existing feature annotations that we could use as ground truth to both train and test the algorithms and to select appropriate source code artifact properties. The approach we develop is designed to be programming language independent and so the subject system could be written in any programming language. The subject system we used in our experiment was the Clafer Tool set.

Clafer Tools

Clafer Tools is a tool set containing 4 tools developed in HTML and JavaScript. The tool set was developed using clone-and-own and feature annotations were created whiles the features in the system were being implemented. Therefore, each feature in the system has several associated feature annotations, which link the feature to its implementing source code. We had access to the system's version control history consisting of commits of feature annotated code over the development life cycle of the systems in the tool chain. We could therefore, replay this history to simulate the prediction of feature locations during its development. The properties of the Clafer Tool chain are summarized in the table below:

General Project Properties	
Programming Language	JavaScript
Size (LoC)	10,000
Number of developers	15
Features	
No. Features in project	82
No. Features with Annotations	20
Avg. No. Annotations per Feature	60
Feature Annotations	
Total No. Feature Annotations	1400
No. Single-Line Annotations	150
No. Multi-Line Annotations	300
No. File Annotations	700
No. Folder Annotations	250
Code Evolution History	
No. of Commits in VC history	150
Avg. Annotation per commit	5

Table 5.4: Clafer Tools Source Code Properties

Description of Data-set

As shown in 5.4, our data set included a total of 1400 feature annotations tracing the 81 features of the Clafer Tools System to its source code. The Annotations are distributed over 150 commits of feature code which were made by 15 developers over the life span of the system. The average number of feature annotation per commit was 5. Each feature had an average of 60 annotations. The annotations were extracted from the source code of Clafer Tools. Out of these annotations, 150 were Single-Line Annotations, 300 were Multi-Line Annotations, 700 were File Annotations and 250 were Folder Annotations.

5.3 Experimental Results

The results of the experiments are now presented in the subsections below:

Table 5.5: Average F-Measure Scores for ML Algorithm and Source Code Properties

	SCP			
ML Algorithm	SCP Set 1	SCP Set 2	SCP Set 3	**Averages**
SVM	**0,99**	**0,97**	0,49	0,82
kNN	0,97	**0,97**	**0,66**	**0,87**
Averages	**0,98**	0,97	0,58	

Figure 5.3: F-Measure Scores for Combinations of SCLD, TSM, NAEFA Over Time

5.3.1 RQ2.1: Best Source Code Properties for prediction

Several combinations of SCPs were tested in order to find out which SCP or combination of SCPs provides the highest predictive accuracy when used for feature location prediction. The results of the experiments show that the best is the combination of SCPs: SCLD, TSM and NAEFA). It achieves the best micro F-measure score across all training stages and classifier at 98% accuracy. The second-best combination of source code properties was the set as the TSM and NAEFA with an average F-measure score of 97%. This combination produces an acceptable prediction accuracy rate compared to the maximum set even with one less predictive attribute. The next best performing set of source code properties was TSM by itself. Its produces an average F-measure score of 58% as shown in the table 5.5.

5.3.2 RQ2.2: Source Code Granularity for prediction

Feature locations in terms of granularity can also be a LoC, file or a folder container multiple sub folders. Another important question open to answer is thus at what source code granularity a classifier must classify newly added code. An intuitive approach would be to use the same level of granularity as the developer has used for annotating the source code. Thus, if the developer annotates a line of code and a folder, the folder and line of code will both be transformed into a training data point and used to train the machine. Another approach would be to split the entire annotated sample source into file clusters. Each file will then be converted into a training data point. Using this option, newly added code is also aggregated/split into files and then classified accordingly. Thus, features will only be mapped to files using this option. On an even more coarse level the sample source code can be aggregated/split into folder units and used for both training and prediction. As the

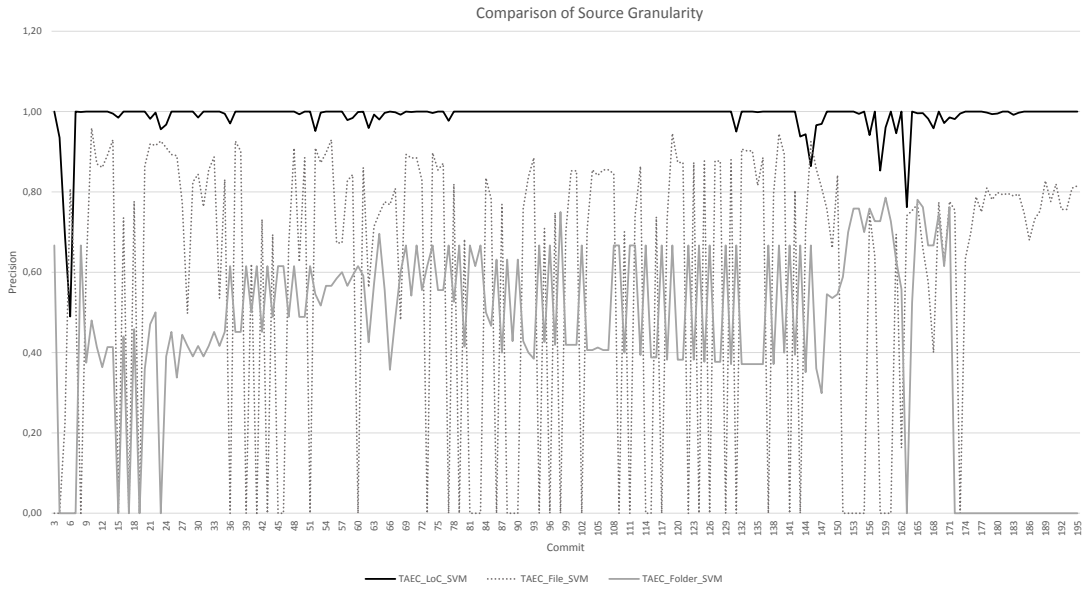


Figure 5.4: F-Measure Scores for Source Code Granularity over Time

Source Code Granularity	LoC	File	Folder
F-Measure	**0,97**	0,65	0,28

Table 5.6: Average F-Measure Scores for Source Code Granularity over Time

other extreme end, the entire sample source code can be split into LoCs and used for training. Newly added code is also then split into LoCs and classified individually. Because it is not clear which of these options results in a better prediction performance, an experiment was set up to compare the performance of each of these options. As shown in Figure 5.4 and 5.6 overall, the LoC granularity level provided the best average predictive performance with an average F-measure score of 97% followed by the file level granularity with an F-Measure score of 65%. The folder level granularity performed poorest with an F-Measure score of 28%.

5.3.3 RQ2.3: Best performing classification algorithm

The SVM algorithm performed consistently better than all other tested algorithms throughout the life-span of the Clafer tools system whiles using the maximum set of source code properties (i.e. SCLD, TSM and NAEFA) with an average F-Measure score of 99%. The kNN algorithm was the next best performing algorithms with an F-Measure score of 97%. While using the source code property set TSM and NAEFA, the kNN algorithm and SVM algorithm perform equally well with an F-Measure score of 97% each. Finally using only the TSM source code property the DT algorithm outperforms the SVM algorithm with an F-Measure score of 66% against 49%. On the average however, over all three sets of source code properties, the DT classifier performed better than the SVM algorithm with an average of 87% against 82%.
What these results show is that; the choice of source code property affects the

performance of the algorithms. Thus, if the user decides to use the source code properties TSM and NAEFA, then the DT algorithm it is not very important which classifier the user should select as each of them performs equally well with this set. However, if the user chooses to use the maximum set of source code properties (i.e. SCLD, TSM and NAEFA) then the SVM classifier is the go to algorithm. One the other hand if the user would like to only use the TSM source code property exclusively, the DT algorithm is the better choice.

As mentioned in the Section 5.2.4, the default settings of the algorithms as provided in the WEKA [25] was used for each of the algorithms. Thus, the results presented here are valid only for this configuration. Thus, the performance of the algorithms could be entirely different if the algorithms are tuned to match the data set being analyzed. The goal of this work is not to find the best configuration of each of the algorithms but to evaluate on a high level the feasibility of using these algorithms in general for the feature location task.

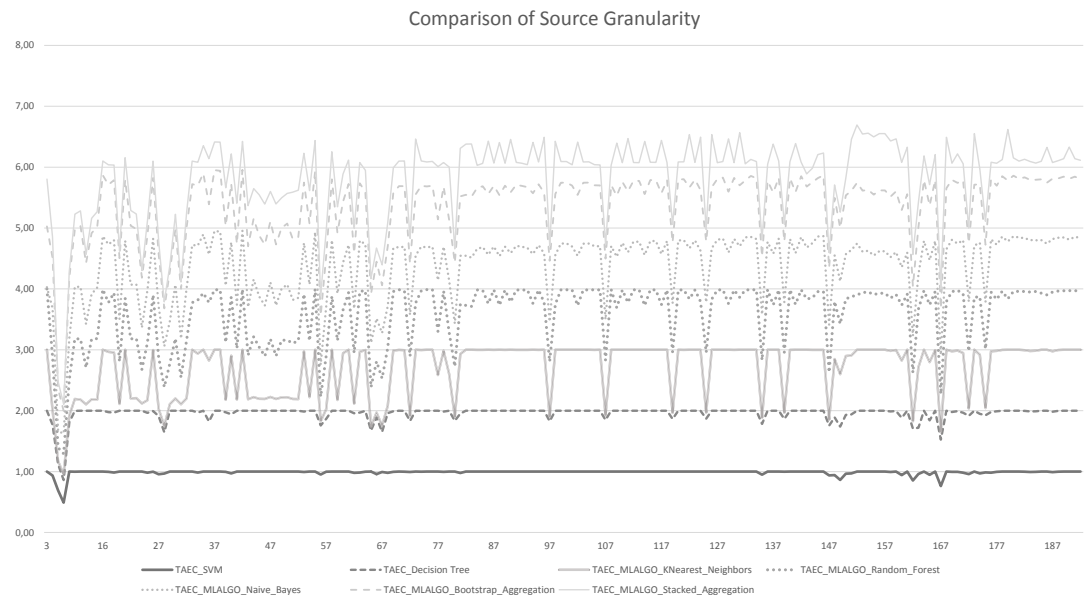


Figure 5.5: F-Measure Scores for evaluated machine learning algorithms over time

Table 5.7: Average F-Measure scores of evaluated algorithms

ML Algorithm	Average F-Measure (All stages)
SVM	0.99
DT	0,97
KNN	0,74
Random Forest	0,89
Naive Bayes	0,79
Stacked Aggregation	0,45

Table 5.8: Average F-Measure Scores for Training Intervals

Training Interval	Every Commit	Every Fifth Commit	Every Tenth Commit
F-Measure	**0,99**	0,96	0,94

5.3.4 RQ2.4: Initial manual effort requirement

An important question to answer is how much effort is required for the initial effort required of the programmers to create the initial set of feature locations for training the classifier.

As shown in the Figure 5.4 while predicting on the LoC granularity, the size of initial training set required for getting good predictions is really low for both classifiers. Both classifiers become very stable very early in the development cycle (i.e. commit 16) when NAEFA is around 600 feature locations. However, the SVM algorithm becomes stable much earlier than the DT algorithm at commit 10 when the number of annotations are around 6.

5.3.5 RQ2.5: Required Interval for Retraining Classifier

As a software development process is continuous process of adding or removing code to implement new requirements or to remove no longer needed ones, the classifier must also be continuously updated in order to stay up to date with the changing nature of the properties of a feature (which it tracks) in order to maintain its predictive performance. However, a question that remains open is how frequent this retaining process should occur in order to maintain an acceptable level of predictive performance. To answer this question, experiments were run where the retraining interval was varied to see the effect of different intervals on predictive performance. Retraining was done in intervals of; after every commit, after every 5th commit, after every 10th commit. The results of the experiments, as shown in table 5.9 shows that the best average predictive performance is recorded when retraining after every commit at 99% accuracy, followed by an second best average accuracy of 96%, when retraining after every 5th commit. Finally, the comparatively worse average accuracy of 94% is recorded when retraining the machine after every 10th commit. Thus, the results appear to favor a more frequent retraining interval to a less frequent one.

However looking at the graphs in Figure 5.8, the effect of the retraining interval is more sever in the early phase of the project (i.e. when the size of training examples is small). Eventually, the predictive performance of the classifiers become similar regardless of the retraining interval after a certain number of commits depending on the retraining interval. For the intervals of 1 and 5 commits the machine becomes stable on the 7 commits and for the 10th interval the machine becomes stable after the 11th commit.

Table 5.9: Average F-Measure, Precision and Recall Scores for Un-annotated Data

	Recall	Precision	F-Measure
SVM	0,07	0,23	0,10
kNN	0,52	0,45	0,48

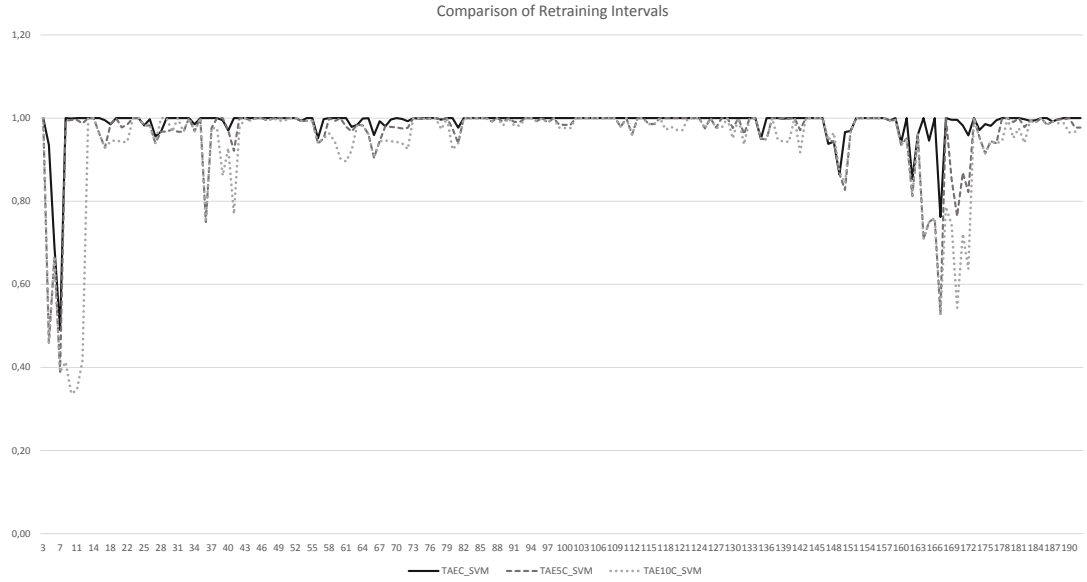


Figure 5.6: F-Measure Scores of Training Intervals over Time

5.3.6 RQ2.6: Classifier Accuracy for Unlabeled Test Data

To find out how a classifier behaves when predicting source code that do not directly implement any features, an experiment of this nature was carried out. The experiment was carried out using the LoC level of granularity and the best performing SCPs (i.e. the combination of the SCPs, SCLD, TSM and NAEFA) and the two best performing MLA algorithms kNN and SVM. To reduce the number of data points to process, (i.e. as processing takes a long time when the entire set of all un-annotated LoCs for each stage is used) a sample of the test data was selected instead of running the experiment on all of the un-annotated data. It was ensured that exactly 40% of the test data were unlabeled. The results of the experiment as illustrated in Figure 5.7 shows that, there is a decrease in the performance of both of the best performing algorithms:SVM and kNN. Figure 5.9 and Table 5.10 show a comparison graph of the accuracy achieved by the kNN in R.Q 2.3 (i.e. reduced by 40% per stage) and the performance of the same algorithm in RQ 2.6. It is worthy to note that, the percentage reduction in the performance overall seems to have a relationship to the percentage of un-annotated data added to the second experiment (R.Q 2.6) as shown in Table 5.10. The kNN algorithm, contrary to the earlier experiment also performed better than the SVM algorithm.

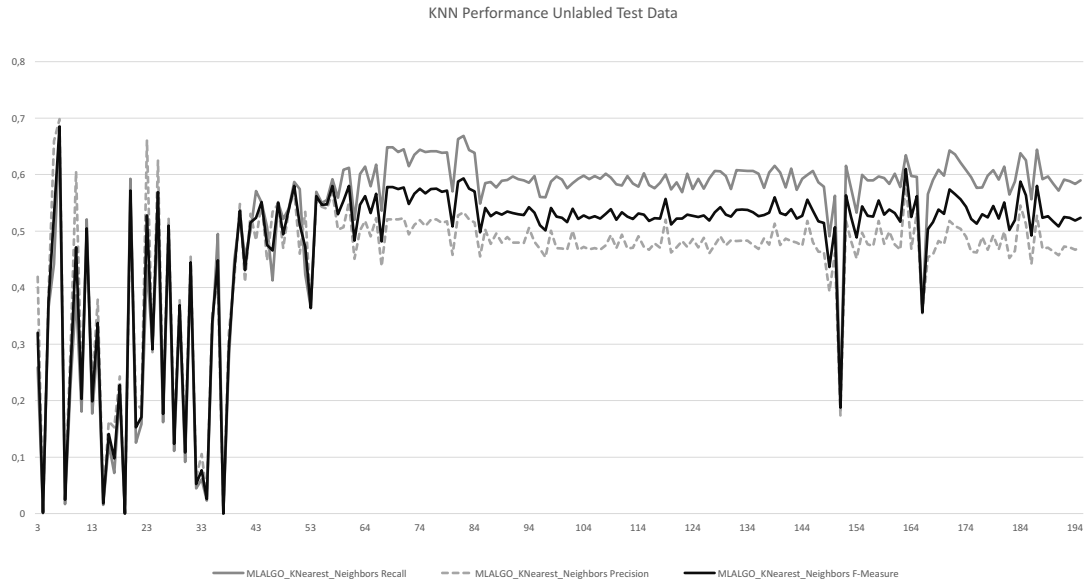


Figure 5.7: Precision, Recall and F-Measure scores of KNN Algorithm over Time

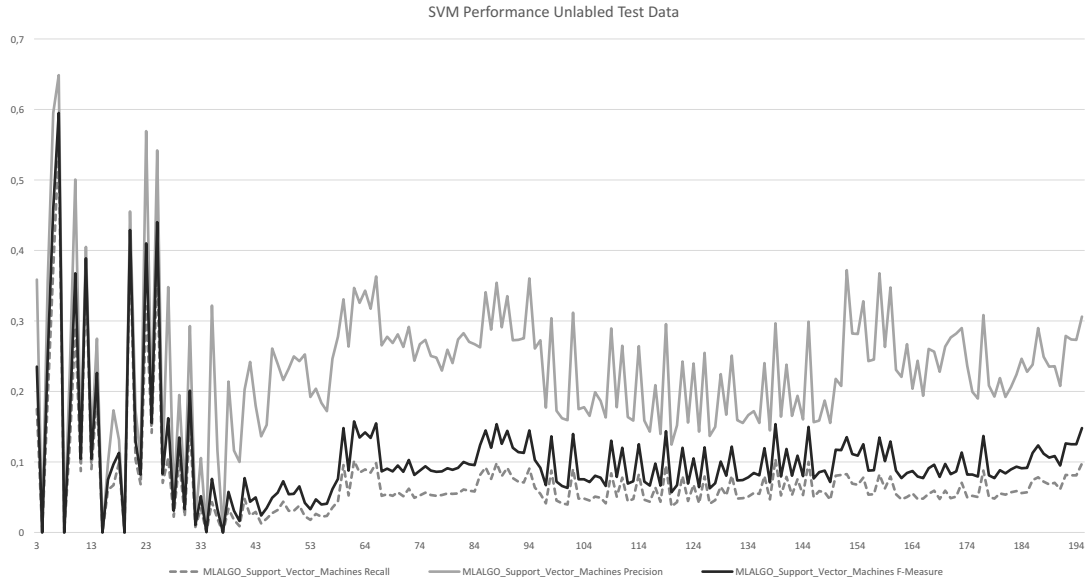


Figure 5.8: Precision, Recall and F-Measure scores of SVM Algorithm over Time

Table 5.10: Average F-Measure, Precision and Recall Scores for Un-annotated Data

ML Algorithm	Average F-Measure (All stages)
kNN(R.Q 2.3) 0.74 * 0.6	0.444
kNN(R.Q 2.6)	0,48

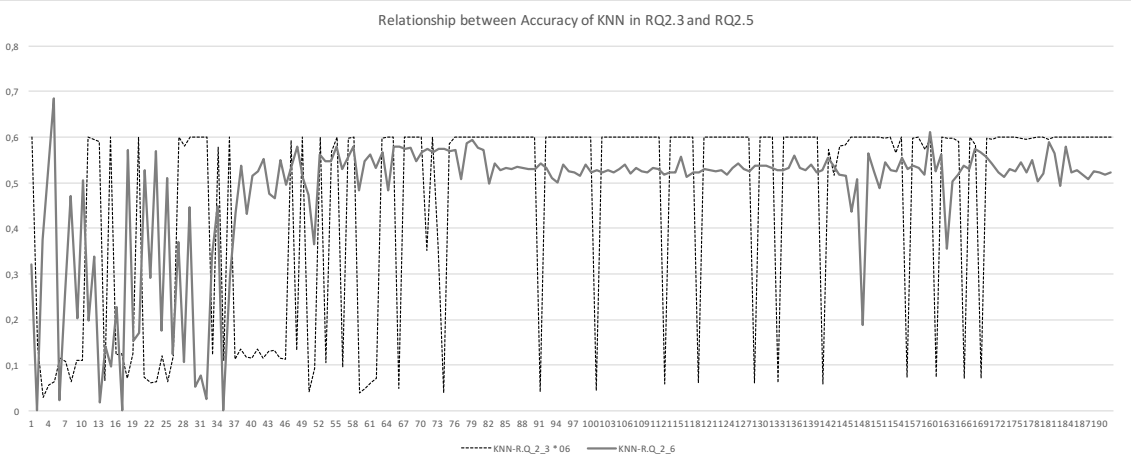


Figure 5.9: Relationship between performance difference of KNN and percentage of added unannotated source code (i.e. RQ2.3 and RQ2.5)

5.4 Limitations of Study

To run the experiments for RQ 2.1 to RQ 2.5 only source code with annotations were selected for creating both the training and test data vectors. This was done in order to be sure that the data used for the experiments were accurate (i.e as there were no experts to explain why the rest of the code were not annotated). Even though this was done to have clean ground truth needed for testing the accuracy of the trained-classifiers, this could also hinder the use of the predictive model in real life. The reason is that, in real life, not all source code may directly implement a feature of the system. Some source code may be infrastructure code that contribute to all features of the system or that supports the general running of the system. In the study all such code may have been removed with the unannotated code. Thus, the accuracy of the classifier for such pieces of code is not shown in this study.
The results presented in Section 5.3 show the classifiers' overall performance for all features/feature types in each stage. The averages thereof show the average classification performance for all features/feature types in all stages. The classifiers may however perform differently for different features/feature types of the system. This study however is of a preliminary nature, to assess the overall potential of using machine learning for the task of feature location, and the factors that may influence such a predictive performance. Further studies may focus on performance differences based on individual features, feature types or annotation types etc.
The results presented in Section 5.3.6 are based on making the assumption that all unannotated code in the subject system were code that did not directly implement

any features. This was a best guess, as there were no experts available to explain why these pieces of code were not annotated. As such, the results may not directly reflect the classifiers ability to classify code that do not implement any features (i.e. the code may not have been annotated for any of the reasons presented in Section 5.2.7.

6
Conclusion

Maintaining feature-source traces manually is a tedious, inefficient and error-prone activity. Software artifacts change so quickly that, maintaining feature-source traces requires a huge time investment in constant revision; a task that is hard to keep up in real life software development scenarios. Feature-source trace documentation is thus often outdated or entirely unavailable. When this knowledge is needed for performing software development tasks, feature location techniques are used to recover it. Several approaches have been proposed in past research for this task. However, a large part of these techniques (i.e. dynamic feature location techniques) require hard-to-get input which are often unavailable in real-life. The remaining part (i.e. static feature location techniques) use available input, but return results with an unacceptable rate of false positives. This thesis contributes to the research on feature-source trace documentation, feature location techniques and feature-source documentation exploitation.

For the task of documenting and exploiting feature-source traces, it presents a lightweight tool to support developers by automating the repetitive tasks of documenting source code and also in exploiting the documented knowledge by providing visualizations and navigation capabilities on them as the volume of documentation can be huge for large industrial systems. The idea is that, the retained embedded knowledge will support developers during other feature oriented software development tasks. For example, the tool can support management of a feature oriented variant-rich system that is not consolidated in a software product line with an integrated platform. To document feature locations in a robust way, it relies on a programming-language-independent, embedded feature-annotation system proposed in previous work. Thereafter, the embedded feature-source documentation is processed and visualized using different kinds of views. Standard metrics about the feature-source relationship is then calculated to help the developer get a feel for the system's feature properties. To evaluate the tools usefulness for the stated purpose, it was used to visualize feature annotations in a large industrial system with 3.2M lines of code. Feedback from 2 experts working on the system provided positive feedback that the tool provides useful feature oriented views of the system and further provided feedback on additional views that may also be useful.

In the second part, the thesis proposes a semi-automated machine-learning-based approach for feature location in source code. Furthermore, it provides experimental results on the evaluation of the approach on a real system. The approach identifies feature locations by leveraging feature location knowledge of developers, and then uses this knowledge as a training data set for a classifier. The trained classifier is then used to predict feature locations for the remaining code in the project or for

future added code. To train the machine, properties of the source code, such as identifier words used in the source code, location of annotated source code in the project structure, and the number of already existing annotations of a feature are used as indicators. Metrics based on the above-mentioned properties are calculated and used to create a data vector which represents a piece of annotated source code from which they are calculated. The data vectors are then used to train the machine. The study presents several contributions. It proposes several source code properties that may be good indicators for a machine learning algorithm, to learn the differences between source code that implement each feature's in the system for which it has a training data set. Further, it proposed an encoding of a piece of code, based on these source code properties, for representing it to a machine. To find out what algorithms are effective for the task, it then explored the effectiveness of using several popular machine learning algorithms for the task of feature location. In addition, the study evaluates the accuracy achieved while varying the granularity level of the source code being classified, as well as the effect of varying retraining intervals of a machine after new codes has been added.

To study the effect of varying these parameters several experiments were then run to simulate the evolution of a project over a 192-commit long life-span. The case study has 10,000 lines of code 82 features and developed by 20 in parallel.

The results of the experiments show that, cosine text similarity, no of already existing annotations for a feature and the location of a piece of source code (i.e. code to be classified, compared to existing known locations of a feature in the project source code structure) together produced the most precise predictions of feature locations for all of the evaluated machine learning algorithms. Nonetheless, using only the TSM and NAEFA properties along also produced acceptably precise results at an average of 98% F-Measure score for both algorithms tested. Using the TSM property alone produced an impressive average F-Measure score of 58%.

The results of the experiments indicate for the granularity question that, classifying source code on the level of LoC produces the most precise predictions.

In the matter of selecting a machine learning algorithm, the results show that when using all three source code properties, the SVM performs better than the DT classification algorithm with an average F-Measure score of 99% and 97% respectively. On the contrary, the DT outperforms the SVM algorithm when only the best source code property (TSM) is used, DT achieves an average F-Measure score of 66% against 49%. There were no performance differences however, between the two algorithms when the TSM and NAEFA source code properties are used together with an average F-Measure score of 97% each.

Future work will focus on extending FLOrIDA with further metrics (e.g., process metrics extracted from an underlying version-control system) and views. It is also planned that in the future, the tool will have the capability of importing feature models and descriptions from other tools and exports metrics to excel. However, most importantly a more thorough evaluation is planed with teams of developers who are maintaining and evolving a large industrial-automation system with many variants.

On the part of the feature location approach, future work will be focus on finding out if tuning the parameters of the algorithms affect the performance of the algorithms

evaluated in this study.

Bibliography

[1] A. V. Aho. Pattern matching in strings. *Formal Language Theory: Perspectives and Open Problems*, pages 325–347, 1980.

[2] B. Andam, A. Burger, T. Berger, and M. R. Chaudron. Florida: Feature location dashboard for extracting and visualizing feature traces. In *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems*, pages 100–107. ACM, 2017.

[3] M. Antkiewicz, K. Bąk, A. Murashkin, R. Olaechea, J. Liang, and K. Czarnecki. Clafer tools for product line engineering. In *SPLC*, 2013.

[4] M. Antkiewicz, W. Ji, T. Berger, K. Czarnecki, T. Schmorleiz, R. Lämmel, S. Stănciulescu, A. Wąsowski, and I. Schäfer. Flexible product line engineering with a virtual platform. In *ICSE*, 2014.

[5] G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia. Identifying the starting impact set of a maintenance request: A case study. In *Software Maintenance and Reengineering, 2000. Proceedings of the Fourth European*, pages 227–230. IEEE, 2000.

[6] G. Antoniol and Y.-G. Guéhéneuc. Feature identification: An epidemiological metaphor. *IEEE Transactions on Software Engineering*, 32(9):627–641, 2006.

[7] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature- Oriented Software Product Lines*. Springer, 2013.

[8] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines*. Springer, 2013.

[9] S. Apel and C. Kästner. An overview of feature- oriented software development. *Journal of Object Technology (JOT)*, 8(5):49–84, 2009.

[10] A. Armaly, J. Klaczynski, and C. McMillan. A case study of automated feature location techniques for industrial cost estimation. In *ICSME*, 2016.

[11] T. Berger and J. Guo. Towards system analysis with variability model metrics. In *VaMoS*, 2014.

[12] T. Berger, D. Lettner, J. Rubin, P. Grünbacher, A. Silva, M. Becker, M. Chechik, and K. Czarnecki. What is a feature? A qualitative study of features in industrial software product lines. In *SPLC*, 2015.

[13] T. Berger, D. Nair, R. Rublack, J. M. Atlee, K. Czarnecki, and A. Wasowski. Three cases of feature-based variability modeling in industry. In *MODELS*, 2014.

[14] T. J. Biggerstaff, B. G. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *ICSE*, 1993.

[15] J. G. Carbonell, R. S. Michalski, and T. M. Mitchell. An overview of machine learning. In *Machine learning*, pages 3–23. Springer, 1983.

[16] B. Chandra and P. P. Varghese. Fuzzifying gini index based decision trees. *Expert Systems with Applications*, 36(4):8549–8559, 2009.

[17] K. Chen and V. Rajich. Ripples: tool for change in legacy software. In *Software Maintenance, 2001. Proceedings. IEEE International Conference on*, pages 230–239. IEEE, 2001.

[18] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[19] M. Dash and H. Liu. Feature selection for classification. *Intelligent data analysis*, 1(1-4):131–156, 1997.

[20] 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.

[21] D. Edwards, S. Simmons, and N. Wilde. An approach to feature location in distributed systems. *Journal of Systems and Software*, 79(1):57–68, 2006.

[22] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Transactions on software engineering*, 29(3):210–224, 2003.

[23] A. D. Eisenberg and K. De Volder. Dynamic feature traces: Finding features in unfamiliar code. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 337–346. IEEE, 2005.

[24] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software – Practice and Experience*, 30(11):1203–1233, 2000.

[25] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[26] J. L. C. Izquierdo, V. Cosentino, B. Rolandi, A. Bergel, and J. Cabot. Gila: Github label analyzer. In *SANER*, 2015.

[27] W. Ji, T. Berger, M. Antkiewicz, and K. Czarnecki. Maintaining feature traceability with embedded annotations. In *SPLC*, 2015.

[28] C. Kästner. CIDE: decomposing legacy applications into features. In *SPLC (2)*, pages 149–150, 2007.

[29] C. Kästner, T. Thum, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel. FeatureIDE: A tool framework for feature-oriented software development. In *ICSE*, 2009.

[30] C. W. Krueger. Easing the transition to software mass customization. In *PFE'01*, 2001.

[31] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *ICSE*, 2010.

[32] K. Lukoit, N. Wilde, S. Stowell, and T. Hennessey. Tracegraph: Immediate visual location of software features. In *icsm*, pages 33–39, 2000.

[33] D. P. Mandic and V. S. L. Goh. Adaptive and learning systems for signal, processing, communications, and control. *Complex Valued Nonlinear Adaptive Filters: Noncircularity, Widely Linear and Neural Models*, pages 325–325, 2001.

[34] A. Marcus, V. Rajlich, J. Buchta, M. Petrenko, and A. Sergeyev. Static techniques for concept location in object-oriented code. In *Program Comprehension,*

2005. IWPC 2005. Proceedings. 13th International Workshop on, pages 33–42. IEEE, 2005.

[35] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 214–223. IEEE, 2004.

[36] H. A. Müller, S. R. Tilley, M. A. Orgun, B. Corrie, and N. H. Madhavji. A reverse engineering environment based on spatial and visual software interconnection models. In *ACM SIGSOFT Software Engineering Notes*, volume 17, pages 88–98. ACM, 1992.

[37] G. Navarro. Nr-grep: a fast and flexible pattern-matching tool. *Software: Practice and Experience*, 31(13):1265–1312, 2001.

[38] L. Passos, K. Czarnecki, S. Apel, A. Wąsowski, C. Kästner, and J. Guo. Feature-oriented software evolution. In *VaMoS*, 2013.

[39] S. Paul, A. Prakash, E. Buss, and J. Henshaw. Theories and techniques of program understanding. In *Proceedings of the 1991 conference of the Centre for Advanced Studies on Collaborative research*, pages 37–53. IBM Press, 1991.

[40] T. N. Phyu. Survey of classification techniques in data mining. In *Proceedings of the International MultiConference of Engineers and Computer Scientists*, volume 1, pages 18–20, 2009.

[41] A. Pleuss and G. Botterweck. Visualization of variability and configuration options. *International Journal on Software Tools for Technology Transfer*, 14(5):497–510, 2012.

[42] K. Pohl, G. Böckle, and F. J. v. d. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[43] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 33(6):420–432, June 2007.

[44] V. Rajlich. A methodology for incremental changes, 2002.

[45] M. P. Robillard. Automatic generation of suggestions for program investigation. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 11–20. ACM, 2005.

[46] M. P. Robillard and G. C. Murphy. Feat: a tool for locating, describing, and analyzing concerns in source code. In *Proceedings of the 25th International Conference on Software Engineering*, pages 822–823. IEEE Computer Society, 2003.

[47] M. P. Robillard and G. C. Murphy. Representing concerns in source code. *ACM Trans. Softw. Eng. Methodol.*, 16(1), Feb. 2007.

[48] J. Rubin and M. Chechik. A survey of feature location techniques. In *Domain Engineering*. 2013.

[49] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *Proceedings of the 6th international conference on Aspect-oriented software development*, pages 212–224. ACM, 2007.

[50] S. Simmons, D. Edwards, N. Wilde, J. Homan, and M. Groble. Industrial tools for the feature location problem: an exploratory study. *Journal of Software: Evolution and Process*, 18(6):457–474, 2006.

[51] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil. An examination of software engineering work practices. In *CASCON First Decade High Impact Papers*, pages 174–188. IBM Corp., 2010.

[52] G. Spanoudakis, A. S. d. Garcez, and A. Zisman. Revising rules to capture requirements traceability relations: A machine learning approach. In *SEKE*, pages 570–577, 2003.

[53] E. Spyromitros, G. Tsoumakas, and I. Vlahavas. An empirical study of lazy multilabel classification algorithms. In *Hellenic conference on Artificial Intelligence*, pages 401–406. Springer, 2008.

[54] P. Tonella and M. Ceccato. Aspect mining through the formal concept analysis of execution traces. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 112–121. IEEE, 2004.

[55] G. Tsoumakas and I. Katakis. Multi-label classification: An overview. *International Journal of Data Warehousing and Mining*, 3(3), 2006.

[56] N. Wilde, M. Buckellew, H. Page, and V. Rajlich. A case study of feature location in unstructured legacy fortran code. In *Software Maintenance and Reengineering, 2001. Fifth European Conference on*, pages 68–76. IEEE, 2001.

[57] N. Wilde and M. C. Scully. Software reconnaissance: mapping program features to code. *Journal of Software Maintenance: Research and Practice*, 7(1):49–62, 1995.

[58] W. E. Wong, S. S. Gokhale, J. R. Horgan, and K. S. Trivedi. Locating program features using execution slices. In *Application-Specific Systems and Software Engineering and Technology, 1999. ASSET'99. Proceedings. 1999 IEEE Symposium on*, pages 194–203. IEEE, 1999.

[59] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang. Sniafl: Towards a static noninteractive approach to feature location. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(2):195–226, 2006.

A

Experiment Results

A.1 Comparison of Source Code Property Sets

Commit Number	CTS_NEA_CL_SVM	CTS_NEA_SVM	CTS_SVM
3	1.00	1.00	0.75
5	0.94	0.94	0.43
6	0.69	0.69	0.37
7	0.49	0.47	0.20
9	1.00	0.77	0.35
10	1.00	1.00	0.35
11	1.00	0.94	0.35
12	1.00	1.00	0.37
13	1.00	0.99	0.34
14	1.00	1.00	0.35
16	1.00	0.99	0.30
17	0.99	1.00	0.20
18	0.98	0.94	0.32
20	1.00	0.82	0.38
21	1.00	1.00	0.34
22	1.00	0.95	0.36
23	1.00	1.00	0.34
24	1.00	0.84	0.37
25	0.98	0.98	0.36
26	1.00	0.95	0.33
27	0.96	0.98	0.34
28	0.97	0.80	0.32
29	1.00	1.00	0.30
30	1.00	1.00	0.35
31	1.00	1.00	0.29
32	1.00	0.99	0.42
33	1.00	1.00	0.46
34	0.99	0.99	0.50
35	1.00	1.00	0.41
36	1.00	1.00	0.71
37	1.00	1.00	0.53

38	1.00	1.00	0.52
39	0.99	0.99	0.60
40	0.97	0.92	0.52
41	1.00	1.00	0.61
42	1.00	1.00	0.55
43	1.00	1.00	0.68
44	1.00	1.00	0.54
45	1.00	1.00	0.68
46	1.00	1.00	0.66
47	1.00	1.00	0.55
48	1.00	1.00	0.67
49	1.00	1.00	0.55
50	1.00	1.00	0.54
51	1.00	1.00	0.59
52	1.00	1.00	0.59
53	0.99	0.99	0.45
54	1.00	1.00	0.64
55	1.00	1.00	0.61
56	0.95	0.96	0.41
57	1.00	0.94	0.42
58	1.00	1.00	0.41
59	1.00	1.00	0.42
60	1.00	1.00	0.39
61	1.00	0.95	0.40
62	0.98	0.98	0.55
63	0.98	0.97	0.51
64	1.00	0.96	0.52
65	1.00	0.99	0.36
66	0.96	0.83	0.24
67	0.99	0.97	0.47
68	0.98	0.98	0.46
69	1.00	0.96	0.50
70	1.00	0.99	0.51
71	1.00	1.00	0.51
72	0.99	0.98	0.55
73	1.00	1.00	0.66
74	1.00	1.00	0.52
75	1.00	1.00	0.51
76	1.00	1.00	0.49
77	1.00	1.00	0.62
78	1.00	0.98	0.56
79	1.00	1.00	0.70
80	1.00	0.97	0.39
81	0.98	0.98	0.64

82	1.00	1.00	0.62
83	1.00	1.00	0.67
84	1.00	0.99	0.52
85	1.00	1.00	0.51
86	1.00	1.00	0.67
87	1.00	0.95	0.52
88	1.00	1.00	0.68
89	1.00	0.95	0.51
90	1.00	1.00	0.66
91	1.00	0.95	0.52
92	1.00	1.00	0.51
93	1.00	1.00	0.50
94	1.00	1.00	0.64
95	1.00	0.97	0.49
96	1.00	1.00	0.71
97	1.00	0.95	0.41
98	1.00	1.00	0.67
99	1.00	0.95	0.50
100	1.00	1.00	0.48
101	1.00	1.00	0.48
102	1.00	1.00	0.67
103	1.00	0.94	0.48
104	1.00	1.00	0.48
105	1.00	1.00	0.48
106	1.00	1.00	0.48
107	1.00	1.00	0.41
108	1.00	0.98	0.47
109	1.00	1.00	0.66
110	1.00	0.95	0.45
111	1.00	1.00	0.67
112	1.00	0.94	0.47
113	1.00	1.00	0.47
114	1.00	1.00	0.67
115	1.00	0.96	0.46
116	1.00	1.00	0.45
117	1.00	1.00	0.66
118	1.00	0.94	0.45
119	1.00	1.00	0.46
120	1.00	0.95	0.44
121	1.00	1.00	0.45
122	1.00	1.00	0.67
123	1.00	0.97	0.43
124	1.00	1.00	0.66
125	1.00	0.93	0.39

126	1.00	1.00	0.69
127	1.00	0.94	0.44
128	1.00	1.00	0.44
129	1.00	1.00	0.67
130	1.00	0.95	0.43
131	1.00	1.00	0.68
132	1.00	0.92	0.42
133	1.00	1.00	0.43
134	1.00	1.00	0.44
135	0.95	0.95	0.42
136	1.00	1.00	0.37
137	1.00	1.00	0.70
138	1.00	0.96	0.37
139	1.00	1.00	0.39
140	1.00	0.94	0.38
141	1.00	1.00	0.71
142	1.00	0.94	0.38
143	1.00	1.00	0.71
144	1.00	0.95	0.39
145	1.00	1.00	0.41
146	1.00	0.81	0.41
147	0.94	0.94	0.36
148	0.94	0.96	0.31
149	0.86	0.89	0.21
150	0.97	0.94	0.39
151	0.97	0.97	0.67
152	1.00	0.95	0.95
153	1.00	1.00	0.93
154	1.00	1.00	0.93
155	1.00	1.00	0.95
156	1.00	1.00	0.93
157	1.00	1.00	0.91
158	0.99	0.99	0.87
159	1.00	1.00	0.88
160	0.94	0.94	0.74
161	1.00	1.00	0.83
162	0.85	0.85	0.55
163	0.96	0.98	0.40
164	1.00	0.97	0.39
165	0.95	0.96	0.35
166	1.00	1.00	0.46
167	0.76	0.76	0.33
168	1.00	1.00	0.64
169	1.00	0.99	0.33

170	1.00	1.00	0.58
171	0.98	0.88	0.32
172	0.96	0.95	0.30
173	1.00	1.00	0.65
174	0.97	0.92	0.28
175	0.99	0.99	0.45
176	0.98	0.96	0.30
177	0.99	0.99	0.32
178	1.00	0.93	0.30
179	1.00	1.00	0.64
180	1.00	0.91	0.32
181	1.00	1.00	0.31
182	1.00	0.98	0.32
183	0.99	0.99	0.31
184	0.99	0.99	0.30
185	1.00	1.00	0.29
186	1.00	1.00	0.51
187	0.99	0.95	0.29
188	1.00	0.99	0.30
189	1.00	1.00	0.30
190	1.00	1.00	0.49
191	1.00	0.96	0.29
192	1.00	1.00	0.29

Table A.1: Comparison of Source Code Property Sets

A.2 Comparison of Retraining Intervals

Commit Number	TAEC_SVM	TAE5C_SVM	TAE10C_SVM	TAE20C_SVM
3	1.00	1.00	1.00	1.00
5	0.94	0.46	0.46	0.46
6	0.69	0.67	0.67	0.67
7	0.49	0.39	0.39	0.39
9	1.00	0.99	0.41	0.41
10	1.00	1.00	0.34	0.34
11	1.00	1.00	0.35	0.35
12	1.00	0.99	0.42	0.42
13	1.00	1.00	1.00	0.36
14	1.00	1.00	1.00	0.36
16	1.00	0.96	0.96	0.30
17	0.99	0.93	0.93	0.26
18	0.98	0.98	0.94	0.37
20	1.00	1.00	0.95	0.37
21	1.00	0.98	0.94	0.36

22	1.00	0.98	0.94	0.36
23	1.00	1.00	1.00	1.00
24	1.00	1.00	1.00	1.00
25	0.98	0.98	0.98	0.98
26	1.00	0.98	0.98	0.98
27	0.96	0.94	0.94	0.94
28	0.97	0.97	1.00	1.00
29	1.00	0.97	1.00	1.00
30	1.00	0.97	0.97	0.97
31	1.00	0.97	1.00	1.00
32	1.00	0.97	0.97	0.97
33	1.00	1.00	1.00	0.95
34	0.99	0.97	0.97	0.94
35	1.00	1.00	1.00	0.95
36	1.00	0.75	0.75	0.71
37	1.00	0.98	0.98	0.93
38	1.00	1.00	0.98	0.92
39	0.99	1.00	0.86	0.85
40	0.97	0.97	0.93	0.89
41	1.00	0.92	0.77	0.73
42	1.00	1.00	0.98	0.93
43	1.00	1.00	1.00	1.00
44	1.00	0.99	0.99	0.99
45	1.00	1.00	1.00	1.00
46	1.00	1.00	1.00	1.00
47	1.00	0.99	0.99	0.99
48	1.00	1.00	1.00	1.00
49	1.00	1.00	0.99	0.99
50	1.00	1.00	0.99	0.99
51	1.00	1.00	1.00	1.00
52	1.00	1.00	1.00	1.00
53	0.99	0.99	0.99	0.99
54	1.00	0.99	0.99	0.99
55	1.00	0.99	0.99	0.99
56	0.95	0.94	0.94	0.95
57	1.00	0.95	0.95	0.91
58	1.00	1.00	0.96	0.92
59	1.00	0.99	0.94	0.91
60	1.00	1.00	0.90	0.85
61	1.00	0.98	0.90	0.86
62	0.98	0.97	0.92	0.89
63	0.98	0.98	0.98	0.98
64	1.00	0.98	0.98	0.98
65	1.00	0.96	0.96	0.96

66	0.96	0.90	0.90	0.90
67	0.99	0.94	0.94	0.94
68	0.98	0.98	0.95	0.95
69	1.00	0.98	0.94	0.94
70	1.00	0.98	0.94	0.94
71	1.00	0.98	0.94	0.94
72	0.99	0.98	0.93	0.93
73	1.00	1.00	1.00	1.00
74	1.00	1.00	1.00	0.91
75	1.00	1.00	1.00	0.91
76	1.00	1.00	1.00	0.91
77	1.00	1.00	1.00	1.00
78	1.00	1.00	0.97	0.89
79	1.00	1.00	1.00	1.00
80	1.00	0.97	0.92	0.77
81	0.98	0.94	0.94	0.97
82	1.00	1.00	1.00	1.00
83	1.00	1.00	1.00	1.00
84	1.00	1.00	1.00	1.00
85	1.00	1.00	1.00	1.00
86	1.00	1.00	1.00	1.00
87	1.00	0.99	0.99	0.99
88	1.00	1.00	1.00	1.00
89	1.00	0.99	0.98	0.98
90	1.00	1.00	1.00	1.00
91	1.00	0.99	0.98	0.98
92	1.00	0.99	0.98	0.98
93	1.00	1.00	1.00	0.98
94	1.00	1.00	1.00	1.00
95	1.00	0.99	0.99	0.97
96	1.00	1.00	1.00	1.00
97	1.00	0.99	0.99	0.96
98	1.00	1.00	1.00	1.00
99	1.00	0.98	0.98	0.96
100	1.00	0.98	0.98	0.96
101	1.00	0.98	0.98	0.96
102	1.00	1.00	1.00	1.00
103	1.00	1.00	1.00	1.00
104	1.00	1.00	1.00	1.00
105	1.00	1.00	1.00	1.00
106	1.00	1.00	1.00	1.00
107	1.00	1.00	1.00	1.00
108	1.00	1.00	1.00	1.00
109	1.00	1.00	1.00	1.00

110	1.00	0.98	0.98	0.98
111	1.00	1.00	1.00	1.00
112	1.00	0.96	0.96	0.96
113	1.00	1.00	1.00	0.96
114	1.00	1.00	1.00	1.00
115	1.00	0.99	0.99	0.95
116	1.00	0.99	0.99	0.95
117	1.00	1.00	1.00	1.00
118	1.00	1.00	0.97	0.93
119	1.00	1.00	0.98	0.94
120	1.00	1.00	0.97	0.93
121	1.00	1.00	0.97	0.93
122	1.00	1.00	1.00	1.00
123	1.00	1.00	1.00	1.00
124	1.00	1.00	1.00	1.00
125	1.00	0.98	0.98	0.98
126	1.00	1.00	1.00	1.00
127	1.00	0.98	0.98	0.98
128	1.00	1.00	0.98	0.98
129	1.00	1.00	1.00	1.00
130	1.00	0.98	0.95	0.95
131	1.00	1.00	1.00	1.00
132	1.00	0.96	0.94	0.94
133	1.00	1.00	1.00	0.94
134	1.00	1.00	1.00	0.94
135	0.95	0.95	0.95	0.89
136	1.00	0.95	0.95	0.88
137	1.00	1.00	1.00	1.00
138	1.00	1.00	0.95	0.89
139	1.00	1.00	0.94	0.87
140	1.00	1.00	0.94	0.88
141	1.00	1.00	1.00	1.00
142	1.00	0.97	0.92	0.85
143	1.00	1.00	1.00	1.00
144	1.00	1.00	1.00	1.00
145	1.00	1.00	1.00	1.00
146	1.00	1.00	1.00	1.00
147	0.94	0.95	0.95	0.95
148	0.94	0.94	0.96	0.96
149	0.86	0.87	0.87	0.87
150	0.97	0.83	0.84	0.84
151	0.97	0.97	0.97	0.97
152	1.00	1.00	1.00	1.00
153	1.00	1.00	1.00	0.97

154	1.00	1.00	1.00	0.97
155	1.00	1.00	1.00	1.00
156	1.00	1.00	1.00	0.97
157	1.00	1.00	1.00	0.97
158	0.99	0.99	0.99	0.99
159	1.00	1.00	1.00	0.97
160	0.94	0.94	0.94	0.94
161	1.00	0.95	0.95	0.97
162	0.85	0.81	0.81	0.85
163	0.96	0.96	0.96	0.96
164	1.00	0.71	0.71	0.71
165	0.95	0.75	0.75	0.75
166	1.00	0.76	0.76	0.76
167	0.76	0.53	0.53	0.53
168	1.00	1.00	0.79	0.79
169	1.00	0.85	0.74	0.74
170	1.00	0.76	0.54	0.54
171	0.98	0.87	0.72	0.72
172	0.96	0.82	0.64	0.64
173	1.00	1.00	1.00	0.60
174	0.97	0.95	0.95	0.71
175	0.99	0.91	0.91	0.55
176	0.98	0.94	0.94	0.73
177	0.99	0.94	0.94	0.60
178	1.00	1.00	0.94	0.74
179	1.00	1.00	1.00	0.60
180	1.00	0.99	0.95	0.72
181	1.00	1.00	0.98	0.68
182	1.00	0.98	0.94	0.62
183	0.99	0.99	0.99	0.99
184	0.99	0.99	0.99	0.99
185	1.00	1.00	1.00	1.00
186	1.00	0.98	0.98	0.98
187	0.99	0.99	0.99	0.99
188	1.00	1.00	0.99	0.99
189	1.00	1.00	0.99	0.99
190	1.00	1.00	0.97	0.97
191	1.00	0.98	0.97	0.97
192	1.00	0.98	0.97	0.97

Table A.2: Comparison of Retraining Intervals

A.3 Comparison of Source Code Granularity

Commit Number	TAEC$_{LoC_SVM}$	TAEC$_{File_SVM}$	TAEC$_{Folder_SVM}$
3	1.00	0.00	0.67
4	0.94	0.00	0.00
5	0.69	0.22	0.00
6	0.49	0.81	0.00
7	1.00	0.55	0.00
8	1.00	0.00	0.67
9	1.00	0.61	0.38
10	1.00	0.96	0.48
11	1.00	0.87	0.41
12	1.00	0.86	0.36
13	1.00	0.89	0.41
14	0.99	0.93	0.41
15	0.98	0.00	0.00
16	1.00	0.74	0.44
17	1.00	0.00	0.00
18	1.00	0.78	0.46
19	1.00	0.00	0.00
20	1.00	0.86	0.36
21	0.98	0.92	0.47
22	1.00	0.92	0.50
23	0.96	0.93	0.00
24	0.97	0.91	0.39
25	1.00	0.89	0.45
26	1.00	0.89	0.34
27	1.00	0.77	0.44
28	1.00	0.50	0.42
29	1.00	0.82	0.39
30	0.99	0.84	0.42
31	1.00	0.76	0.39
32	1.00	0.85	0.42
33	1.00	0.89	0.45
34	1.00	0.54	0.42
35	0.99	0.83	0.45
36	0.97	0.00	0.62
37	1.00	0.93	0.45
38	1.00	0.90	0.45
39	1.00	0.00	0.62
40	1.00	0.62	0.50
41	1.00	0.00	0.62
42	1.00	0.73	0.45
43	1.00	0.00	0.62
44	1.00	0.69	0.49
45	1.00	0.00	0.62

46	1.00	0.00	0.62
47	1.00	0.67	0.49
48	1.00	0.91	0.62
49	0.99	0.63	0.49
50	1.00	0.89	0.49
51	1.00	0.00	0.62
52	0.95	0.91	0.55
53	1.00	0.87	0.52
54	1.00	0.90	0.57
55	1.00	0.93	0.57
56	1.00	0.67	0.58
57	1.00	0.67	0.60
58	0.98	0.83	0.57
59	0.98	0.84	0.59
60	1.00	0.00	0.62
61	1.00	0.86	0.59
62	0.96	0.56	0.43
63	0.99	0.71	0.57
64	0.98	0.75	0.70
65	1.00	0.78	0.56
66	1.00	0.77	0.36
67	1.00	0.81	0.49
68	0.99	0.48	0.60
69	1.00	0.89	0.67
70	1.00	0.88	0.54
71	1.00	0.88	0.67
72	1.00	0.83	0.56
73	1.00	0.00	0.62
74	1.00	0.90	0.67
75	1.00	0.85	0.56
76	1.00	0.87	0.56
77	0.98	0.00	0.67
78	1.00	0.82	0.53
79	1.00	0.00	0.67
80	1.00	0.68	0.42
81	1.00	0.00	0.67
82	1.00	0.00	0.62
83	1.00	0.00	0.67
84	1.00	0.83	0.50
85	1.00	0.78	0.47
86	1.00	0.00	0.63
87	1.00	0.77	0.40
88	1.00	0.00	0.63
89	1.00	0.00	0.43

90	1.00	0.00	0.63
91	1.00	0.76	0.43
92	1.00	0.84	0.40
93	1.00	0.88	0.38
94	1.00	0.00	0.67
95	1.00	0.71	0.43
96	1.00	0.00	0.67
97	1.00	0.75	0.42
98	1.00	0.00	0.75
99	1.00	0.70	0.42
100	1.00	0.85	0.42
101	1.00	0.85	0.42
102	1.00	0.00	0.67
103	1.00	0.71	0.41
104	1.00	0.85	0.41
105	1.00	0.84	0.41
106	1.00	0.85	0.41
107	1.00	0.85	0.41
108	1.00	0.85	0.67
109	1.00	0.00	0.67
110	1.00	0.70	0.40
111	1.00	0.00	0.67
112	1.00	0.74	0.67
113	1.00	0.86	0.39
114	1.00	0.00	0.67
115	1.00	0.00	0.39
116	1.00	0.74	0.39
117	1.00	0.00	0.67
118	1.00	0.73	0.38
119	1.00	0.95	0.67
120	1.00	0.87	0.38
121	1.00	0.87	0.38
122	1.00	0.00	0.67
123	1.00	0.87	0.38
124	1.00	0.00	0.67
125	1.00	0.88	0.38
126	1.00	0.00	0.67
127	1.00	0.88	0.38
128	1.00	0.88	0.38
129	1.00	0.00	0.67
130	1.00	0.88	0.37
131	0.95	0.00	0.67
132	1.00	0.91	0.37
133	1.00	0.90	0.37

134	1.00	0.90	0.37
135	1.00	0.82	0.37
136	1.00	0.89	0.37
137	1.00	0.00	0.67
138	1.00	0.80	0.37
139	1.00	0.94	0.67
140	1.00	0.89	0.40
141	1.00	0.00	0.67
142	1.00	0.80	0.39
143	0.94	0.00	0.67
144	0.94	0.71	0.35
145	0.86	0.93	0.67
146	0.97	0.86	0.36
147	0.97	0.81	0.30
148	1.00	0.76	0.55
149	1.00	0.66	0.54
150	1.00	0.84	0.55
151	1.00	0.00	0.59
152	1.00	0.00	0.70
153	1.00	0.00	0.76
154	0.99	0.00	0.76
155	1.00	0.00	0.70
156	0.94	0.75	0.76
157	1.00	0.64	0.73
158	0.85	0.00	0.73
159	0.96	0.00	0.79
160	1.00	0.00	0.73
161	0.95	0.69	0.63
162	1.00	0.16	0.56
163	0.76	0.74	0.00
164	1.00	0.75	0.52
165	1.00	0.77	0.78
166	1.00	0.66	0.76
167	0.98	0.58	0.67
168	0.96	0.40	0.67
169	1.00	0.77	0.74
170	0.97	0.62	0.62
171	0.99	0.78	0.76
172	0.98	0.75	0.00
173	0.99	0.00	0.00
174	1.00	0.63	0.00
175	1.00	0.70	0.00
176	1.00	0.79	0.00
177	1.00	0.75	0.00

178	1.00	0.81	0.00
179	0.99	0.78	0.00
180	0.99	0.80	0.00
181	1.00	0.79	0.00
182	1.00	0.80	0.00
183	0.99	0.79	0.00
184	1.00	0.79	0.00
185	1.00	0.75	0.00
186	1.00	0.68	0.00
187	1.00	0.73	0.00
188	1.00	0.75	0.00
189	1.00	0.83	0.00
190	1.00	0.78	0.00
191	1.00	0.82	0.00
192	1.00	0.76	0.00
193	1.00	0.76	0.00
194	1.00	0.81	0.00
195	1.00	0.82	0.00

Table A.3: Comparison of Source Granularity

A.4 Comparison of Machine Learing Alrogrithms

Commit Number	TAEC SVM	TAEC Decision Tree	TAEC MLALGOKNearest Neighbors	TAEC MLALGo Random Forest	TAEC MLALGO Naive Bayes	TAEC MLALGO Bootstrap Aggregation	TAEC MLALGO Stacked Aggregation
3	1.00	1.00	1.00	1.00	0.03	1.00	0.78
5	0.94	0.83	0.24	0.88	0.80	0.83	0.35
6	0.69	0.43	0.05	0.22	0.27	0.69	0.13
7	0.49	0.37	0.09	0.34	0.33	0.37	0.10
9	1.00	0.81	0.11	0.60	0.67	1.00	0.10
10	1.00	1.00	0.19	0.99	0.85	0.94	0.25
11	1.00	1.00	0.18	1.00	0.86	1.00	0.25
12	1.00	1.00	0.11	0.60	0.71	1.00	0.15
13	1.00	1.00	0.18	0.99	0.74	1.00	0.24
14	1.00	1.00	0.18	1.00	0.84	1.00	0.24
16	1.00	1.00	1.00	0.99	0.87	1.00	0.24
17	0.99	0.98	0.99	0.81	0.95	0.98	0.33
18	0.98	0.98	0.98	0.98	0.87	0.98	0.25
20	1.00	1.00	0.11	0.70	0.70	1.00	0.11
21	1.00	1.00	1.00	0.85	0.94	1.00	0.36

22	1.00	1.00	0.20	0.98	0.89	0.97	0.25
23	1.00	1.00	0.21	0.98	0.89	0.91	0.25
24	1.00	1.00	0.12	0.56	0.69	0.82	0.12
25	0.98	0.98	0.21	0.94	0.87	0.89	0.25
26	1.00	1.00	1.00	0.91	0.92	1.00	0.27
27	0.96	0.95	0.12	0.85	0.83	0.86	0.20
28	0.97	0.68	0.10	0.64	0.67	0.62	0.11
29	1.00	1.00	0.11	0.67	0.69	0.82	0.11
30	1.00	1.00	0.20	0.99	0.88	0.91	0.24
31	1.00	1.00	0.11	0.44	0.64	0.82	0.11
32	1.00	1.00	0.20	0.95	0.85	1.00	0.25
33	1.00	1.00	1.00	0.78	0.94	1.00	0.38
34	0.99	0.98	0.97	0.88	0.90	0.98	0.38
35	1.00	1.00	1.00	0.93	0.96	1.00	0.46
36	1.00	0.82	1.00	0.95	0.79	0.82	0.75
37	1.00	1.00	1.00	0.99	0.96	1.00	0.46
38	1.00	1.00	1.00	0.98	0.96	1.00	0.47
39	0.99	0.98	0.21	0.94	0.93	0.98	0.62
40	0.97	0.97	0.96	0.96	0.90	0.96	0.49
41	1.00	1.00	0.18	0.84	0.72	1.00	0.67
42	1.00	1.00	1.00	0.99	0.96	1.00	0.47
43	1.00	1.00	0.19	0.72	0.79	1.00	0.67
44	1.00	1.00	0.22	0.99	0.94	1.00	0.49
45	1.00	1.00	0.19	0.85	0.83	1.00	0.67
46	1.00	1.00	0.19	0.70	0.83	1.00	0.67
47	1.00	1.00	0.22	0.95	0.93	1.00	0.49
48	1.00	1.00	0.19	0.71	0.83	1.00	0.67
49	1.00	1.00	0.22	0.94	0.84	1.00	0.49
50	1.00	1.00	0.22	0.94	0.91	1.00	0.49
51	1.00	1.00	0.19	0.91	0.71	1.00	0.77
52	1.00	1.00	0.19	0.96	0.71	1.00	0.77
53	0.99	0.99	0.99	0.94	0.84	0.99	0.48
54	1.00	1.00	0.22	0.92	0.93	1.00	0.54
55	1.00	1.00	1.00	1.00	0.93	1.00	0.51
56	0.95	0.81	0.07	0.40	0.53	0.81	0.32
57	1.00	0.87	0.16	0.88	0.81	0.84	0.43
58	1.00	1.00	1.00	0.88	0.92	1.00	0.45
59	1.00	1.00	0.18	0.98	0.77	1.00	0.42
60	1.00	1.00	0.94	0.71	0.78	1.00	0.45
61	1.00	1.00	1.00	0.94	0.78	1.00	0.39
62	0.98	0.98	0.16	0.83	0.73	0.91	0.41
63	0.98	0.99	1.00	1.00	0.83	0.94	0.34
64	1.00	1.00	1.00	0.98	0.76	0.86	0.36
65	1.00	0.67	0.06	0.65	0.73	0.80	0.25

66	0.96	0.93	0.08	0.84	0.71	0.93	0.23
67	0.99	0.66	0.10	0.78	0.75	0.78	0.33
68	0.98	0.98	0.12	0.95	0.68	0.98	0.38
69	1.00	0.99	1.00	0.97	0.71	0.90	0.42
70	1.00	1.00	1.00	0.99	0.70	1.00	0.41
71	1.00	1.00	1.00	0.97	0.73	1.00	0.41
72	0.99	0.84	0.08	0.98	0.68	0.84	0.41
73	1.00	1.00	1.00	0.77	0.78	1.00	0.91
74	1.00	1.00	1.00	0.97	0.73	1.00	0.41
75	1.00	1.00	1.00	0.99	0.70	1.00	0.40
76	1.00	1.00	1.00	0.99	0.71	1.00	0.40
77	1.00	1.00	0.59	0.72	0.84	1.00	0.87
78	1.00	0.99	1.00	0.98	0.73	0.99	0.40
79	1.00	1.00	0.60	0.68	0.85	1.00	0.86
80	1.00	0.83	0.07	0.83	0.70	1.00	0.23
81	0.98	0.98	0.98	0.82	0.80	0.98	0.79
82	1.00	1.00	1.00	0.73	0.82	1.00	0.83
83	1.00	1.00	1.00	0.71	0.80	1.00	0.87
84	1.00	1.00	1.00	0.98	0.68	1.00	0.37
85	1.00	1.00	1.00	0.97	0.71	1.00	0.37
86	1.00	1.00	1.00	0.76	0.80	1.00	0.87
87	1.00	1.00	1.00	0.99	0.71	1.00	0.37
88	1.00	1.00	1.00	0.73	0.81	1.00	0.87
89	1.00	1.00	1.00	0.98	0.71	1.00	0.37
90	1.00	1.00	1.00	0.78	0.80	1.00	0.87
91	1.00	1.00	1.00	0.98	0.72	1.00	0.37
92	1.00	1.00	1.00	0.98	0.71	1.00	0.37
93	1.00	1.00	1.00	0.98	0.69	1.00	0.37
94	1.00	1.00	1.00	0.76	0.81	1.00	0.83
95	1.00	1.00	1.00	0.98	0.73	1.00	0.37
96	1.00	1.00	1.00	0.78	0.81	1.00	0.91
97	1.00	0.83	0.07	0.91	0.66	1.00	0.25
98	1.00	1.00	1.00	0.75	0.81	1.00	0.87
99	1.00	1.00	1.00	0.98	0.76	1.00	0.35
100	1.00	1.00	1.00	0.99	0.76	1.00	0.34
101	1.00	1.00	1.00	0.98	0.71	1.00	0.34
102	1.00	1.00	1.00	0.73	0.82	1.00	0.87
103	1.00	1.00	1.00	0.98	0.76	1.00	0.34
104	1.00	1.00	1.00	0.98	0.76	1.00	0.34
105	1.00	1.00	1.00	0.99	0.72	1.00	0.34
106	1.00	1.00	1.00	0.98	0.72	1.00	0.34
107	1.00	0.85	0.08	0.91	0.67	1.00	0.23
108	1.00	1.00	1.00	0.99	0.71	1.00	0.33
109	1.00	1.00	1.00	0.69	0.84	1.00	0.87

110	1.00	1.00	1.00	0.98	0.77	1.00	0.32
111	1.00	1.00	1.00	0.78	0.83	1.00	0.87
112	1.00	1.00	1.00	0.99	0.79	1.00	0.30
113	1.00	1.00	1.00	0.98	0.79	1.00	0.30
114	1.00	1.00	1.00	0.73	0.83	1.00	0.87
115	1.00	1.00	1.00	0.99	0.80	1.00	0.29
116	1.00	1.00	1.00	0.98	0.80	1.00	0.29
117	1.00	1.00	1.00	0.74	0.83	1.00	0.87
118	1.00	1.00	1.00	0.98	0.80	1.00	0.29
119	1.00	0.87	0.10	1.00	0.80	1.00	0.30
120	1.00	1.00	1.00	0.98	0.81	1.00	0.29
121	1.00	1.00	1.00	0.99	0.82	1.00	0.29
122	1.00	1.00	1.00	0.83	0.83	1.00	0.87
123	1.00	1.00	1.00	0.98	0.81	1.00	0.28
124	1.00	1.00	1.00	0.79	0.83	1.00	0.87
125	1.00	0.87	0.10	0.99	0.82	1.00	0.20
126	1.00	1.00	1.00	0.83	0.84	1.00	0.87
127	1.00	1.00	1.00	0.98	0.82	1.00	0.27
128	1.00	1.00	1.00	0.99	0.84	1.00	0.26
129	1.00	1.00	1.00	0.76	0.84	1.00	0.87
130	1.00	1.00	1.00	0.98	0.84	1.00	0.25
131	1.00	1.00	1.00	0.86	0.84	1.00	0.87
132	1.00	1.00	1.00	0.98	0.86	0.93	0.27
133	1.00	1.00	1.00	0.99	0.87	1.00	0.27
134	1.00	1.00	1.00	0.99	0.83	1.00	0.27
135	0.95	0.83	0.10	0.95	0.75	0.95	0.28
136	1.00	1.00	1.00	0.99	0.79	1.00	0.27
137	1.00	1.00	1.00	0.74	0.84	1.00	0.80
138	1.00	1.00	1.00	0.99	0.84	1.00	0.27
139	1.00	0.87	0.10	0.99	0.82	1.00	0.28
140	1.00	1.00	1.00	0.99	0.83	1.00	0.27
141	1.00	1.00	1.00	0.75	0.84	1.00	0.80
142	1.00	1.00	1.00	0.99	0.87	1.00	0.23
143	1.00	1.00	1.00	0.83	0.85	1.00	0.21
144	1.00	1.00	1.00	0.85	0.90	1.00	0.23
145	1.00	1.00	1.00	0.96	0.91	0.96	0.38
146	1.00	1.00	1.00	0.96	0.91	1.00	0.37
147	0.94	0.82	0.10	0.78	0.78	0.94	0.25
148	0.94	0.95	0.96	0.95	0.75	0.95	0.21
149	0.86	0.88	0.86	0.82	0.70	0.88	0.26
150	0.97	0.97	0.97	0.94	0.73	0.97	0.28
151	0.97	0.97	0.97	0.96	0.77	0.97	0.85
152	1.00	1.00	1.00	0.90	0.84	1.00	0.95
153	1.00	1.00	1.00	0.94	0.67	1.00	0.93

154	1.00	1.00	1.00	0.95	0.68	1.00	0.93
155	1.00	1.00	1.00	0.91	0.64	1.00	0.95
156	1.00	1.00	1.00	0.93	0.69	1.00	0.93
157	1.00	1.00	1.00	0.93	0.69	1.00	0.93
158	0.99	0.99	0.99	0.87	0.67	0.99	0.91
159	1.00	1.00	1.00	0.91	0.70	1.00	0.86
160	0.94	0.94	0.94	0.91	0.62	0.94	0.78
161	1.00	1.00	1.00	0.92	0.68	0.96	0.77
162	0.85	0.85	0.12	0.79	0.61	0.81	0.52
163	0.96	0.76	0.99	0.95	0.57	0.77	0.42
164	1.00	1.00	1.00	0.97	0.82	1.00	0.40
165	0.95	0.90	0.96	0.94	0.73	0.89	0.34
166	1.00	1.00	0.99	0.96	0.83	1.00	0.42
167	0.76	0.76	0.12	0.63	0.64	0.76	0.33
168	1.00	1.00	1.00	0.79	0.86	1.00	0.84
169	1.00	0.98	1.00	0.98	0.85	0.99	0.27
170	1.00	1.00	0.99	0.99	0.77	1.00	0.48
171	0.98	0.98	0.98	0.97	0.88	0.98	0.27
172	0.96	0.96	0.12	0.94	0.82	0.96	0.25
173	1.00	1.00	1.00	0.86	0.85	1.00	0.84
174	0.97	0.98	0.97	0.95	0.89	0.96	0.24
175	0.99	0.94	0.12	0.97	0.77	0.94	0.39
176	0.98	1.00	1.00	0.97	0.87	0.99	0.27
177	0.99	0.99	0.99	0.84	0.90	0.97	0.37
178	1.00	1.00	1.00	0.96	0.90	0.99	0.27
179	1.00	1.00	1.00	0.84	0.94	1.00	0.84
180	1.00	1.00	1.00	0.96	0.91	1.00	0.29
181	1.00	1.00	1.00	0.96	0.89	0.96	0.29
182	1.00	1.00	1.00	0.96	0.87	1.00	0.30
183	0.99	0.99	0.99	0.97	0.85	0.99	0.30
184	0.99	0.99	0.99	0.97	0.84	0.99	0.27
185	1.00	1.00	1.00	0.93	0.87	1.00	0.30
186	1.00	1.00	1.00	0.90	0.84	1.00	0.58
187	0.99	0.99	0.99	0.97	0.87	0.99	0.26
188	1.00	1.00	1.00	0.97	0.87	0.97	0.29
189	1.00	1.00	1.00	0.97	0.88	1.00	0.29
190	1.00	1.00	1.00	0.97	0.84	1.00	0.52
191	1.00	1.00	1.00	0.97	0.88	0.99	0.30
192	1.00	1.00	1.00	0.97	0.87	0.97	0.31

Table A.4: Comparison of Machine Learning Algrithms