FeatRacer: Locating Features Through Assisted Traceability

Mukelabai Mukelabai, Kevin Hermann, Thorsten Berger, Jan-Philipp Steghöfer

Abstract—Locating features is one of the most common software development activities. It is typically done during maintenance and evolution, when developers need to identify the exact places in a codebase where specific features are implemented. Unfortunately, locating features is laborious and error-prone, since feature knowledge fades, projects are developed by different developers, and features are often scattered across the codebase. Recognizing the need, many *automated feature location techniques* have been proposed, which try to retroactively recover features, i,.e., very domain-specific information from the codebase. Unfortunately, such techniques require large training datasets, only recover coarse-grained locations and produce too many false positives to be useful in practice. An alternative is *recording features during development*, when they are still fresh in a developer's mind. However, recording is easily forgotten and also costly, especially when the software evolves and such recordings need to be updated. We address the infamous *feature location problem* (a.k.a., *concern location* or *concept assignment problem*) differently. We present FeatRacer, which combines feature recording and automated feature location in a way that allows developers to proactively and continuously record features and their locations during development, while addressing the shortcomings of both strategies. Specifically, FeatRacer relies on embedded code annotations and a machine-learning-based recommender system. When a developer forgets to annotate, FeatRacer reminds the developer about potentially missing features, which it learned from the feature recording practices in the project at hand. FeatRacer also facilitates fine-grained locations as decided by the developer.
Our evaluation shows that FeatRacer outperforms traditional automated feature location based on Latent Semantic Indexing (LSI) and Linear Discriminant Analysis (LDA)—two of the most common methods to realize such techniques—when predicting features for 4,650 commit changesets from the histories of 16 open-source projects spanning an average of three years between 1985 and 2015. Compared to the traditional techniques, FeatRacer showed a 3x higher precision and a 4.5x higher recall, with an average precision and recall of 89.6 % among all 16 projects. It can accurately predict feature locations within the first five commits of our evaluation projects, being effective already for small datasets. FeatRacer takes on average 1.9ms to learn from past code fragments of a project, and 0.002ms to predict forgotten feature annotations in new code.

Index Terms—feature location, traceability, recommender

1 Introduction

Software systems are often designed and developed around the notion of *features* [1], [2]. Stakeholders use features to define, manage, and communicate the functional and non-functional aspects of a system. Almost all modern and agile methods, such as SCRUM, XP, and FDD, rely on features to organize processes, software releases [3], and teams. Moreover, when engineering a variant-rich system—realized via forks or branches, or via a configurable software platform (a.k.a., software product line [4]–[7])—features provide an intuitive way to reason about and manage the variability of the system [2], [8]–[13].

Reusing, maintaining, and evolving features requires knowing their locations in software assets (e.g., folders, files, or file fragments). Locating features is in fact one of the most common activities of developers [14]–[17]. However, it is also a challenging and tedious activity, especially when features are cross-cutting the codebase or when they have been implemented a long time ago, possibly by another developer. To this end, the research community has focused on this *feature location* [17] or *concept assignment problem* [16] for decades and proposed a large range of manual [18] and automated feature-location techniques [17]. Manual techniques guide developers with strategies to systematically explore code and its dependencies around an entry point and then validate the result, e.g., via debugging [14]. However, manual feature location is still laborious and error-prone—a survey [14] reports an average of 15 minutes to locate a feature in systems with 73k lines of code or less.

Automated techniques try to identify the software assets (files or methods) that belong to a particular feature using natural language processing, machine learning, or other reasoning or search-based techniques. They classify into static and dynamic techniques. Static techniques typically require providing features and feature descriptions or formulating search queries for the feature of interest—which is then matched against assets to find those that implement the feature. Information retrieval techniques, such as *term frequency-inverse document frequency* (tf-idf) and *Latent Semantic Indexing* (LSI) [19] are most commonly used. Some techniques also require providing an entry point (e.g., a menu entry) from where they can explore the code (e.g., by following dependencies). Dynamic techniques require executing the system and exercising the feature. In this case, the developer needs to specify entry points or write test cases that exercise the desired feature. Finally, both static and dynamic techniques are typically difficult to setup for the project at hand.

Studies also confirm that automated techniques exhibit low precision [17], [20] or require significant effort to improve the precision [17]. The fact that features are very domain-specific entities (developers and projects often use different notions) that are often cross-cutting and vary highly in their granularity, further explains the uphill battle traditional automated techniques have faced. As such, the lack in adoption of automated feature location techniques is not surprising.

We address the feature location problem differently: instead of retroactively retrieving feature locations long after their implementation, we propose FeatRacer as a novel technique to encourage developers to continuously record feature locations during development when the knowledge about features is still fresh in their minds. FeatRacer focuses on supporting developers to record the features they currently implement, while allowing to record more fine-grained and accurate locations with reasonable effort.

The novelty, giving FeatRacer an advantage over existing techniques, relies on two main ideas. First, FeatRacer builds on the assumption that software assets implementing the same features share similar characteristics, and that one can learn with machine-learning algorithms what these similarities are per feature. This allows FeatRacer to learn the project- or developer-specific notion of features for the current project. To support this assumption, we investigated how different development-process-related metrics affect the accuracy in feature location. Second, the continuous recording of accurate and fine-grained feature locations with embedded code annotations allows FeatRacer to create very detailed training data for machine-learning models. These models rely on metrics we designed to measure the relatedness of software assets, independent of code structure and programming language. The trained models allow FeatRacer to make predictions upon every commit that, as we will show, easily outperform traditional automated techniques relying on LSI or LDA.

Developers use FeatRacer as follows. When writing code, they are asked to immediately record feature locations with embedded annotations: programming-language-independent code comments with feature labels. Once developers added the first annotations, FeatRacer can learn from these annotations and nudge developers by reminding them (e.g., on commit) when they forgot to annotate software assets belonging to features. Developers can then accept or reject the recommendation. Then, when developers need to locate features, they can immediately retrieve the locations from the recordings, looking up the feature name.

In comparison to automated techniques, this kind of feature location is easier, more intuitive, and more accurate. In addition, it requires less effort from developers. Automated techniques are not only much less accurate, but require writing complex descriptions of the features, easily overwhelming developers both in the input required and the output to analyze. A practical feature location technique must adapt as closely as possible to the developer's workflow and development process. In that regard, we show that FeatRacer in fact provides better predictions of feature locations during each commit than traditional (LSI and LDA-based) feature-location techniques. Our evaluation, where we simulated the development of sixteen projects that have feature annotations embedded in the source code, shows that compared to LSI, FeatRacer had 3x higher precision (average of 89.6 % across all projects, while LSI had 33.8 %) and 4.5x higher recall (average of 89.6 % from all projects while LSI had 20.5 %). LDA on the other hand consistently failed to locate features with an average precision of 0.9 % and recall of 0.8 %. FeatRacer also shows a good performance quickly after a developer started adding annotations, addressing the 'cold start' problem efficiently.

In summary, we propose the technique FeatRacer. It relies on embedded feature annotations to let developers record features in a lightweight way during development and, if necessary, nudges developers to *proactively* and *continuously* record feature locations during development. It does so by recommending features for assets for which they might have forgotten to record features. We engineered FeatRacer as a multi-label feature-location recommender system, relying on state-of-the-art multi-label learning algorithms. We designed asset metrics (which mainly measure development-process-related aspects) that are independent of the code structure and the programming language to improve FeatRacer's predictive ability. We evaluated FeatRacer on 16 open-source projects from different domains, all using feature annotations embedded within source code, exhibiting varying granularities of those annotations. We compared FeatRacer's performance to that of LSI and LDA, two of the most widely used techniques integrated in many automated feature location techniques [17], [21].

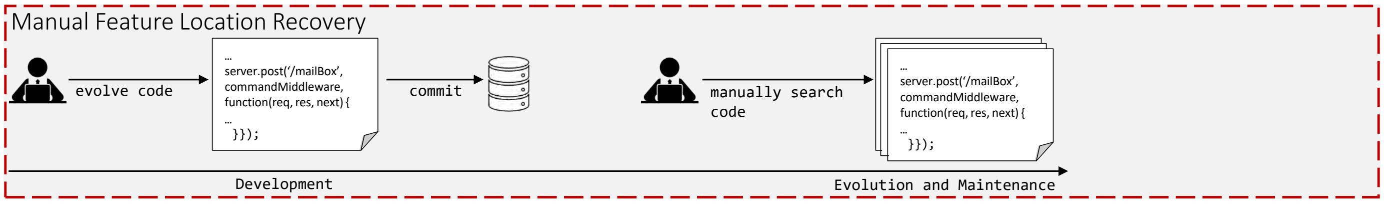
We contribute: (i) the technique FeatRacer with a prototype implementation; (ii) empirical data on FeatRacer's design options, including which learning algorithm and classification metrics best support feature location; (iii) multi-label datasets usable to further tailor FeatRacer or for follow-up research; and (iv) an online appendix [22] with a replication package and more details.

In previous work we also proposed a very early prototype of a recommender system [23], whose results inspired the present work. It used four text similarity and structure metrics, with binary classification, to suggest a single label (software feature) for non-annotated assets in a commit changeset. We evaluated this prototype on a single project and found an average F_1-score of 50 % with the k-Nearest Neighbors (kNN) algorithm as the best performing classifier.

On a final note, we built FeatRacer upon our experience with feature modeling [24]–[28], feature location recovery [23], [29], recording features with embedded annotations [30], [31], as well as with developing IDE-based tool support for recording, browsing, and evolving feature annotations [32], and for visualizing them [33]–[35]. We have evidence [30] that the benefits of recording feature locations using annotations embedded in source code outweigh the costs, since the annotations naturally co-evolve with the code, compared to external traceability databases. However, a challenge remained: reminding developers when they forgot to record, which inspired the nudging technique FeatRacer we present in this article.

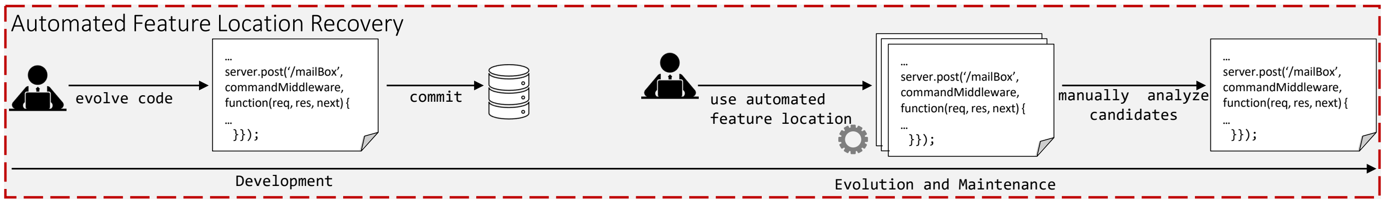
2 Background and Motivation

We motivate FeatRacer by describing the current state of the art in supporting developers who need to locate features. We explain the problems of manually recovering feature locations and describe how automated techniques and feature recording methods have addressed these. Neither approach has found widespread adoption yet, given the open challenges we describe in this section.



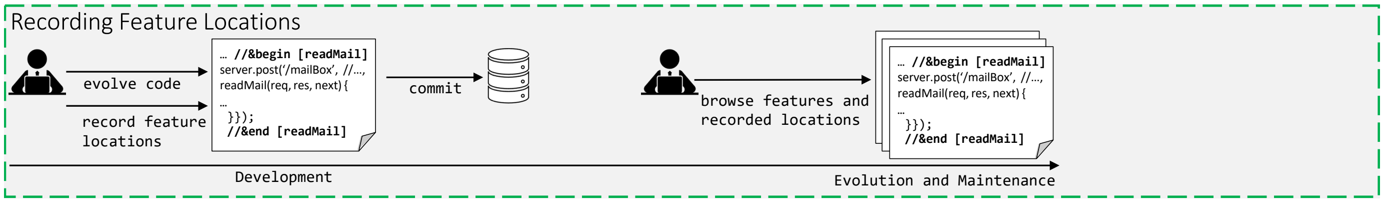
```
…
server.post('/mailBox',
commandMiddleware,
function(req, res, next) {
…
}});
```

(a) Manual feature location recovery is laborious and error-prone, but requires no extra effort during development.


Automated Feature Location Recovery
evolve code
commit
use automated feature location
manually analyze candidates
Development
Evolution and Maintenance

```
…
server.post('/mailBox',
commandMiddleware,
function(req, res, next) {
…
}});
```

(b) Automated feature location recovery also require no effort during development, but the techniques are still laborious to use and imprecise, still requiring significant effort.


Recording Feature Locations
evolve code
record feature locations
commit
browse features and recorded locations
Development
Evolution and Maintenance

```
… //&begin [readMail]
server.post('/mailBox', //…,
readMail(req, res, next) {
…
}});
//&end [readMail]
```

(c) Recording feature locations causes some extra effort during development, but eliminates all feature location effort during maintenance and evolution. Developers simply browse the features and locations. Continuously recording is the preferred strategy, but it comes with its own problems (e.g., costs, and developers not prioritizing it), which FeatRacer addresses.

Figure 1: Illustration of feature location during software maintenance using manual feature location, using automated feature location, and using recorded locations. Scenario (a) and (b) are problematic (red, dashed border), while scenario (c) is our proposed strategy, which FeatRacer enhances by assisting developers with an automated nudging technique.

Let us say a developer writes code, being fully aware that she is implementing the feature *send mail*. In many cases, the developer writes the code, but does not record the feature's location and instead retains that knowledge in her mind. More than half a year later, she receives a bug report that *send mail* is not working correctly. Unfortunately, since the memory about features fades with time [36], our developer cannot immediately remember which files or code fragments implement the feature *send mail*.

2.1 Manual Feature Location Recovery

To recover the locations, she could manually [14], [18] explore the codebase or perform some text searches as shown in Fig. 1a. This is a laborious process. She could also execute the application, select some menu entry that triggers the feature, and then record the stack trace with a debugger. Then, she would further explore the source code in the trace and also follow dependencies. Finally, if unsure about the result, she would verify the found locations, perhaps by writing test cases that exercise the feature. Two main problems complicate the manual recovering of feature locations.

First, **features often cross-cut multiple subsystems, files, or code fragments** [37]. Coupled with the developers' fading memory, the cross-cutting nature of features makes it even more laborious and error-prone to retrieve all locations [14].

Second, **features are highly domain-specific entities** [2] in contrast to traditional implementation assets, such as classes, components or files. Features abstract over these. Developers understand features differently depending on the project and domain. Another developer can easily have a different understanding of which functionality is included in a feature. Developer's understanding of features can also change. In fact, our own experience with developers recording features revealed that it is often difficult to decide what code exactly belongs to a feature [30]–[34]. Still, only the actual developers can make this decision, based on their detailed project knowledge and what feature they are currently implementing.

Empirical studies confirm these problems. An experiment comparing manual and automated (explained shortly) feature location by Perez et al. [20] on a 15-year old software project found that even domain experts who had on average 6 years experience on the project could not manually retrieve all requested features, because they did not have full knowledge of the features.

Summary. In the traditional scenario, no extra effort is imposed during the development phase, and feature information is just forgotten after implementing the feature. But, the feature location effort for developers is huge later on, so the effort is concentrated on the maintenance and evolution phases.

2.2 Automated Feature Location Recovery

Trying to avoid these shortcomings of manual feature location, the developer could use an automated feature location technique. Such techniques classify into static and dynamic techniques [17]. They return ranked lists of the files and methods that are considered the implementations of a feature. As shown in Fig. 1b, the developer can, after formulating a query or a test case, evaluate the candidates provided by the technique.

Static techniques typically act like a search engine, where developers specify search terms or seeds indicating what to look for in the source code. They leverage natural-language information embedded in the code (e.g., identifier names or comments) and semantic similarity between code elements. The analysis typically relies on information retrieval (IR) techniques, such as LSI and tf-idf [38]–[46], or on machine learning [47]–[50]. Some are enhanced with program dependence analysis, which use static dependencies between program elements, or changeset analysis, which exploits historical information about the codebase.

Dynamic techniques are more like execution monitors. They collect feature information based on execution stack traces and are helpful when a seed in terms of a program entry point (e.g., a menu item in the user interface) is known. Typically, a developer writes a test case that exercises a feature. The test case's execution stack trace is then analyzed by the dynamic technique. Some dynamic techniques are enhanced with static techniques [51], for instance by analyzing static dependencies between program elements.

Challenge 1: Laborious Use. Unfortunately, automated feature location techniques are resource-intensive, laborious to use, and difficult to set up—often requiring substantial adaptation to the project at hand. Static techniques often analyze the whole project's codebase. Dynamic techniques often require analyzing each feature individually, requiring to execute the system many times. The identification of seeds and the potentially necessary creation of feature-specific test cases is also laborious for developers. Despite being automated, both kinds of techniques still require extensive developer interaction for guidance on what and where to seek [17], to improve precision and recall.

Even if the query can be automatically generated from feature documentations, such as issue trackers, where the effort would be low for a developer, they still require effort to connect the feature's location in source code to an issue tracker. More importantly, those techniques are limited to features actually recorded in issue tracking, which is a strong limitation and an unrealistic assumption (for our subject systems, even when an issue tracker exists, it is far from having documented the features).

Challenge 2: Low Performance. Generally, the precision and recall of automated feature location techniques is low and strongly depends on how accurately the developer-specified input—search terms, seeds, or test cases—represents the desired feature [17], [20], [51]. For static techniques, coming up with the correct terms can be difficult for developers. The experiment by Perez et al. [20] showed that there was often a mismatch between the terms used for the LSI search query by the developers, and terms used in the source code. Furthermore, some behavior of programs is often undecidable without execution, further contributing to the low performance of static techniques. For dynamic techniques, the challenge lies in providing a correct seed and writing a good test case for the feature. Dynamic techniques are also limited to functional features, which exhibit visible behavior, and cannot locate quality-related (a.k.a., non-functional) features. Since they rely on the context of the execution traces, dynamic approaches under-approximate feature information, so they produce many false negatives (which requires developer interaction to improve precision).

Challenge 3: Domain-Specificity. Also recall that features are highly domain-specific entities (Sec. 2.1). The problem is that automated techniques are often trained [47] on a few large projects and then applied to other projects. As such, they have no chance to adapt to the specific notion of a project or a developer. Instead, a feature location technique should adapt to what developers consider relevant to individual features, in the project at hand.

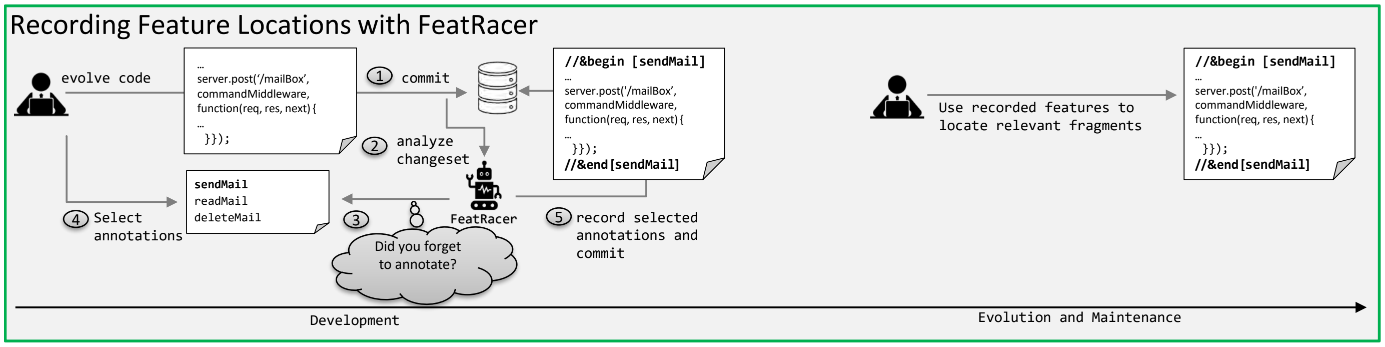
Challenge 4: Coarse Locations. Automated feature location techniques only recovers coarse locations (methods or files). However, in practice, features are more fine-grained [52]. So, even when the feature is correctly located, it can still require manual search effort to identify the exact location.

Summary. Automated feature location techniques promise no extra effort during programming. They focus on the maintenance and evolution phase, when features need to be located. There, they strive to reduce the feature location effort for the developer. Unfortunately, they have not shown to be effective in practice. They are difficult to adopt and produce too many false positives and only recover coarse-grained locations, leading to a huge effort during maintenance and evolution, when features need to be recovered

2.3 Recording Feature Locations

An alternative to using an automated technique, but to avoid the effort of manually recovering feature locations (Sec. 2.1) is to record the features during development (see Fig. 1c). In our example, the developer could document the implemented feature *send mail* in the documentation, in a traceability matrix (e.g., a spreadsheet), or a feature traceability database, such as FEAT [53]. Specifically, the developer records the feature, which is a label with a name, and the locations directly during or right after implementation. When needed during maintenance and evolution, she can immediately retrieve the locations for a feature with a simple search, saving substantial location costs. Notably, evolution also includes migrating software to software product lines, where features are made configurable, so customized software variants can be derived by switching individual features on or off.

While continuous recording provides long-term benefits—when (potentially highly scattered) features need to be located exactly—providing short-term benefits to engineers has been identified as one of the success criteria in establishing traceability [54]. Specifically recording features enables providing simple IDE support, such as feature dashboard plugins, which provide an overview understanding of software in terms of features, and they conveniently support browsing features in code [32], [55], as well as visualizing them [33]–[35]. Connecting features from the development process—most processes use notions of features, especially agile processes (e.g., SCRUM, XP, FDD)—down to the code can provide much more detailed progress tracking for features,


Recording Feature Locations with FeatRacer
evolve code
1 commit
2 analyze changeset
3
Did you forget to annotate?
4 Select annotations
5 record selected annotations and commit
FeatRacer
Use recorded features to locate relevant fragments
Development
Evolution and Maintenance

```
...
server.post('/mailBox',
commandMiddleware,
function(req, res, next) {
...
 }});
```

```
sendMail
readMail
deleteMail
```

```
//&begin [sendMail]
...
server.post('/mailBox',
commandMiddleware,
function(req, res, next) {
...
 }});
//&end[sendMail]
```

```
//&begin [sendMail]
...
server.post('/mailBox',
commandMiddleware,
function(req, res, next) {
...
 }});
//&end[sendMail]
```

Figure 2: Recording feature locations with recommendation. FeatRacer ensures that developers *proactively* and *continuously* annotate code by suggesting feature annotations when the developer forgets during commits. It is easy for the developer to vet the suggestions and select the correct ones. Manual effort during maintenance is significantly reduced.

allowing feature-oriented metrics (e.g., feature size, scattering degree, or tangling degree [52], [56]) that help managing and planning software development [57]. Furthermore, the recorded features and locations also enable feature-oriented software analysis and testing techniques [12], [58]–[61].

In this respect, recording features and their locations for software enables substantial benefits. Unfortunately, recording features during development is still challenging.

Challenge 5: Recording Costs. First, recording of features can be laborious, since it requires additional attention investment [62]–[64] and reasoning about two levels of abstraction. Specifically, developers are usually limited to one abstraction level (code) and have problems actively taking another abstraction level into account (features and the domain). Even though agile development techniques use the notion of features for planning, there is no agreed way of connecting the features to the code. Furthermore, note that recording costs comprise writing down the feature name and recording the location of the feature in code in some form. Costs also arise when the recordings become inconsistent and need to be updated.

Challenge 6: Nudging/Encouragement. Developers are likely to forget to record features and locations. Similar to developers not always commenting (perhaps even rarely) commenting code, recording features might not be prioritized during development. Consequently, some features may not be traced and the developer would have to follow the same procedure as in the first scenario to recover the locations of the none-traced features. So-called nudging techniques [65], [66] are known to change developer behavior. In our case, we believe that intelligent techniques should help developers to remember recording features, providing light-weight recommendations that remind developers when they forgot to record the feature locations for a specific feature.

Challenge 7: Code Evolves. The natural evolution of code causes two problems. First, it can prevent manual recovery of locations of features that were implemented by the same developer, not too long ago. Second, it causes the recorded locations to become inconsistent. For instance, complex refactorings can easily move the locations of features. This causes additional maintenance costs for the recordings. This problem especially arises when the locations are recorded externally to the code, such as in a traceability database. The inconsistency can go unnoticed until the feature needs to be located—then causing the same feature location recovery costs as for manual recovery.

Summary. Continuously recording features adds some effort to the development, while saving huge costs for feature location during maintenance and evolution. It also provides short-term benefits, such as browsing and visualizing features, tracking progress, or planning the software. However, effectively recording features is still an unsolved problem, requiring to encourage developers to *continuously* record feature locations. So, they need to be encouraged to change their behavior and practices—a.k.a., nudging developers in software engineering [65], [66].

3 The FeatRacer Technique

FeatRacer is a **Feat**ure t**Race** **r**ecommender that helps developers proactively and continuously record feature locations. We now describe how developers can use FeatRacer and how it addresses the challenges of locating features by combining automated techniques (Sec. 2.2) with feature recording techniques (Sec. 2.3).

FeatRacer reuses a lightweight feature recording system (embedded feature annotations) we designed and evaluated before [30]–[34]. The core novelty of our work lies in the recommender technique we now present and its combination with the recording system.

3.1 Usage Scenario

The practical use of FeatRacer is based on two core ideas. First, organizations adopt the recording of features with embedded feature annotations [30], [67] as a coding standard. Embedded annotations are lightweight and have been shown to substantially reduce feature location recovery costs, exceeding the costs of adding them significantly [30]. Even though our technique could work with different traceability techniques, we use embedded annotations due to their advantages over external databases [55], [67]. Second, organizations incorporate FeatRacer as a feedback loop into the development process. It learns from previous annotations and reminds developers when it predicts they forgot to annotate new source code with a specific feature.

We designed FeatRacer so that it should be triggered after a substantial software evolution step, such as a commit.

Figure 2 illustrates its use by a developer. In the remainder, we use commits as the unit of evolution, and also evaluate FeatRacer by invoking it per commit on the history of our subject systems. This suggests to invoke FeatRacer as a Git hook, but it could also be provided as an IDE plugin.

When FeatRacer is triggered (step 1 in Fig. 2), FeatRacer analyzes the changeset (step 2), particularly the assets that do not have any feature annotation. It makes its predictions based on the history of the project for which FeatRacer learned which assets are annotated with which feature. When it predicts that some assets are missing feature annotations, it recommends a ranked list of features that is easy to vet by the developer (step 3). When the developer accepts a recommendation (step 4), FeatRacer annotates the specific asset with the recommended features (step 5) and commits the changes to version control (step 6).

Ideally, FeatRacer—or at least embedded annotations—is adopted from the project start. However, FeatRacer can be adopted at any point during the development lifecycle. The important aspect is that after a while (details in our evaluation below), FeatRacer will make accurate recommendations for the features that have been declared so far. It is also possible to retroactively annotate code with features by an expert, which will positively influence the adoption of FeatRacer for the upcoming evolution of a software project.

3.2 Addressed Challenges

FeatRacer addresses the challenges of automated feature location and of manual feature recording techniques as follows.

Challenge 1. Recall that **automated feature location techniques are laborious** to use. FeatRacer is technology- and programming-language-independent and does not require a specific setup for individual technologies. It only assumes that projects adopt a way of recording features (which are simply labels) and their locations within software assets (embedded annotations). FeatRacer acts like a nudging technique that supports developers to actually record feature locations, helping them when they forgot to record. FeatRacer does not require executing the system, as dynamic feature location techniques do. Also, it does not require writing search queries (the exact feature names are recorded in the textual feature model) or writing test cases to find the feature.

Challenge 2. Recall that the **performance of automated feature location techniques is low**, generally producing too many false positives to be useful in practice. FeatRacer fosters a detailed recording with embedded annotations directly during the development, when the feature is still fresh in the developer's mind. Thus, the recorded locations are the most accurate ones that can be achieved. Conceptually, our strategy fosters that all features a developer considers relevant during development are recorded, and the completeness is enhanced through our recommender system, which nudges developers.

Also recall that improving the performance of automated techniques requires substantial user interaction. FeatRacer lessens developer interaction and effort by presenting an easy-to-vet list of recommended features for the non-annotated assets to the developer. It also automatically annotates the assets with the recommended features after confirmation by the developer.

Challenge 3. Recall that **features are highly domain-specific entities**, whose characteristics are also highly influenced by the project at hand. That means that any automated technique should adapt to the project at hand. FeatRacer learns from the locations recorded manually over the current project's history, thereby also taking the developer-specific feature characteristics into account. So, what constitutes a feature is defined by the actual developer. As such, FeatRacer can *learn the specific notion of feature by the project, domain or developer*. This allows FeatRacer to propose more accurate locations—in case the developers forgot to record them.

Challenge 4. Recall that **automated techniques only recover coarse feature locations**. Since FeatRacer relies on fine-grained feature annotations, it circumvents this problem and provides the developer with the locations in the actual granularity the developer used during development. It offers recording both *coarse-grained* and *fine-grained* feature locations, from folders via files to individual lines of code, according to the actual granularity of a specific feature. Furthermore, since FeatRacer operates on changesets and classifies only the fragments of code that have changed, the machine-learning-based recommendation to the developer is limited to the code that was just changed. So, the code upon which a recommendation is done should still be fresh in the mind of a developer, further enhancing precision of the recorded annotations.

Challenge 5. Recall that **recording feature locations causes costs**, both for the actual recording and the maintenance when code evolves (see below). Note that in agile software development, the features themselves are usually already recorded, since they are used for planning. As such, a developer knows the name of the feature (e.g., *send mail*) she is implementing, and only needs to record the locations once she implements the feature. Recall that this kind of cost is called attention investment, which FeatRacer mitigates as follows:

First, FeatRacer uses embedded annotations, which have been shown to be among the most accepted and usable ways of recording feature locations [55], [67]. The annotation system we reuse has been designed by us to be as lightweight as possible [30], [31]. We explain it in detail in Sec. 3.3 below.

Second, FeatRacer provides recommendations on individual changesets, limiting the developer's attention to a small portion of the codebase, which she has just worked on.

Challenge 6. Recall that it is **difficult to encourage developers** to continuously record features and their locations, since recording locations might not be prioritized during development. FeatRacer supports developers in two ways.

First, developers obtain short-term benefits. They can browse features in the source code using an IDE plugin [32] and visualize features [33]–[35]. Our lightweight annotation system is not only about recording feature locations, but it also comprises a simple textual feature model. In this file, developers organize the feature labels in a simple hierarchy (based on the Clafer syntax [68]). In fact, the feature hierarchy is one of the major benefits of a feature model [10], as it helps developers to organize features and keep an overview understanding of the software.

Second, the recommender acts like a typical nudging technique. To limit the nudging to the necessary minimum, it will only nudge the developer when it believes she forgot to annotate newly written code.

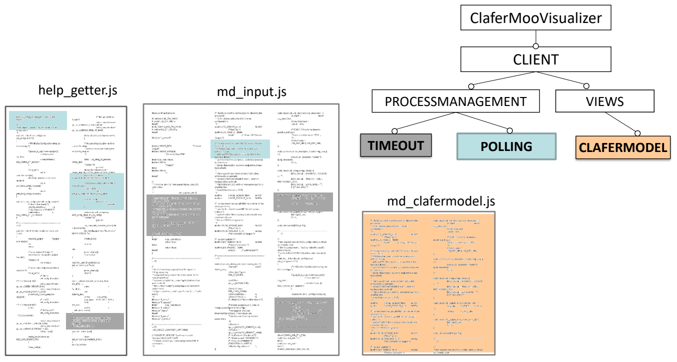


Figure 3: Illustration of feature models (top right) and feature locations in three files

Challenge 7. Recall that **code evolves**. We rely on an annotation system we previously designed to be as robust as possible against change [30], [31]. Evaluations have shown that embedded annotations *naturally evolve with the software assets* when they are moved and copied around. Only few feature annotations require manual maintenance. The cost incurred by this manual work is still amortized by the benefits of annotations [30]. In addition, FeatRacer by design also aims at catching situations where refactored code, which is naturally part of the changeset, is missing annotations. Then, FeatRacer will remind developers to add respective annotations, further mitigating the problem of code evolution and maintenance of feature annotations.

3.3 Embedded Feature Annotations

Features and feature locations are generally recorded for various reasons, mainly to handle complexity in large systems. Especially variability-rich systems, such as software product lines, software ecosystems, or personalization-capable systems, require explicit representations of features and their locations—the latter as variation points in the software. As input to automated configurator tools and variability-aware build systems, they allow to derive concrete variants (a.k.a., products of a product line) by selecting the desired features. To represent features and their dependencies, and as input to configurator tools, feature models [1], [25] have become the most popular notation, given their intuitive and simple syntax. Feature models are tree-like structures of features and their dependencies [1], [24], which abstractly represent complex systems.

Figure 3 (upper right-hand side) illustrates the feature model notation. Many feature modeling tools [12], [69], [70] show feature models in this notation. Alternatives are tree-menu representations, such as in the Linux kernel configurator [71]–[73]. In the figure, the highlighted fragments in the files illustrate the feature locations. In the following, we use the term asset to refer to any structural implementation artifact that can be mapped to a system's features. We distinguish between three granularity levels—folders, files, and fragments, consisting of either single or multiple lines of code.

To avoid having to adopt a dedicated feature-modeling tool, we reuse a lightweight annotation system we designed [30] and formalized [31] before, in multiple iterations. A

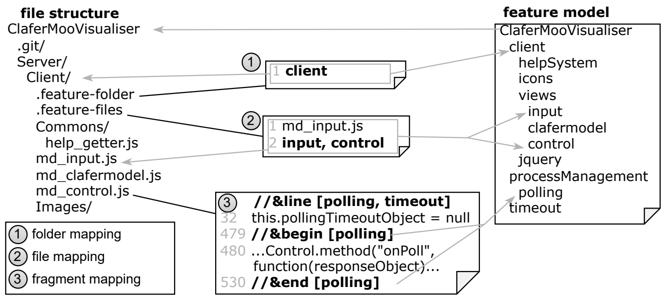


Figure 4: The example from Fig. 3 (extended with more files and features) represented using our lightweight embedded annotations

detailed specification is available.[1] Instead of requiring a graphical editor, it relies on a very simple representation in a text file, which can be put into the project root. The feature locations are stored in simple textual mapping files that map whole folders or files to features. In-file annotations, which are escaped with comments in the source files, map fragments or single lines. This way, the annotation system is programming-language- and technology-independent. As such, the representation of features and their locations is inspired by very established representations, including feature models, feature databases [53], or other annotation systems [67], while being very lightweight.

To illustrate the annotation system, Fig. 4 shows the same example. Given the project file structure (part 1) and a list of features of the project (part 2), the developer can annotate, for instance, the folder *Client* by adding a plain-text file called *.feature-folder* inside the folder, and in there listing all the features the folder implements; in this case the feature called *client* (part 3). To annotate the files inside folder *Client*, the developer adds a text file *.feature-files*, inside the folder, and for each line of file names, e.g., *md_input.js*, the developer adds a second line indicating the names of the features implemented by the files, e.g., feature *input* (part 4). Lastly, the developer can annotate code fragments or even individual lines of code using specially formatted comments shown in part 5 of Fig. 4. While the feature model file could accommodate feature dependencies, these are not important for FeatRacer, since our main purpose is to trace features to source code, not to realize variation points in a configurable product-line platform. Still, the ability to migrate a system with recorded features and locations into such a platform, by replacing annotations with variation points (e.g., `#ifdef`), is one of the benefits of FeatRacer.

4 Design of FeatRacer

The design of FeatRacer facilitates effective and efficient recommendations as follows. We now describe how it calculates metrics to generate the training data to learn characteristics of features and their associated assets.

4.1 Internal Workflow

FeatRacer's internal workflow is illustrated in Fig. 5. In brief, to make recommendations, it needs to create training data

1. https://bitbucket.org/easelab/faxe/raw/master/specification/embedded_annotation_specification.pdf

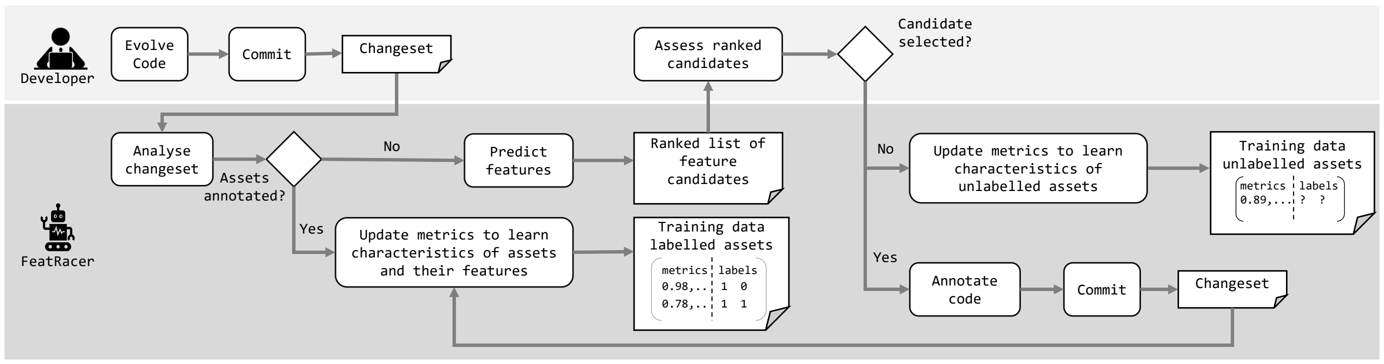


Figure 5: Illustration of FeatRacer's main use case (recommendation scenario), with steps from Fig. 2

(metric values over software assets), train a machine-learning classifier, make predictions, update the training data when the software evolves, and retrain the classifier.

Importantly, FeatRacer separately and simultaneously trains and predicts feature locations for three granularity levels: folders, files, and code fragments (hunks in the diffs of the commit), which improves prediction performance, since characteristics of assets at these granularity levels may differ.

When FeatRacer is first run on a project, it creates training data for the project codebase. It analyzes the current revision by parsing existing annotations and filling an internal representation that represents the revision. It then calculates metrics over the features and the assets implementing them (i.e., the feature locations). Intuitively, the metrics indicate which assets are associated with a feature based on the assumption that assets implementing the same feature share similar characteristics, and that one can learn per project and feature what those similar characteristics are. For instance, for each asset, FeatRacer measures how many developers contribute to that asset (DDEV), how often the asset has been changed (COMM), how many other assets it is usually changed with (ACCC), the size of the asset in lines of code (NLOC), and the number of features usually modified together with the asset (DNFMA).

Then, FeatRacer uses a machine-learning algorithm to learn the characteristics of the features and the assets that were annotated by developers, in order to build a machine-learning model. If there are non-annotated assets in the changeset, it then uses this model to recommend feature annotations to the developer for each one of them, as we explained above (Sec. 3.1). Otherwise, the recommendation step is skipped and it will only use the assets in the changeset for learning the characteristics of features.

Whenever FeatRacer is invoked, it updates the training data incrementally when the software evolves, by loading features and feature locations for assets modified in a changeset, and by recalculating metrics for assets that are affected in each revision (commit) of a project. It then retrains the machine-learning model using this updated training data. FeatRacer keeps the calculated information about assets in a cache. We observed in our experiments that the training of a machine-learning model is fast (at least for projects as large as our evaluation subjects), so it can be done for each evolution step. This incremental update of training data avoids reading all contents of a repository and calculating metrics for all files in the repository at each revision, since, in some cases only a few files may be affected. For instance, in a project with 1.000 files, perhaps only two files may be affected in a revision, so, FeatRacer calculates metrics for the two files only and updates its training data accordingly, while metrics for the 998 unaffected files remain unchanged, since they were already calculated during previous commits which affected them. Note that, since the training data is incrementally built, FeatRacer trains on all assets of a project for which metrics have been calculated, not only those in the changeset.

When FeatRacer predicts that a non-annotated asset can be annotated, it awaits the user's input (cf. Sec. 3.1). When the developer accepts one or multiple recommendations, FeatRacer annotates the specific assets with the recommended features, updates the training data with those assets' metrics, and commits all accepted annotations to version control (step 6); otherwise, FeatRacer updates its training data with the metrics of the non-annotated assets. Therefore, the training data consists of assets with and without annotations.

As stated above, FeatRacer calculates metrics only for assets affected in each revision to incrementally update its training data. The initial commit from which metrics are calculated can be at any point in the revision history of the project, e.g., the first commit or even the 100th commit. For this initial commit, FeatRacer extracts metrics from all files of the project at the state of the commit. Then, for subsequent commits, it only extracts metrics for affected files.

Table 1: FeatRacer's asset metrics

metric	description	range
CSDEV	Average cosine similarity between the names of the distinct developers contributing to the asset; *csdev(a,C)*. Each name is compared to every other name and similarity values are averaged.	0.0-1.0
DDEV	Distinct number of developers contributing to the asset; *ddev(a,C)*	≥ 1
DCONT	Average percentage of lines contributed by each developer contributing to the asset; *dcont(a,C)*	0.0-1.0
HDCONT	Percentage of lines by the developer who contributed most to the asset; *hdcont(a,C)*	0.0-1.0
COMM	Number of commits in which the asset has been modified *comm(a,C)*	≥ 1
CCC	Collective change count—number of *all* assets modified alongside this asset in the current commit; *ccc(a,c)*.	≥ 1
ACCC	*Average* number of assets modified together with this asset across all commits in which *a* is modified; *accc(a,C)*.	≥ 1
DNFMA	Number of distinct features mapped to all assets modified together with the asset up to the current commit; *nfcca(a,C)*	≥ 0
NFMA	Number of features modified in current commit with asset; *nfma(a,c)*	≥ 0
NLOC	Number of lines of code associated with the asset in the current commit; *nloc(a)*	≥ 1
NFF	Number of features within file containing asset *a* at current commit; *nff(a,c)*	≥ 0

Let *a* be an asset modified in the current commit *c*, and let *C* be the set of all commits in which asset *a* has been modified, from the initial to the current.

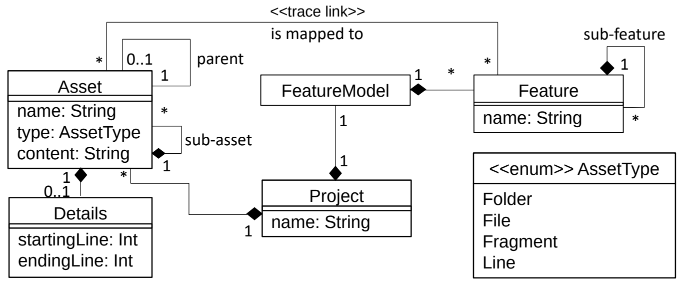


Figure 6: The feature information FeatRacer expects to be recorded in a project

4.2 Internal Representation of Assets and Features

While we advocate using our annotation system from Sec. 3.3, FeatRacer is designed to work with any other representation, where feature information is also embedded in assets to obtain the benefits we described above, such as ease of use. The minimum information developers need to capture during development is represented in the meta-model in Fig. 6.

FeatRacer expects this information in a project to make recommendations. Specifically, each feature can be implemented by (i.e., traced to) multiple software assets, and each asset can implement (i.e., trace to) multiple features. An asset is any structural implementation artifact and can be of type folder, file, or fragment. A fragment is any block of consecutive lines of code. For instance, we consider the block of code annotated with feature *sendMail* in Fig. 2 as a fragment. Any non-annotated lines of code before this *sendMail* annotation would be considered as one fragment, and those after the annotation would be another fragment.

The minimum information developers need to capture during development is represented in the meta-model in Fig. 6. This meta-model as well as the concrete syntax in terms of the embedded feature annotations (cf. Sec. 3.3) we use are programming-language-independent. Recall that our representation of features and their locations is inspired by very established representations—feature models [1], [25], feature databases [53], and other annotation systems [67].

4.3 Learning Features

To learn a project's specific notion of features, FeatRacer generates training data for its machine-learning classifiers by calculating metrics that represent characteristics of the project's assets (folders, files, and fragments).

Asset Metrics. FeatRacer learns characteristics of the features of a project by measuring the relatedness of assets implementing each feature. We assume that assets implementing the same feature(s) are likely to have similar characteristics, and that it can be learned what those similarities are per feature. We first experimented with structural characteristics, such as the number of lines associated with the asset in the commit and cosine similarity between textual content of assets. However, we found that using characteristics revolving around the process is more effective in learning features than structural metrics. This is similar to studies in software defect prediction that show that process metrics are better than structural metrics in predicting which files are more prone to defects [74]. Therefore, FeatRacer considers similar process metrics, such as the developers of the assets, how frequently the assets change, which assets are often changed together, as well some structure metrics, such as the number of lines of code associated with each asset and feature. Then, when it finds assets in a new commit that are missing feature annotations, it predicts annotations for these assets.

FeatRacer aims to recommend features for assets that developers forgot to annotate, by learning from previously recorded annotations. To this end, we engineered metrics (Table 1) that measure characteristics of features. These are utilized to estimate the relatedness of assets that implement a given feature. This way, when FeatRacer finds a non-annotated asset during a revision, its machine-learning classifiers can predict, based on the asset's metrics, which of the existing features in the project the asset implements.

Given A, as the set of all assets modified in the commit c, and C, as the set of all commits (from initial to current one) in which asset $a \in A$, has been modified, FeatRacer calculates metrics for asset a. These asset metrics are described in Table 1. Note that some metrics require historical information, e.g., the number of commits in which an asset has been modified (COMM), developers who ever contributed to the asset (DDEV, DCONT, HDCONT), or which other assets an asset is usually changed with (ACCC), while others only require information from the current commit, e.g., how many assets or features are affected together in the current commit (CCC, NFMA), or number of features in the asset's file, and size of the asset (NFF, NLOC). We conjecture that these metrics are good indicators of related assets, since assets of a given feature are likely to be often modified together, by about the same number of developers, who sometimes spell their names differently, and work on about the same number of features. Therefore, our metrics cover four aspects of development: *developers* (CSDEV, DDEV, DCONT, HDCONT), *frequency of changes* (COMM), *co-changes* (CCC, ACCC, DNFMA, NFMA), and *code and feature size* (NLOC, NFF).

The metrics DDEV, DCONT, HDCONT, COMM, CCC, ACCC, NLOC stem from defect prediction [75]–[77] (although named differently). We adapted them to FeatRacer. We also introduced four new metrics: CSDEV (similarity in the names of the developers who ever contributed to the asset), DNFMA (number of features that are usually modified when this asset is modified—averaged across all past commits), NFF (number of features referenced in the file containing this asset—applies to fragments) and NFMA (number of features modified together with this asset in the current commit c—we count all features mapped to assets in the changeset).

Generating Training Data. FeatRacer's training data consists of the metrics in Table 1 as the attributes of the dataset, the software features mapped to each asset as the classification labels of the dataset, and the assets (folders, files, and fragments) as the instances (or datapoints) of the dataset. The dataset is multilabel, because each asset can implement multiple features, and a feature can be implemented by multiple assets. FeatRacer updates its training data at each revision of the project after it calculates metrics for the assets affected in the revision.

Let D be a multilabel dataset (MLD) composed of N instances $E_i = (x_i, Y_i), i = 1..N$. Each instance E_i is associated with a metric vector $\mathbf{x_i} = (x_{i1}, x_{i2}, ..., x_{iM})$ described by M metrics $X_j, j = 1..M$, and a subset of labels

Table 2: Example dataset with four metrics and three labels

	COMM	DDEV	ACCC	NFF	server[1]	client[1]	bubbleGraph[1]
E_1	4	2	0.5678	5	1	0	1
E_2	2	1	0.2346	4	1	0	0
E_3	4	3	0.9678	8	0	1	0
...	...	...	...	...	...	...	...
E_n	8	4	0.452	5	?	?	?

[1] labels (i.e., the software features mapped to assets)

$Y_i \subseteq L$, where $L = \{y_1, y_2, ...y_q\}$ is the set of q labels (in our case, all features recorded in the project so far).

To illustrate the structure of FeatRacer's MLD, Table 2 shows the metric vector $\mathbf{x}$ = (COMM, DDEV, ACCC, NFF)—only four shown here, but in reality we use the full set of metrics in Table 1—and the set of labels $L = \{server, client, bubbleGraph\}$—only three shown here, but in reality we use the full set of features in a project at the state of the commit for which the dataset is generated. As stated above, each instance in the dataset is an asset (E_i), which could be a folder, file, or fragment of code. Instance E_1 has a metric vector $\mathbf{x_1} = (4, 2, 0.5678, 5)$ and a labelset $Y_1 = \{server, bubbleGraph\}$—i.e., the asset E_1 with the indicated metric values, is annotated with features *server* and *bubbleGraph*, while assets E_2 and E_3 are annotated with single features *server* and *client* respectively. The multi-label classification task is to generate a classifier H which, given an unseen instance $E = (x, ?)$ (e.g., E_n in Table 2), is capable of accurately predicting its subset of labels Y , i.e., $H(E) \to Y$. The classification task is multi-label, since each instance, seen or unseen, can be associated with several labels.

FeatRacer creates separate datasets for the three different granularities of software assets: folders, files, and fragments. That is, a file-level training dataset only has files as instances, and the set of labels L in the dataset only includes features mapped (i.e., through annotations) to files; similarly, a fragment-level dataset only includes fragments and features mapped to fragments only as the labels of the dataset. This is to reduce label-imbalance. Consider a scenario where a dataset would contain both file- and fragment-level assets. If the dataset has one file mapped to one feature (featureA), and 300 fragments of code mapped to one feature (featureB), the two labels are highly imbalanced by a ratio of 1:300. This would bias classifiers towards the dominant fragment-level feature (featureA) and make it difficult to correctly predict annotations for the file-level feature (featureB). Therefore, having separate datasets for files and fragments of code in this scenario alleviates this problem.

FeatRacer's training data consists of both feature annotated and non-annotated assets. Whenever new assets are added during a commit, the training data is updated and the classifier retrained. That is, metrics are only calculated for the newly added or modified assets, but retraining of the classifier is done on the entire dataset.

Machine Learning Algorithms. The machine learning algorithms used by FeatRacer are multi-label (can predict and rank multiple features per asset), are capable of working with small datasets, and can be trained to predict locations within a short time suitable to give feedback to the developer quickly after some reasonable software evolution step (e.g., committing code to version control). We describe the 13 algorithms we evaluated in more detail in Sec. 5.2.

4.4 Predicting Feature Locations

When FeatRacer finds non-annotated assets when the developer commits code, it creates a test dataset, similar in structure to the training dataset. The instances are the non-annotated assets. Again, three test datasets are created for the three different granularity levels of assets. Then, FeatRacer simultaneously predicts and recommends annotations for each asset in the test datasets. The recommendation is a ranked list of feature names with which the asset could be annotated. The machine learning algorithms that FeatRacer uses are capable of ranking predicted labels. The developer can then accept or reject the recommendation.

5 Evaluation Methodology

We now describe the evaluation of FeatRacer. We first describe the software projects we used, then our research questions, and then how we simulated the development of the subject projects. Lastly, we explain how we measured the prediction performance, also over time.

5.1 Subject Projects

To evaluate FeatRacer's effectiveness in suggesting relevant feature locations, we selected a diverse set of 16 projects that have feature locations recorded as annotations in the source code. These projects have been used in studies on feature-oriented defect prediction [75], feature location [29], [30], and feature analysis [52], [78]. They also exemplify different programming languages (JavaScript, C, C++) and have revision histories that span between 1 and 11 years (average 3).

All projects use some form of feature annotation. The four projects belonging to the ClaferWebTools [79] use a custom annotation system. Those annotations are, essentially, specially formatted source code comments [30] (see Fig. 4). The other projects use the C/C++ pre-processor, where conditional compilation directives wrap code belonging to a specific feature. Note that we excluded pre-processor annotations that appeared in comments. As shown in many studies [24], [52], such projects declare features and refer to them in the code, where features are defined as preprocessor macros and used in conditional compilation directives. Commercial projects represent features the same way [9], [80], [81]. Some of our projects, including Busybox and the four ClaferWebTools projects, even model features explicitly in a feature model. Marlin prescribes a feature-oriented development process where new features need to be declared as preprocessor macros, and code needs to be annotated[2] [29]. While our FeatRacer prototype parses both kinds of annotations, it can easily be adjusted to other annotation systems. Moreover, the varying granularities of these annotations makes our evaluation dataset suitable to evaluate FeatRacer's effectiveness in recommending relevant feature locations.

Table 3 summarizes our subject projects. We show the domain and size of each project, the main programming language for files containing feature annotations, the number of commits we used for evaluating FeatRacer from the

2. https://marlinfw.org/docs/development/coding_standards.html

Table 3: Summary of our subject projects

project	short name	description	lang.[1]	dev.[2]	size[3]	commits[4]	dev. period	years	features[5]	URL
ClaferConfig-urator	config	software variant management	JS	7	1.4K	182	2013–2014	2	43	github.com/redmagic4/Clafer-Configurator
ClaferIDE	ide	software variant management	JS	4	1K	269	2013–2014	2	31	github.com/redmagic4/ClaferIDE
ClaferMooVi-sualizer	viz	software variant management	JS	9	2.7K	542	2012–2014	3	72	github.com/redmagic4/ClaferMoo-Visualizer
CommonPlat-form	tools	software variant management	JS	2	472K	147	2013–2014	2	50	github.com/redmagic4/ClaferTools-UICommonPlatform
Blender	blender	3D modelling tool	C/C++	23	451K	664	2002–2003	2	211	github.com/sobotka/blender
Busybox	busybox	UNIX toolkit	C/C++	6	36K	260	1999–2000	2	607	git.busybox.net/busybox/
Emacs	emacs	text editor	C/C++	10	198K	546	1985–1992	7	464	github.com/emacs-mirror/emacs
Gimp	gimp	graphics editor	C/C++	42	349K	129	1997–1998	2	382	gitlab.gnome.org/GNOME/gimp
Gnumeric	gnumeric	spreadsheet	C/C++	44	73K	902	1998–1999	2	107	gitlab.gnome.org/GNOME/gnu-meric
Gnuplot	gnuplot	plotting tool	C/C++	2	76K	331	1987–1998	11	569	github.com/gnuplot/gnuplot
Irssi	irssi	IRC client	C/C++	1	48K	720	1999–2000	2	174	github.com/irssi/irssi
Libxml2	libxml2	XML parser	C/C++	10	320K	370	1998–2000	3	136	gitlab.gnome.org/GNOME/libxml2
Lighttpd1.4	lighttpd	web server	C/C++	5	49K	306	2005–2005	1	197	git.lighttpd.net/lighttpd/lighttpd1.4-.git/
Marlin	marlin	3D printer firmware	C/C++	68	756K	966	2011–2015	5	432	github.com/MarlinFirmware/Marlin
MPSolve	mpsolve	polynomial solver	C/C++	3	37K	232	2011–2011	1	71	github.com/robol/MPSolve
Parrot	parrot	virtual machine	C/C++	15	33K	1001	2001–2002	2	40	github.com/parrot/parrot

[1] main language of files with annotations [2] commit authors [3] LOC (code+comments) [4] first *n* commits analyzed. [5] features in annotations.

project's revision history, the number of developers of those commits, the time covered by the commits and number of features appearing in annotations. We observe a diversity in project characteristics: a development period ranging from 1985 to 2015, the number of features ranging from 31 to 607, and the number of developers ranging from 1 to 68.

5.2 Research Questions

We evaluated FeatRacer's prediction performance (RQ1), identified the metrics that help to best characterize features and related assets (RQ2), and investigated the influence of commit practices on FeatRacer's precision and recall, also over time (RQ3). In addition, we evaluated FeatRacer's performance against widely used feature-location techniques relying on LSI and LDA (RQ4).

Selecting Classifiers. We evaluated the performance for different machine-learning algorithms.

RQ1: What multi-label learning algorithm has the best performance when predicting features for software assets?

We used 13 multi-label classification algorithms from the MULAN [82] library: Binary Relevance (BR), Classifier Chains (CC), Calibrated Label Ranking (CLR), Ensemble of BRs (EBR), Ensemble of Classifier Chains (ECC), Ensemble of Pruned Sets (EPS), Multi-Label Stacking (MLS), Label Powerset (LP), Pruned Sets (PS), Disjoint Random Pruned Sets (RAkEld), AdaBoost.MH (AMH), Multi-Label k-Nearest Neighbors (MLkNN), and Instance-Based Logistic Regression (IBLR_ML). We used WEKA's J48 as the underlying binary classifier for all transformation-based classifiers, which transform a given dataset into multiple binary datasets and operate on them using traditional binary classification approaches—i.e., all the above classifiers except MLkNN and IBLR. Note that these classifiers do not require us to set any thresholds as cut-off values.

We performed an initial evaluation of our 13 classifiers using standard cross-validation (of ten groups, seven for training and three for testing) with data from all commits in the revision history of project ClaferMooVisualizer. We picked on project ClaferMooVisualizer because it has more feature annotations than the other three ClaferWebTools projects, and because our primary interest is to work with feature annotations and not just variability annotations. After cross-validating our classifiers on the ClaferMooVisualizer project, we dropped CLR and AMH, which had the worst performance, and MLS, EPS, and PS that generated errors during execution which we could not resolve.

Next, we decided to evaluate the remaining classifiers using a real scenario in which we would want to predict feature locations for a given separate test set. We, therefore, created training data and test data based on assets changed in the entire revision history of project ClaferMooVisualizer. For each commit, the training dataset comprised assets present in the project at that commit, while the test dataset comprised only assets that were affected by the next commit—i.e., the training data comprised assets in the current nth commit white the test set only comprises assets modified in the n+1th commit. We did this for all three granularity levels. This led us to drop IBLR and MLkNN since they did not work on datasets with very few instances. This left us with BR, CC, EBR, ECC, LP, and RAkEld. Then, we decided to use a project with preprocessor-based feature annotations—Marlin, which is one of our largest subject projects. After evaluating the remaining classifiers on datasets generated from the first 130 commits of Marlin, we dropped CC, EBR, and ECC, since they took too long to train for large datasets (sometimes taking up to 45 minutes or more), which is undesirable for FeatRacer's recommendation scenario. Hence, we only evaluated BR, LP, and RAkEld on all projects. Recall that our training datasets are generated incrementally per commit, simulating how FeatRacer would work in practice. That is, when predicting feature annotations for assets modified in commit #130 of Marlin, the training data consists of all assets modified from the first commit up to commit #129.

Evaluating Performance. Since we aim to remind developers to record feature annotations whenever they commit changes to their repositories, we evaluated FeatRacer for the worst-case scenario in which the developer would forget to annotate all assets in a commit. Rather than cross-validate our classifiers, we let them predict annotations for all assets that changed in every subsequent commit and measured their precision and recall. We did this for each granularity level—folder, file, and fragment. We applied this evaluation of the classifiers for all our research questions.

RQ2: Which of our proposed metrics are best predictors of *features for software assets?* We used a multi-label attribute selection technique to rank our eleven metrics (Table 1) based on their ability to predict feature locations. We applied *ReliefF with Binary relevance (ReliefF-BR)* and *ReliefF with LabelPowerset (ReliefF-LP)* [83] on all datasets for each granularity level and project to rank the metrics for that particular level and project. Then, we measured FeatRacer's performance when predicting feature locations using four sets of metrics based on the ranking—i.e., when trained with the top 4 (36%), 6 (54%), 8 (72%), and 11 (100%) of the metrics.

ReliefF is a *filter approach* [84] to attribute selection. Filters use general characteristics of a dataset to select some attributes and exclude others, regardless of the learning algorithm. As such, they may not choose the best attributes for some algorithms. However, they are widely used in research on multi-label learning [85] and have been shown to outperform other techniques, such as *information gain*.

RQ3: What influence do developers' commit practices have on the prediction performance?

With this question, we sought to understand how structural changes to source code introduced by a commit may impact FeatRacer's performance. First, we measured FeatRacer's performance over time for each project's revision history, for the most substantial granularity level (fragments). Next, we manually reviewed commits where FeatRacer performed poorly (see performance drops in Fig. 7) to obtain insights on what could be the cause. We then calculated commit metrics (based on structural changes to code), such as the number of added lines in a commit and hunk size, for each commit in a project. Then, using Spearman's correlation coefficient, we measured correlations between the commit metrics and FeatRacer's average performance for the commit. We also measured correlations between the asset metrics in Table 1 to FeatRacer's performance in each commit for additional insights. The commit metrics we extracted differ from the asset metrics in Table 1: while we use the latter to characterize assets of a given feature to facilitate feature-location recommendations for non-annotated assets, the former mainly measure structural changes, such as hunk size, files added, and code churn (added lines minus deleted lines), that developers introduce with each commit, which may impact prediction performance, but may not provide characteristics of assets of a given feature. So, we could not use the commit metrics in our prediction models, which rely on measuring the relatedness of assets of a given feature.

RQ4: How does FeatRacer's performance compare to that of traditional automated feature location techniques?

We compared FeatRacer's performance to that of the two information-retrieval techniques LSI [19] and LDA [86] commonly used in automated feature location [20], [21], [87].

Recall that, as we stated in Sec. 1, developers usually know the names of the features they wish to locate during code maintenance. Therefore, it is more intuitive and requires less effort for them to search their code base by the feature names than by writing complex descriptions of the features, which would overwhelm them both in the input required and the output to analyze. Hence, a practical feature location technique must adapt as closely as possible to the developer's workflow and development process—require minimal input from the developer to locate features, but be very precise to not overwhelm the developer with irrelevant results. To this end, we compare FeatRacer's performance to that of LSI and LDA when recovering feature locations in a way more intuitive to the developer—searching by feature name.

For LSI, we used Apache Lucene's implementation to retrieve a feature's assets at each commit. Our steps were: (Step 1) For each commit in a project, we checked out the whole project (which contains feature annotations as our ground truth) at the state of the commit. We generated a list of all feature names in the source code, together with the names of the assets annotated with the features. Thereafter, we stripped off all feature annotations from the source code. (Step 2) We indexed both files and code fragments using three fields (asset full path, asset textual content, asset's parent—used to map fragments to their parent files). (Step 3) For each feature name in the list (generated in Step 1), we cleaned it by removing any non-word characters, and split compound words. We then queried the index (from Step 2) for the cleaned feature name, searching the *full path* and *textual content* fields of the index. (Step 4) We then generated (i) detailed search results showing assets matched for each feature and the score assigned to each asset by Lucene, and (ii) summarized results showing precision, recall, and F_1-score for each feature, based on the ground truth mapping generated in Step 1 for assets whose score was 0.7 (70%) or more.

Please note that we stripped feature annotations from the source code because we assume that the developer has not annotated the source code being queried, hence the need for automatically locating the features. Otherwise, if the source code already has feature annotations, the developer only needs to use some visualization tool or regular expression query to retrieve corresponding assets and does not need a special feature location tool.

For LDA, we created separate training and test datasets, similar to how we created them for FeatRacer, i.e., using all assets and features present in the repository up to the nth commit as part of the training set and those changed in the n+1th commit for testing. In this case, the assets are code fragments. Since LDA expects preprocessed textual data, we preprocessed our fragments by splitting words on camel case and underscores; removing non-letters and converting all words to lower case; and removing English stop words and programming language related keywords.

To create the training datasets, we used the textual data of the assets present in the repository until the current nth commit. After preprocessing, we then fed the data into LDA. Here, we used the python library Gensim [88] and a publicly available repository to query LDA using Apache Lucene [89]. Then, LDA tries to cluster the terms it reads in the training set to divide them into *n* topics, where n is the number of features present up until this point. As LDA is an unsupervised method, we do not need the labels for training. Instead, we use them to count the number of features currently present. Based on that it trains the model, which is a set of topics containing the weighted frequencies of terms it has read. We set α to 0.01 and β to a symmetric distribution and the number of passes to 20. After experimenting with different cutoff values, we decided to not use any threshold as the F_1-score was higher when not using any.

Thereafter, we used the resulting model to make predictions on every fragment in the test set. For each fragment, the model creates a likelihood vector containing

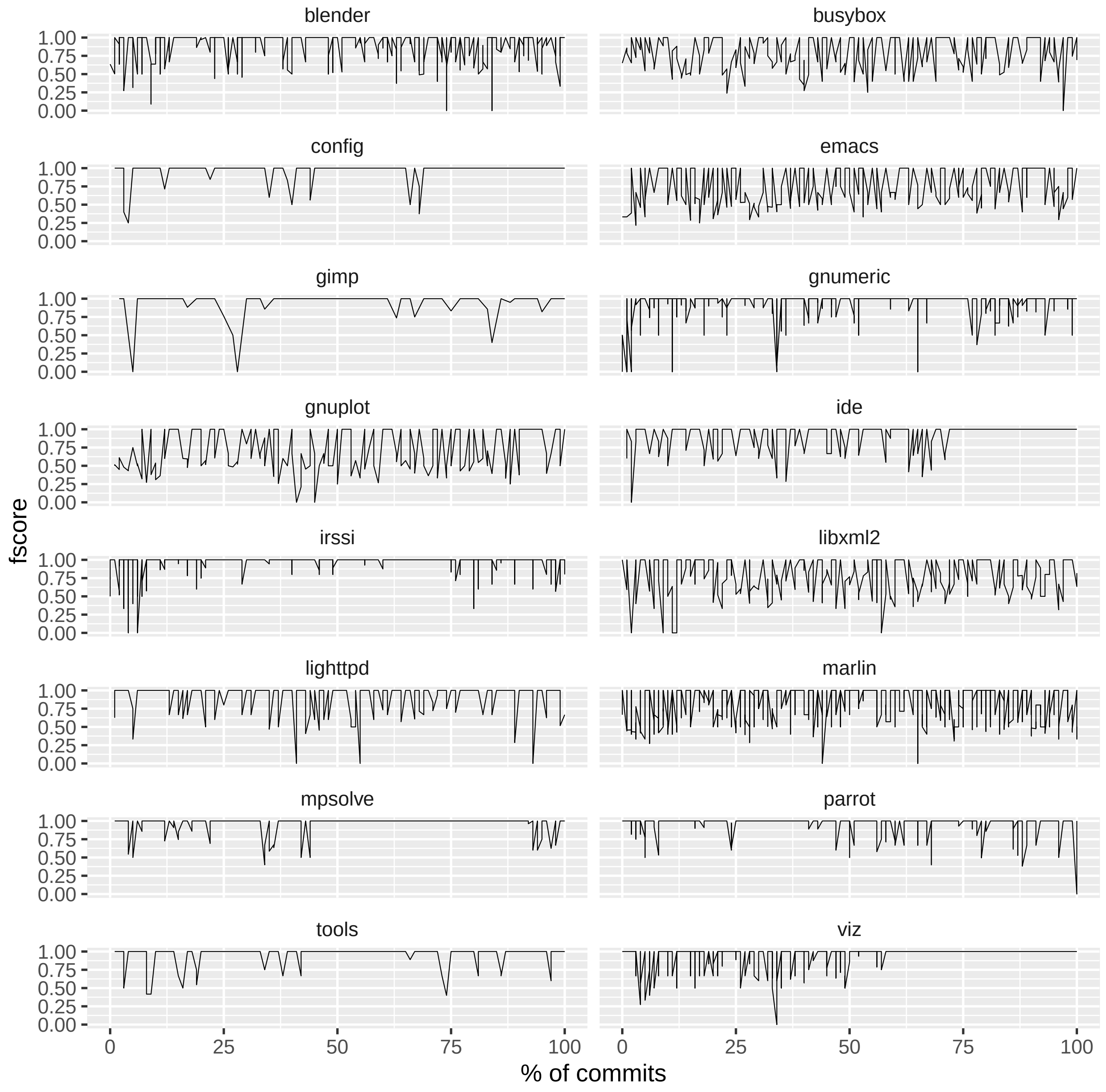


Figure 7: Fragment-level predictions using RAkELd classifier. The evaluation period is normalized to the percentage of commits analyzed in a project

the likelihoods for each feature. We fed this information into Apache Lucene, which creates indexes that we can use to query for the feature names (similar to how we did for LSI).

5.3 Simulating Software Development with FeatRacer

To answer our research questions, we simulated the development of our subject projects by letting FeatRacer predict features for assets along the projects' revision histories. We evaluated FeatRacer on the worst-case where a developer would have annotated assets normally until the commit under evaluation, but in that commit would have forgotten to annotate all relevant assets (that would have been annotated, as per the original annotations) in a commit. We then compared the predicted features of assets with the ground truth. We did that for every commit, and assumed that for the next one, the developer would have fixed all annotations based on her expert knowledge.

More specifically, for each subject project, we performed two steps: (Step 1) We gave FeatRacer a range of consecutive commits over a substantially long (see Table 3) time span from the project's revision history. We let it generate training data for each revision as well as predict feature annotations for the following commit's changeset. More precisely, given

Table 4: Average asset (fragment) training dataset composition per project vs prediction performance using RAkELd classifier

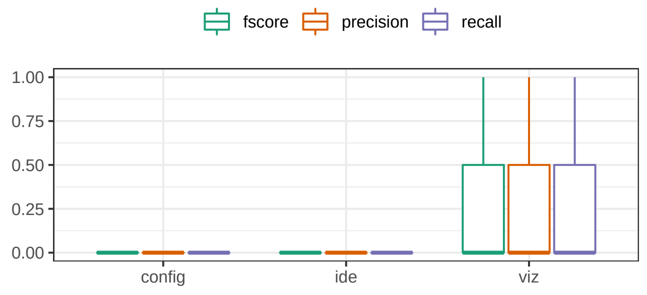
	annotated	non-annotated	% annotated	% non-annotated	fscore
config	54	350	13	87	0.95
ide	209	150	60	40	0.91
viz	241	481	30	70	0.95
tools	96	1,445	7	93	0.94
blender	4,964	1,008	81	9	0.90
busybox	2,169	515	81	9	0.79
emacs	1,402	96	92	8	0.76
gimp	2,382	926	72	28	0.93
gnumeric	263	1,567	18	82	0.96
gnuplot	3,964	153	96	4	0.71
irssi	365	788	38	62	0.97
libxml2	1,940	205	89	11	0.77
lighttpd	939	171	86	14	0.90
marlin	1,893	622	78	22	0.82
mpsolve	173	488	24	76	0.96
parrot	216	686	23	77	0.96

a set of n consecutive commits of a project, with i=1..n, we checked out each ith commit (using *git reset* –HARD <*ith_commitHash*>) to obtain the state of the project at that commit. We then trained FeatRacer based on feature annotations present in the project at the ith commit and generated a corresponding test dataset using all assets that have changed in the subsequent i+1th commit. If a commit changes existing annotations present in the training set, FeatRacer updates all their records with the latest information to reflect the new annotations.

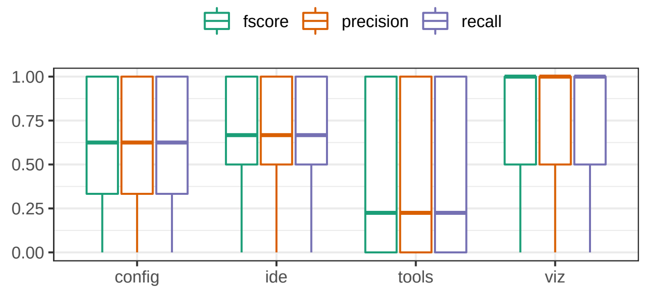
(Step 2) To evaluate FeatRacer and see how it performs on the i+1th commit, we removed all feature annotations from the changed assets recorded by developers in this commit. They were instead used as ground truth. So, we compared FeatRacer's suggestions to what developers actually recorded in the i+1th commit. As explained in Sec. 4.4, we used separate training datasets for each granularity level (folder, file, and fragment) to avoid imbalance and improve predictions.

5.4 Measuring Prediction Performance

We measured prediction performance for all assets that changed in the i+1th commit by calculating the following measures for each asset: $precision$ (relevant labels out of all recommended labels), $recall$ (relevant recommended labels out of all those actually mapped to the asset), and F$_1$-score $((2 * precision * recall)/(precision + recall)))$—we use the harmonic mean to balance between precision and recall. For instance, if an asset is mapped to features A,B,C, and our ranked list of predicted labels has A,B,D; both precision and recall would be 66 %. If the predicted list has A,B, then precision is 100 % while recall is 66 %. For assets that have no annotations in the ground-truth, we consider FeatRacer's prediction to be correct (i.e. 100% precision and recall) if it predicts no annotations for such assets (i.e., the set of predicted labels should be empty), and incorrect (0% precision and recall) otherwise. To calculate the average performance of a classifier on a test dataset (changed assets), we averaged performance values from all assets in the dataset. We visualize the results using standard box-plots. The middle bar is the median, the bottom and top of the box show the interquartile range. The top vertical line indicates the top 25%.



(a) Distribution of folder-level predictions (using RAkELd classifier) from all commits in each project



(b) Distribution of file-level predictions (using RAkELd classifier) from all commits in each project

Figure 8: Results of file-level predictions from all projects

6 Evaluation Results

We now present the answers to our research questions.

6.1 RQ1: Best Performing Classifier

We first present characteristics of our datasets followed by results of our classifier evaluation.

6.1.1 Dataset Characteristics

Table 5 shows characteristics of our evaluation datasets we created for each subject system. We generated one dataset per asset granularity level (folder, file, and fragment) for each commit that modified assets containing feature annotations. All subject systems have annotations within files, so we generated fragment-level datasets for all subjects, but folder- and file-level annotations were only available in the Clafer-WebTools projects. We observe that across all projects the average number of instances per dataset was 1,984 for fragments, 321 for files, and 7 for folders. The average number of labels (software features) in a dataset was 114 for fragments, 20 for files, and 6 for folders; while the maximum number of labels mapped to an asset was 6 for fragments, 3 for files, and 1 for folders. We also observe very low average SCUMBLE [90] values for all datasets. SCUMBLE measures how often dominant and rare labels appear together in a label set (i.e., set of labels for each asset instance). The higher this value, the more difficult it is to learn from a dataset [91]. Table 4 shows the average percentage of annotated vs non-annotated fragments per training dataset in each project against the average F$_1$-score obtained. We observe that some projects have more fragments with missing feature annotations

Table 5: Summary of training datasets generated from subject projects

	project	datasets[1]	avg. instances[2]	avg. labels	min. cardinality[3]	avg. cardinality	max. cardinality	avg. scumble[4]	avg. train time[5]	avg. pred. time[5]
folder	config	132	5	5	1	1.00	1	0.00E+00	0.000	0.000
	ide	196	7	5	1	1.00	1	0.00E+00	0.000	0.000
	viz	436	7	6	1	1.00	1	0.00E+00	0.000	0.000
	tools	125	5	5	1	1.00	1	0.00E+00	0.000	0.000
	summary	889[6]	7[7]	6[7]	1[8]	1.00[7]	1[9]	0.00E+00[7]	0.000[7]	0.000[7]
file	config	132	215	19	1	1.11	2	4.59E-04	0.000	0.000
	ide	196	29	9	1	1.00	1	0.00E+00	0.000	0.000
	viz	436	176	26	1	1.14	3	4.20E-03	0.000	0.000
	tools	125	1,395	17	1	1.00	1	0.00E+00	0.000	0.000
	summary	889[6]	321[7]	20[7]	1[8]	1.06[7]	3[9]	2.13E-03[7]	0.000[7]	0.000[7]
fragment	config	127	403	14	1	1.13	2	1.08E-03	0.000	0.000
	ide	195	365	17	1	1.42	2	1.69E-02	0.000	0.000
	viz	429	717	23	1	1.18	3	1.94E-04	0.000	0.000
	tools	125	1,545	27	1	1.18	2	7.85E-04	0.000	0.000
	blender	392	5,951	133	1	1.04	2	4.23E-03	2.640	0.000
	busybox	207	2,689	274	1	1.12	6	1.10E-03	1.971	0.005
	emacs	220	1,505	192	1	1.01	3	5.40E-05	0.630	0.000
	gimp	68	3,306	262	1	1.00	2	2.82E-18	5.453	0.006
	gnumeric	646	1,835	59	1	1.03	2	1.88E-06	0.447	0.000
	gnuplot	166	4,120	441	1	1.03	3	2.67E-03	18.832	0.028
	irssi	521	1,154	127	1	1.03	3	9.40E-05	0.000	0.000
	libxml2	228	2,150	84	1	1.01	3	1.45E-04	0.416	0.000
	lighttpd	199	1,113	160	1	1.01	3	5.11E-05	0.000	0.000
	marlin	619	2,521	167	1	1.01	3	5.63E-04	5.520	0.002
	mpsolve	192	664	32	1	1.01	2	0.00E+00	0.000	0.000
	parrot	316	903	18	1	1.00	2	1.12E-19	0.000	0.000
	summary	4,650[6]	1,984[7]	114[7]	1[8]	1.08[7]	6[9]	1.37E-03[7]	1.914[7]	0.002[7]
summary all		6,428[6]	1,480[7]	86[7]	1[8]	1.05[7]	6[9]	1.29E-03[7]	1.602[7]	0.001[7]

[1] number of commits from which training datasets were generated; one dataset per commit—some commits analyzed did not affect assets with feature annotations, hence # of datasets $\leq$ # of commits in Table 3 [2] average number of assets in a training dataset
[3] label cardinality–the average number of labels per asset [4] SCUMBLE [90]-concurrence among very frequent and rare labels [5] time in milliseconds
[6] sum [7] average [8] minimum [9] maximum

compared to those with annotations, for instance, project CommonPlatform only has 7 % fragments annotated on average. Other projects have the opposite: for instance, gnuplot has 96 % of the fragments annotated on average. Interestingly however, FeatRacer's prediction performance is good in both cases, with an F_1-score of 94 % and 71 % respectively.

Furthermore, our classifiers took very little time to train and predict feature annotations—they took an average of 1.9ms to train and 0.002ms to predict fragment-level annotations—promising very good usability. These results were obtained from a laptop computer with an Intel i7 processor, 32GB RAM, running Windows 10 Professional.

6.1.2 Predictions Per Granularity Level

We found the best performance when predicting fragment-level annotations (Fig. 9), with an average F_1-score of 90 % across all projects, and a median F_1-score of 100 % for 11 of our 16 projects. Predictions for file-level annotations (Fig. 8b) had an average F_1-score of 65 % across all projects and a median F_1-score ranging from 62.5 % to 100 % for 3 of the 4 ClaferWebTools projects. Folder-level annotations (Fig. 8a) performed worst, with only project ClaferMooVisualizer recording an average F_1-score of 33 %, while the rest had close to 0 %. Project CommonPlatform (tools) is missing in the evaluation for the folder level, since it did not contain any subsequent commits we could test after the first one.

We investigated the low performance for folder-level annotations and found that it can be attributed to the *rare* folder annotations and *fewer folders annotated*. We found, for instance, that ClaferConfigurator has only three commits used as a testing set (i.e., commits adding folder annotations after the initial one), while ClaferIDE has only four such commits in the entire revision history analyzed. In project ClaferConfigurator, the training dataset (commit #16) for the first test commit (#17) only had two instances (two folders) mapped to two features (named *jquery,client*), while the test dataset has two folders newly annotated with two unknown features (*JSClass, icons*). Similarly, for project ClaferIDE the training set (commit #43) for the first test commit (#45) only has four folders annotated, while the test dataset has one folder annotated with an unknown feature (*claferTextEditor*). These results did not improve despite applying the multilabel SMOTE [92] algorithm to our datasets. Project CommonPlatform does not even appear in Fig. 8a since it only had folder-level annotations in one commit from which the first training dataset was generated, and had no subsequent commits we could test—i.e., it had no newer commits that added folder-level annotations. We expected this performance since commits modify existing files much more frequently than adding new files or folders. Therefore, there are more fragment-level annotations on which to train the classifiers.

6.1.3 Best Performing Classifier

Table 6 presents a summary of the performance scores (F_1-score, precision, and recall) for each of the three classifiers (BR, LP, and RAkELd). We present both the median and mean across all datasets considered for each granularity level of each project. We highlight (in bold) the maximum values for the top performing classifier in each case.

We found that RAkELd had the best overall performance when predicting feature annotations for all levels. However, BR and LP had comparable performance. In fact, considering

Table 6: Summary of prediction results (mean and median values across all commits)

	project	BR						LP						RAkELd					
		F_1-score		Precision		Recall		F_1-score		Precision		Recall		F_1-score		Precision		Recall	
		Avg	Med	Avg	Med	Avg	Med	Avg	Med	Avg	Med	Avg	Med	Avg	Med	Avg	Med	Avg	Med
folder	config	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	ide	0.00	0.00	0.00	0.00	0.00	0.00	**0.06**	0.00	**0.06**	0.00	**0.06**	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	viz	**0.33**	0.00	**0.33**	0.00	**0.33**	0.00	0.00	0.00	0.00	0.00	0.00	0.00	**0.33**	0.00	**0.33**	0.00	**0.33**	0.00
file	config	**0.59**	**0.63**	**0.59**	**0.63**	**0.59**	**0.63**	0.55	0.59	0.55	0.59	0.55	0.59	**0.59**	**0.63**	**0.59**	**0.63**	**0.59**	**0.63**
	ide	0.60	**0.67**	0.60	**0.67**	0.60	**0.67**	0.54	0.50	0.54	0.50	0.54	0.50	**0.63**	**0.67**	**0.63**	**0.67**	**0.63**	**0.67**
	viz	0.75	**1.00**	0.75	**1.00**	0.75	**1.00**	**0.76**	**1.00**	**0.76**	**1.00**	**0.76**	**1.00**	0.75	**1.00**	0.75	**1.00**	0.75	**1.00**
	tools	**0.39**	**0.23**	**0.39**	**0.23**	**0.39**	**0.23**	0.38	0.00	0.38	0.00	0.38	0.00	**0.39**	**0.23**	**0.39**	**0.23**	**0.39**	**0.23**
fragment	config	**0.97**	**1.00**	**0.97**	**1.00**	**0.97**	**1.00**	0.96	**1.00**	0.96	**1.00**	0.96	**1.00**	0.96	**1.00**	0.96	**1.00**	0.96	**1.00**
	ide	0.92	**1.00**	0.91	**1.00**	0.91	**1.00**	0.92	**1.00**	**0.92**	**1.00**	**0.92**	**1.00**	**0.92**	**1.00**	0.91	**1.00**	0.91	**1.00**
	viz	**0.95**	**1.00**	**0.95**	**1.00**	**0.95**	**1.00**	0.94	**1.00**	0.94	**1.00**	0.94	**1.00**	0.95	**1.00**	0.95	**1.00**	0.95	**1.00**
	tools	**0.94**	**1.00**	**0.94**	**1.00**	**0.94**	**1.00**	0.94	**1.00**	0.94	**1.00**	0.94	**1.00**	**0.94**	**1.00**	**0.94**	**1.00**	**0.94**	**1.00**
	blender	**0.90**	**1.00**	**0.90**	**1.00**	**0.90**	**1.00**	0.90	**1.00**	0.90	**1.00**	0.90	**1.00**	0.90	**1.00**	0.90	**1.00**	0.90	**1.00**
	busybox	0.79	0.83	0.79	0.83	0.79	0.83	0.75	0.79	0.75	0.79	0.75	0.79	**0.80**	**0.86**	**0.79**	**0.86**	**0.79**	**0.86**
	emacs	**0.76**	**0.75**	**0.76**	**0.75**	**0.76**	**0.75**	0.71	0.73	0.71	0.73	0.71	0.73	0.76	**0.75**	0.76	**0.75**	0.76	**0.75**
	gimp	0.93	**1.00**	0.93	**1.00**	0.93	**1.00**	0.91	**1.00**	0.91	**1.00**	0.91	**1.00**	**0.93**	**1.00**	**0.93**	**1.00**	**0.93**	**1.00**
	gnumeric	0.95	**1.00**	0.95	**1.00**	0.95	**1.00**	0.95	**1.00**	0.95	**1.00**	0.95	**1.00**	**0.96**	**1.00**	**0.96**	**1.00**	**0.96**	**1.00**
	gnuplot	0.71	**0.67**	0.71	**0.67**	0.71	**0.67**	0.69	**0.67**	0.69	**0.67**	0.69	**0.67**	**0.71**	**0.67**	**0.71**	**0.67**	**0.71**	**0.67**
	irssi	0.97	**1.00**	**0.97**	**1.00**	**0.97**	**1.00**	0.97	**1.00**	0.97	**1.00**	0.97	**1.00**	**0.97**	**1.00**	0.97	**1.00**	**0.97**	**1.00**
	libxml2	**0.78**	**0.80**	**0.78**	**0.82**	**0.78**	**0.80**	0.77	**0.80**	0.77	0.80	0.77	0.80	0.77	**0.80**	0.77	0.80	0.77	0.80
	lighttpd	**0.90**	**1.00**	**0.90**	**1.00**	**0.90**	**1.00**	0.87	**1.00**	0.87	**1.00**	0.87	**1.00**	**0.90**	**1.00**	**0.90**	**1.00**	**0.90**	**1.00**
	marlin	**0.82**	**1.00**	**0.82**	**1.00**	**0.82**	**1.00**	0.78	**1.00**	0.78	**1.00**	0.78	**1.00**	0.82	**1.00**	0.82	**1.00**	0.82	**1.00**
	mpsolve	0.96	**1.00**	0.96	**1.00**	0.96	**1.00**	0.95	**1.00**	0.95	**1.00**	0.95	**1.00**	**0.96**	**1.00**	**0.96**	**1.00**	**0.96**	**1.00**
	parrot	0.96	**1.00**	0.96	**1.00**	0.96	**1.00**	0.95	**1.00**	0.95	**1.00**	0.95	**1.00**	**0.96**	**1.00**	**0.96**	**1.00**	**0.96**	**1.00**

Med–Median, Avg–Average

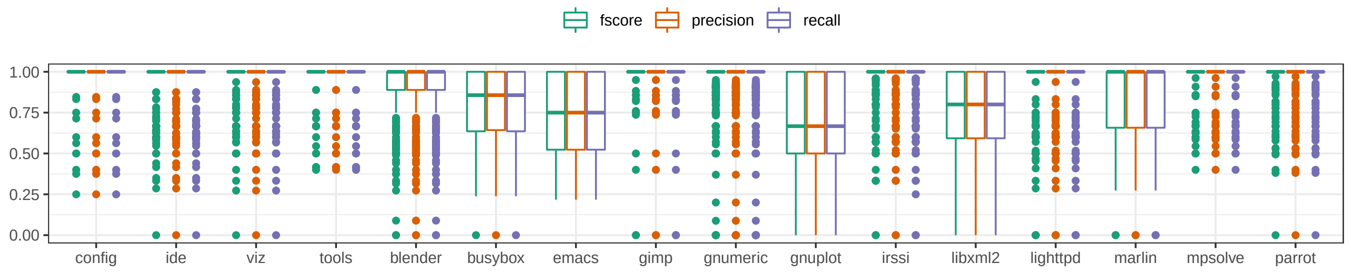


Figure 9: Distribution of fragment-level predictions (using RAkELd classifier) from all commits in each project

all projects, we found that, for fragment-level annotations, RAkELd and BR had an average F_1-score of 90 %, and LP had 88 %; while for file-level annotations RAkELd had an average F_1-score of 65 %, BR of 64 %, and LP of 63 %.

6.1.4 Discussion

Our results show that FeatRacer can reliably predict feature locations—especially for fragment-level assets using the RAkELd classifier, since it was the most accurate and robust to the size of training data available. While our results show marginal differences between BR and RAkELd, large numbers of labels may negatively impact the training time for BR, since it does not scale—the highest number of labels (features mapped to assets) in our evaluation was 607 from project busybox's first 260 commits (see Table 3).

The high precision and recall values we obtained reinforce the importance of using developer's own recorded feature locations (i.e., recorded while the feature is still fresh in the memory) to locate features during maintenance. Furthermore, compared to the text similarity metrics we engineered in our previous work [23], our classifiers performed best when predicting feature locations using our asset metrics in Table 1, which do not rely on code structure, but quantify aspects of the development process to associate features with their assets. The text similarity and code structure metrics were not good at measuring the relatedness of assets of a given feature, since our assets are of varying granularity. For instance, the cosine similarity between two code fragments, one with 2 lines of code, and another with 100 lines of code, may not be a good indicator of whether or not the fragments belong to the same feature, despite them being written by the same set of developers, and often modified together. Thus, our process-based asset metrics performed better. We also note that training our classifiers on both annotated and non-annotated assets significantly improved prediction performance (see Table 4): In our preliminary experiments, we trained the classifiers only on annotated assets but predicted for both annotated and non-annotated assets and found very poor performance with median F_1-score < 50 % for fragment-level annotations from all projects.

The poor performance, especially for the folder level is expected, considering that in practice (based on our real-world projects) folder annotations are rare and few.

6.2 RQ2: Best Metrics for Predictions

To investigate which of our asset metrics in Table 1 best characterize assets of a given feature, we used the filter approach *ReliefF* with Binary Relevance and Label Powerset

Table 7: Average F_1-score obtained after predicting with the top 4, 6, 8, and 11 metrics, respectively

project	BR				LP				RAk				ranking of metrics
	4	6	8	11	4	6	8	11	4	6	8	11	
config	**0.97**	**0.97**	**0.97**	**0.97**	**0.97**	**0.97**	**0.97**	0.96	**0.97**	**0.97**	**0.97**	0.96	*nff, dnfma, ddev, comm, dcont, nfma, ccc, hdcont, accc, csdev, nloc*
ide	**0.92**	**0.92**	**0.92**	0.91	**0.92**	**0.92**	**0.92**	0.92	**0.92**	**0.92**	**0.92**	0.91	*nff, nfma, dnfma, dcont, comm, hdcont, accc, ddev, ccc, nloc, csdev*
viz	**0.96**	**0.96**	**0.96**	0.95	**0.96**	**0.96**	**0.96**	0.94	**0.96**	**0.96**	**0.96**	0.95	*csdev, dcont,nff, nfma, dnfma, hdcont, ccc, accc, ddev, comm, nloc*
tools	**0.94**	**0.94**	**0.94**	**0.94**	**0.94**	**0.94**	**0.94**	**0.94**	**0.94**	**0.94**	**0.94**	**0.94**	*nff, ccc, accc, dcont, hdcont, dnfma, nfma, ddev, comm, nloc, csdev*
blender	**0.91**	**0.91**	**0.91**	0.90	**0.91**	**0.91**	**0.91**	0.90	**0.91**	**0.91**	**0.91**	0.90	*nfma, dnfma, dcont, nff, hdcont, ccc, accc, comm, ddev, nloc, csdev*
busybox	**0.80**	0.79	0.78	0.79	**0.80**	**0.80**	0.75	0.75	**0.80**	0.79	0.78	0.79	*nff, dnfma, ccc, nfma, accc, dcont, hdcont, nloc, ddev, comm, csdev*
emacs	**0.77**	**0.77**	0.76	0.76	**0.77**	**0.77**	0.71	0.71	**0.77**	**0.77**	0.76	0.76	*hdcont, dcont, dnfma, nfma, nff, ccc, accc, nloc, comm, ddev, csdev*
gimp	**0.94**	**0.94**	0.93	0.93	**0.94**	**0.94**	0.91	0.91	**0.94**	**0.94**	0.93	0.93	*nfma, dnfma, ccc, nff, accc, dcont, hdcont, nloc, ddev, comm, csdev*
gnu-meric	**0.97**	**0.97**	**0.97**	0.95	**0.96**	**0.96**	**0.96**	0.95	**0.97**	**0.97**	**0.97**	0.96	*nff, nfma, hdcont, dnfma, dcont, ccc, accc, comm, nloc, ddev, csdev*
gnuplot	**0.71**	**0.71**	**0.71**	**0.71**	0.70	**0.71**	**0.71**	0.69	**0.71**	**0.71**	**0.71**	**0.71**	*nff, dcont, hdcont, ddev, ccc, accc, nloc, dnfma, nfma, comm, csdev*
irssi	**0.97**	**0.97**	**0.97**	**0.97**	**0.97**	**0.97**	**0.97**	0.97	**0.97**	**0.97**	**0.97**	**0.97**	*nfma, dnfma, ccc, accc, hdcont, dcont, nff, nloc, comm, ddev, csdev*
libxml2	0.77	0.77	**0.78**	**0.78**	**0.77**	**0.77**	0.76	0.77	**0.77**	**0.77**	**0.77**	**0.77**	*nff, dcont, dnfma, hdcont, nfma, ccc, nloc, accc, ddev, comm, csdev*
lighttpd	**0.90**	**0.90**	**0.90**	**0.90**	**0.88**	**0.88**	**0.88**	0.87	**0.90**	**0.90**	**0.90**	**0.90**	*nff, ccc, dcont, nloc, dnfma, accc, hdcont, nfma, ddev, comm, csdev*
marlin	**0.83**	**0.83**	**0.83**	0.82	**0.83**	**0.83**	**0.83**	0.78	**0.83**	**0.83**	**0.83**	0.82	*csdev, nfma, nff, dnfma, ccc, accc, hdcont, ddev, dcont, comm, nloc*
mpsolve	**0.96**	**0.96**	**0.96**	**0.96**	**0.96**	**0.96**	**0.96**	0.95	**0.96**	**0.96**	**0.96**	**0.96**	*dnfma, nfma, ccc, nff, accc, dcont, hdcont, comm, nloc, csdev, ddev*
parrot	**0.96**	**0.96**	**0.96**	**0.96**	**0.96**	**0.96**	**0.96**	0.95	**0.96**	**0.96**	**0.96**	**0.96**	*ccc, dnfma, accc, nfma, hdcont, nff, dcont, ddev, nloc, comm, csdev*

classifiers [83] to rank the metrics. As stated in Sec. 5.2, filter approaches to attribute selection use the characteristics of a dataset to select the best attributes and exclude others.

6.2.1 Top Asset Metrics per Granularity Level

We averaged the rank of each metric from all projects and show the overall ranking of the top 8 metrics in Table 8 for the three different granularity levels.

We dove deeper into the fragment-level, which had the better performance. Table 7 shows the F_1-score each classifier obtained for fragment-level annotations in each project when predicting with the top 4, 6, 8, and 11 metrics. The last column shows how all 11 metrics were ranked by *ReliefF* in each project. We observe that the RAkELd classifier had the best performance in all projects when predicting with the top 4 metrics. However, the differences between the F_1-score for each of the metric sets are marginal—mostly 0.01. BR and LP had similar performance.

6.2.2 Discussion

Our results suggest that the top 4 metrics to predict feature annotations of each fragment are the number of features in the parent file of the fragment (NFF), the number of features often changed with the fragment (DNFMA, NFMA), and the contribution (DCONT) of the developers of the fragment. This also suggests that modularized code supports better prediction performance for fragment-level feature annotations. For file-level annotations, the top 4 metrics are the number of developers (DDEV) of the file, the number of assets modified with the file (ACCC, CCC), and the contribution (DCONT) of the developers of the file. Lastly, developer contribution (DCONT, HDCONT), similarity of developer names (CSDEV), and the number of assets modified with a folder (CCC) constitute the top 4 metrics for predicting feature annotations for each folder. Therefore, for all granularity levels, the number of features in a file (NFF), the number of features modified when an asset is modified (DNFMA, NFMA), the number of assets modified together with an asset (CCC, ACCC), the number of developers of an asset (DDEV), and their contribution to the project (DCONT, HDCONT) constitute the top predictors of feature annotations for the asset.

Table 8: Average ranking of the best eight asset metrics per granularity level across all projects

level	1	2	3	4	5	6	7	8
folder	DCONT	HDCONT	CSDEV	CCC	COMM	ACCC	DDEV	DNFMA
file	DDEV	ACCC	CCC	DNFMA	DCONT	NFMA	HDCONT	COMM
frag.	NFF	DNFMA	NFMA	DCONT	CCC	HDCONT	ACCC	DDEV

The ranking of the metrics is as we expected. For instance, fragments of a given feature are more likely to be found in a single file (for feature code not very scattered across several files). Hence, such fragments are more likely to have about the same number of features in their parent file (NFF). This in turn increases the likelihood of such features being modified together (DNFMA, NFMA). For a file asset, however, the number of developers of the file (DDEV) and the number of files in the changeset (CCC, ACCC) would be more useful for associating files that belong to a given feature, since files implementing the same feature are more likely to be modified together quite often by about the same number of developers.

6.3 RQ3: Influence of Commit Practices and Performance over Time

For each of our projects, we analyzed what influence developers' commit practices, such as adding or removing few or many lines of code, have on FeatRacer's prediction performance. As shown in Fig. 7, we observed several drops in performance for all our projects. Our manual inspection

Table 9: Significant Spearman correlations between commit metrics and FeatRacer's fragment- level prediction performance for all projects

metric	measure	coef	p_value
maximum lines added (mLA)	recall	-0.121	1.84E-13
maximum lines added (mLA)	fscore	-0.121	2.02E-13
total lines added (tLA)	recall	-0.120	2.49E-13
total lines added (tLA)	fscore	-0.120	2.75E-13
maximum lines added (mLA)	precision	-0.119	4.46E-13
total lines added (tLA)	precision	-0.118	6.26E-13
total code churn (tCh)	recall	-0.115	3.54E-12
total code churn (tCh)	fscore	-0.114	3.97E-12
total code churn (tCh)	precision	-0.113	7.99E-12
average lines added (aLA)	recall	-0.110	2.01E-11
average lines added (aLA)	fscore	-0.110	2.19E-11
average lines added (aLA)	precision	-0.108	4.56E-11
maximum hunk size (mHS)	recall	-0.108	5.03E-11
maximum hunk size (mHS)	fscore	-0.108	5.37E-11
average code churn (aCh)	recall	-0.107	8.45E-11
average code churn (aCh)	fscore	-0.107	9.40E-11
maximum hunk size (mHS)	precision	-0.106	1.05E-10
total hunks (tH)	recall	-0.106	1.41E-10
total hunks (tH)	fscore	-0.105	1.53E-10
total files in changeset (tFC)	recall	-0.105	1.71E-10
average code churn (aCh)	precision	-0.105	1.76E-10
total files in changeset (tFC)	fscore	-0.105	1.85E-10
total hunks (tH)	precision	-0.104	2.75E-10
total files in changeset (tFC)	precision	-0.104	3.35E-10
average hunk size (aHS)	recall	-0.081	9.11E-07
average hunk size (aHS)	fscore	-0.081	9.61E-07
average hunk size (aHS)	precision	-0.079	1.60E-06
total files modified (tFM)	precision	-0.077	2.71E-06
total files modified (tFM)	recall	-0.077	2.80E-06
total files modified (tFM)	fscore	-0.077	2.88E-06
total files deleted (tFD)	recall	-0.063	1.34E-04
total files deleted (tFD)	fscore	-0.063	1.43E-04
total files deleted (tFD)	precision	-0.059	3.86E-04
average number of hunks (aH)	recall	-0.054	9.80E-04
average number of hunks (aH)	fscore	-0.054	1.02E-03
average number of hunks (aH)	precision	-0.053	1.28E-03
total lines removed (tLR)	recall	-0.048	3.46E-03
total lines removed (tLR)	fscore	-0.048	3.62E-03
maximum lines removed (mLR)	recall	-0.047	4.65E-03
maximum lines removed (mLR)	fscore	-0.047	4.85E-03
total lines removed (tLR)	precision	-0.046	5.20E-03
maximum lines removed (mLR)	precision	-0.045	6.84E-03
total files added (tFA)	recall	-0.035	3.35E-02
total files added (tFA)	fscore	-0.035	3.44E-02
average lines removed (aLR)	recall	-0.034	3.83E-02
average lines removed (aLR)	fscore	-0.034	3.95E-02

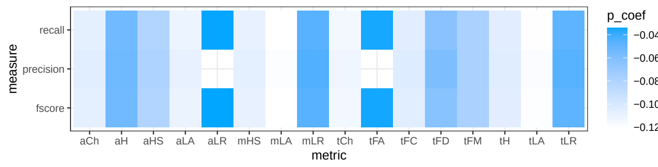


Figure 10: All significant correlations between commit metrics and FeatRacer's fragment-level prediction performance.

of the commits revealed that drops in performance mostly occurred for commits where several new files or lines of code were added. This pattern is expected, since developers often refactor code or import files into their projects.

6.4 Correlation of Commit Metrics and Performance

However, to understand whether these structural changes have a statistically significant impact on FeatRacer's performance, we measured correlations using Spearman's correlation coefficient, which does not assume normality of data. For each project and commit, we counted the number of lines added (LA), lines removed (LR), code churn (Ch), i.e., added minus deleted lines, number of files added (FA), files deleted (FR), files modified (FM), files renamed (FRe), files in the change-set (FC), the number of hunks (H) i.e., consecutive changes in a file, and the sizes of the hunks (HS). For some of the metrics, we measured the average (aCh, aH, aHS, aLA, and aLR), the maximum (mHS, mLA, mLR), and the total (tCh, tFA, tFC, tFM, tFRe, tH, tLA, tLR). For instance, we measured average number of hunks in files modified in a commit (aH), maximum size of a hunk in a commit (mHS), and total number of hunks in a commit (tH). We only calculated these metrics for commits where FeatRacer had prediction results in each project. Please note that these metrics measure the amount of structural changes introduced by each commit, but they do not characterize related assets of a feature, so we did not use them in FeatRacer's model for predicting feature locations as we do for our asset metrics in Table 1.

Table 10: Significant Spearman's correlations between FeatRacer asset metrics (averaged from all fragments per commit) and FeatRacer's fragment- level prediction performance for all projects

metric	measure	coef	p_value
features in parent file (NFF)	fscore	-0.257	1.11E-68
features in parent file (NFF)	precision	-0.257	1.18E-68
features in parent file (NFF)	recall	-0.257	1.65E-68
all features modified with the asset (DNFMA)	fscore	-0.249	4.93E-64
all features modified with the asset (DNFMA)	precision	-0.248	6.77E-64
all features modified with the asset (DNFMA)	recall	-0.248	8.65E-64
features modified in current commit (NFMA)	fscore	-0.232	5.67E-56
features modified in current commit (NFMA)	precision	-0.232	8.59E-56
features modified in current commit (NFMA)	recall	-0.232	1.03E-55
all asset commits (COMM)	precision	-0.139	8.62E-21
all asset commits (COMM))	recall	-0.139	9.54E-21
all asset commits (COMM)	fscore	-0.139	1.06E-20
average change-set size for all commits (ACCC)	fscore	-0.137	3.41E-20
average change-set size for all commits (ACCC)	precision	-0.137	3.95E-20
average change-set size for all commits (ACCC)	recall	-0.136	4.66E-20
change-set size for current commit (CCC)	fscore	-0.116	6.03E-15
change-set size for current commit (CCC)	precision	-0.116	7.01E-15
change-set size for current commit (CCC)	recall	-0.116	7.91E-15
all asset developers (DDEV))	precision	-0.086	8.61E-09
all asset developers t (DDEV))	recall	-0.086	9.22E-09
all asset developers (DDEV))	fscore	-0.086	9.74E-09
developers name similarity(CSDEV))	precision	0.052	5.16E-04
developer name similarity(CSDEV))	recall	0.052	5.22E-04
developer name similarity(CSDEV))	fscore	0.052	5.35E-04
highest developer contribution (HDCONT)	precision	-0.042	4.98E-03
highest developer contribution (HDCONT)	fscore	-0.041	6.09E-03
highest developer contribution (HDCONT)	recall	-0.040	7.07E-03
average developer contribution (DCONT)	precision	-0.038	1.18E-02
average developer contribution (DCONT)	fscore	-0.037	1.42E-02
average developer contribution (DCONT)	recall	-0.036	1.62E-02

Figure 10 and Table 9 show significant correlations (p-values < 0.05) for fragment-level annotation predictions from all 16 projects. We observe that the above commit metrics are weakly correlated with FeatRacer's performance, having coefficients less or equal to 12 %. All the significant correlations are negative, indicating that these metrics potentially negatively impact FeatRacer's prediction performance. While lines added, code churn, and maximum hunk size are the top metrics that negatively impact performance overall (F_1-score,precision, and recall), the total number of files added (tFA) and average lines removed (aLR) do not significantly impact precision as they do recall (see the white regions on the bars in Fig. 10 and missing values for precision for the two metrics in Table 9).

We also obtained correlations between our asset metrics in Table 1 and FeatRacer's performance for fragment-level

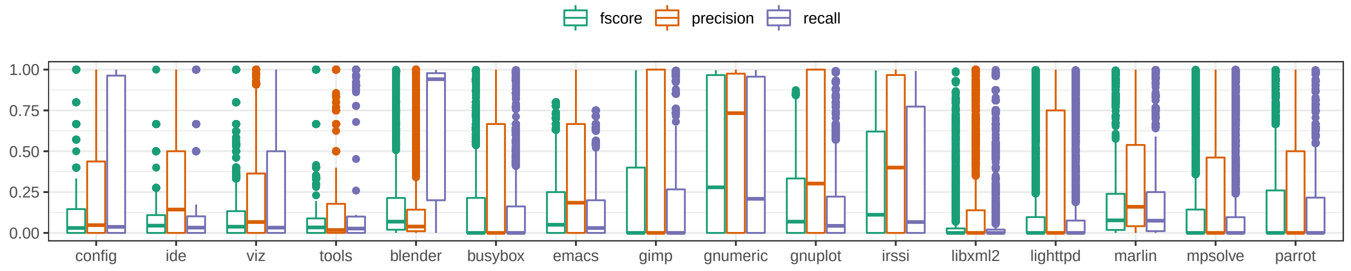


Figure 11: LSI performance for all projects

predictions—we averaged the metric values for all fragments in each commit's fragment-level dataset and also averaged the performance measures (F_1-score, precision, recall). Table 10 shows the results. All ten of our process-based asset metrics are significantly correlated with FeatRacer's prediction performance. Interestingly, nine of the ten metrics are negatively correlated with prediction performance.

6.4.1 Discussion

Considering the significant correlations shown in Table 9, our results suggest that FeatRacer's performance is likely to be low when the number of lines added (mLA, aLA, tLA) or removed (mLR, tLR) is high, when the hunks of a commit (mHS, aHS) are very big, or when the variations between added and removed lines (aCh) are large. The results of Table 10 imply that FeatRacer is better able to make accurate feature location predictions for fragments when there are fewer features within the parent file of the fragment (NFF), when there are fewer features the fragment is modified with in commits (DNFMA, NFMA), when there are fewer developers dedicated to modifying the fragment (DDEV, HDCONT, DCONT), when there are fewer assets the fragment is modified together with in a commit (CCC, ACCC), and when the fragment is often modified by the same developer (CSDEV). This result agrees with findings from our manual investigation of the commits. However, the weak correlations indicate that these commit practices are less likely to limit FeatRacer's usefulness.

As may be noticed from Fig. 7, FeatRacer does not take long to give accurate predictions depending on the available training data in the initial commit (with feature annotations) and characteristics of the test data. Some projects, like ClaferConfigurator, gimp, mpsolve, CommonPlatform, and ClaferMooVisualizer, already had F_1-score of 100 % from the first test commit. For instance, ClaferConfigurator's initial training set is from the 11th commit in the revision history. It had a total of 153 fragments, 7 annotated and 146 not annotated. The first test commit (commit #11) affected two fragments that were not annotated. FeatRacer was able to correctly predict labels for these two fragments with an F_1-score of 100%. However, when many non-annotated fragments are introduced in a commit, e.g., commit #13, which introduced 68 new non-annotated fragments, the performance dropped to 40%. Some projects, like blender, however, started with a low performance: F_1-score of 64% in the first test commit, followed by 50% at commit #4, then jumped to F_1-score of 100% at commit #6, which introduced 11 new fragments, 8 of which had annotations and 3 that were missing annotations. Therefore, though FeatRacer's predictions may fluctuate due to reasons we have discussed, it does not take long before it gives the developer reliable suggestions for missing feature locations.

Table 11: Average LDA performance

project	commits	precision	recall	f1-score
config	28	0.000	0.000	0.000
ide	59	0.032	0.020	0.025
viz	125	0.035	0.039	0.037
busybox	206	0.000	0.000	0.000
emacs	220	0.000	0.000	0.000
gimp	68	0.004	0.013	0.006
gnumeric	646	0.000	0.000	0.000
gnuplot	166	0.001	0.003	0.001
irssi	422	0.003	0.001	0.002
libxml2	228	0.005	0.003	0.004
lighttpd	198	0.015	0.008	0.010
mpsolve	192	0.020	0.015	0.017
parrot	316	0.003	0.003	0.003
average	221	0.009	0.008	0.008

6.5 RQ4: FeatRacer vs. Traditional Feature Location

We compared FeatRacer to LSI [19] and LDA [86]—two of the best and widely used automated techniques in feature location studies [20], [21]—to assess the percentage of correctly mapped assets by each technique. For the LSI, we used an implementation provided by Apache Lucene, which also allows us to query for features.

6.5.1 Performance of LSI and LDA

Figure 11 shows LSI's performance in all projects and commits. As expected, LSI performed poorly, with an average F_1-score of 19 %, precision of 34 %, and recall of 21 % across all projects. In contrast, using the RAkELd classifier, FeatRacer had an average of 90 % in all performance measures from all projects. Since evaluating LDA on large projects with a lot of features such as Blender or Marlin took too long, we decided to test LDA on 13 of the 16 projects. Table 11 shows the measured results for the performance of LDA over these projects. Here, it achieved an average of F_1-score of 0.8 %, precision of 0.8 % and recall of 0.9 %. The low values are not surprising since in our scenario, LDA requires to make predictions on textual data of code fragments, which are usually too short or empty after preprocessing, to reliably map assets to features. This underlines the benefits of using the metrics as opposed to structural data.

6.5.2 Discussion

Our results demonstrate the benefits that FeatRacer offers through incremental, immediate recording of locations. FeatRacer has about 3x more precision and 4.5x recall than

LSI. LSI and LDA, like other automated feature location techniques [17], rely on developers formulating search queries to retrieve a feature's assets. Without recorded feature locations, search results will only be as precise as the query. Formulating queries every time a developer needs to locate features is time-consuming [14]. This is not the case when searching by feature names which are intuitive to developers and are already recorded. For instance, in a mail-client application (see Fig. 1), a developer might want to locate all code related to the feature named `Send Mail`, rather than writing a complex query or even analyzing complex method calls and code dependencies to locate the same feature.

In the case of LDA, it is not able to reliably map the query to a fragment if the feature's name itself does not appear in the asset. Further, as the number of topics is large compared to the contained vocabulary, the assigned weights of each term in each topic is low. Additionally, as the added code fragments are often rather short (many contain less than 10 terms), the resulting likelihood for a given topic will be low. We experimented with different setups to maximize the performance of LDA. As such, we only removed terms that do not hold information to a specific feature, such as programming language related control sequences or english stopwords, to preserve as much usable information as possible. We refrained from performing any lemmatization, as valuable information such as variable names might be deleted. We tried to extract further information by splitting camel cases and removing underscores. However, in our scenario, predictions must be made on short fragments, which is not effective as opposed to doing predictions on e.g., method-level [21]. This setup shows the difficulties automated feature recovery techniques face when trying to locate features on the fragment-level.

We note that our comparison may not be fair, since LSI's and LDA's results are only as precise as the search query supplied. Rather than formulate complex search queries describing the desired features, we only searched by feature names, which, in practice, would be more intuitive and less demanding to a developer. Performing a comparison where more complex queries are used for them would require a separate study. However, as stated in Sec. 2, Perez et al. [20] already demonstrated the ineffectiveness of automated feature location using LSI, while Ji et al. [30] showed the low costs and saved effort for recovering feature locations, of recording annotations during development. Perez et al. [20] showed that there can often be mismatches between terms used in the search query and what is in the source code, and that even experienced developers may not have the full knowledge of the source code to precisely retrieve all locations of a feature. While specifying better queries or providing better feature descriptions could improve LSI and LDA slightly, it would require extra effort from a developer, further emphasizing the advantage of FeatRacer.

7 Threats to Validity

Internal Validity. To mitigate the risk of bugs in FeatRacer, we performed reviews to ensure that metrics and related accuracy measures are calculated as expected. The authors held several meetings to review the implementation of FeatRacer and results obtained from the evaluation. We evaluate FeatRacer on 16 projects from different domains, using parts of their evolution histories (avg. 3 years). As stated in Sec. 5.1, projects from ClaferWebTools were annotated previously in a user study and the other projects have annotations from their original developers, thus eliminating bias. Our evaluation is independent of who annotated code, since we use the annotations as feature labels in our dataset, but analyze characteristics of the code itself.

External Validity. Even though, our evaluation relies on 16 projects from different domains, and three main languages (JavaScript, C, and C++), with each project using embedded feature annotations, FeatRacer is language-independent and can be applied to any project in any domain. Since each project and domain is different, FeatRacer by design adapts itself to each specific project by learn the characteristics of features and their locations in the project at hand, and using them for feature location recommendations. For projects with a larger number of software features than we analyzed in our evaluation, FeatRacer can be configured to use more scalable classifiers only, such as RAkELd, instead of Binary Relevance or others that may not be appropriate.

8 Related Work

A large number of feature location techniques exists, as shown by a survey which found 24 [17]). In the following, we discuss those that are most closely related to FeatRacer.

Seiler et al. [55] trace software features (with tool support they provide) in issue tracking systems using Java annotation tags (e.g., `@Feature ("x")`). The annotations are used for source code, requirements and work items. They also implement a recommender system that suggests annotations. However, the annotation system is Java-dependent, does not trace other levels such as folders and files, and the recommender supports only single-label classification. FeatRacer is language-independent, systematically evaluated on real datasets, and handles multi-label classification, since assets often belong to more than one feature.

LEADT [93] and Suade [94] recommend related code artifacts. LEADT retroactively retrieves variable features (that may or may not be present in some products) from Java code by exploiting feature dependencies. Suade relies on method-call and field-access dependency relations to retrieve related code elements. However, it requires developers to explicitly trigger the recommendation and specify a set of relevant methods and fields from which the recommendations can be obtained. FeatRacer is language-independent and requires developers to only accept or reject recommendations. Furthermore, unlike LEADT, which targets systems in which variation points are already specified in the source code, FeatRacer supports common and variable features.

Several tools have been proposed to visualize software features of a project as a way to encourage developers to record features. While most tools, such as Pleuss et al.'s [95] and Kästner et al.'s [96] target features in software product lines or highly configurable systems, a few target embedded feature annotations. FLORiDA [34] is a standalone Java application that provides several feature views and metrics, such as size of a feature, files mapped to a feature, scattering and tangling degree of a feature; HAnS [32] is an IntelliJ plugin, and FeatureDashboard [33] an Eclipse plugin providing views and metrics similar to FLORiDA; while FeatureVista [35] is a standalone application that provides interactive feature views for object oriented projects. However, these tools do not provide the necessary support to nudge developers to proactively and continuously record features and their locations.

9 Conclusion

We presented FeatRacer, a programming-language-independent recommender that combines the ideas of feature location recording and automated feature location recovery in a way that addresses both their challenges. FeatRacer relies on the philosophy that it is more promising to support developers in recording good feature locations instead of trying to recover highly domain- and developer-specific information—features—from codebases retroactively. So, instead of loosing and recovering this information, FeatRacer helps developers record features proactively during development. Better recordings, in turn, improve predictions for parts of each commit. As such, FeatRacer presents a novel perspective on the fundamental *feature-location problem* in software engineering.

Compared to traditional automated techniques, FeatRacer learns from the commit history of the project, thereby learning the specific use and characteristics of features and providing better recommendations. This means developers are able to record better feature locations. We showed that the recommendations in fact outperform common traditional feature-location techniques relying on LSI and LDA. While the latter could perhaps be improved by specifying much better queries or providing nearly perfect feature descriptions, it would require substantial extra effort from developers to compete with FeatRacer, which outperforms LSI by a factor of 3 on precision and a factor of 4.5 on recall. While FeatRacer recorded average precision and recall of 90 % from all 16 projects, LSI recorded average precision of 34 % and recall of 21 % on the same projects. LDA on the other hand only achieves an average precision of 0.9 % and recall of 0.8 %. We also identified the best prediction granularity, the best classifiers, and the impact of commit practices.

We hope to inspire further work in the community on supporting developers recording feature information early and continuously instead of trying to recover very domain-specific information that is in the mind of developers during development and is lost when not recorded. A valuable extension of FeatRacer would be to support cases where an annotated asset changes in a way that requires changing the annotation as well, or adding an additional one. Currently, FeatRacer relies on the developer to provide correct mappings or to correct them by nudging the developer when she forgot to annotate. Another valuable avenue is to integrate FeatRacer into mainstream development tooling, such as IDEs or version-control systems, such as Git. In fact, for the former, we plan to integrate FeatRacer into IDE tooling, especially the IntelliJ plugin HAnS [32]. For the latter, FeatRacer can support future variation-control systems [97], [98]—version-control systems that work on the level of features and allow checking out and committing combinations of features. These have been designed by the versioning community since the 1980s, but never made it into mainstream, partly due to the feature location and traceability problem.

Acknowledgments

Supported by the Swedish Research Council Vetenskapsrådet, by a fellowship of the Royal Swedish Academy of Sciences and of the Wallenberg Foundation, and by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972.

References

[1] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (foda) feasibility study," Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, Tech. Rep., 1990.

[2] 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.

[3] C. Larman, *Scaling lean & agile development: thinking and organizational tools for large-scale Scrum*. Pearson Education India, 2008.

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

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

[6] A. Wąsowski and T. Berger, *Domain-Specific Languages: Effective Modeling, Automation, and Reuse*. Springer Nature, 2023, ch. Software Product Lines, pp. 395–435.

[7] F. J. Van der Linden, K. Schmid, and E. Rommes, *Software product lines in action: the best industrial practice in product line engineering*. Springer Science & Business Media, 2007.

[8] S. Apel and C. Kästner, "An overview of feature-oriented software development." *Journal of Object Technology*, vol. 8, no. 5, pp. 49–84, 2009.

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

[10] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wąsowski, "A Survey of Variability Modeling in Industrial Practice," in *VaMoS*, 2013.

[11] T. Berger, J.-P. Steghöfer, T. Ziadi, J. Robin, and J. Martinez, "The state of adoption and the challenges of systematic variability management in industry," *Empirical Software Engineering*, vol. 25, pp. 1755–1797, 2020.

[12] D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated analysis of feature models 20 years later: A literature review," *Information Systems*, vol. 35, no. 6, pp. 615–636, 2010.

[13] J. A. Galindo, D. Benavides, P. Trinidad, A.-M. Gutiérrez-Fernández, and A. Ruiz-Cortés, "Automated analysis of feature models: Quo vadis?" *Computing*, vol. 101, pp. 387–433, 2019.

[14] J. Wang, X. Peng, Z. Xing, and W. Zhao, "How Developers Perform Feature Location Tasks: A Human-Centric and Process-Oriented Exploratory Study," *Journal of Software: Evolution and Process*, vol. 25, no. 11, pp. 1193–1224, 2013.

[15] 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*, vol. 33, no. 6, pp. 420–432, 2007.

[16] T. J. Biggerstaff, B. G. Mitbander, and D. Webster, "The Concept Assignment Problem in Program Understanding," in *ICSE*, 1993.

[17] J. Rubin and M. Chechik, "A survey of feature location techniques," in *Domain Engineering*, 2013.

[18] J. Krüger, T. Berger, and T. Leich, *Features and how to find them: a survey of manual feature location*. LLC/CRC Press, 2018.

[19] T. K. Landauer, P. W. Foltz, and D. Laham, "An introduction to latent semantic analysis," *Discourse processes*, vol. 25, no. 2-3, pp. 259–284, 1998.

[20] F. Pérez, J. Echeverría, R. Lapeña, and C. Cetina, "Comparing manual and automated feature location in conceptual models: A controlled experiment," *Information and Software Technology*, vol. 125, p. 106337, 2020.

[21] L. R. Biggers, C. Bocovich, R. Capshaw, B. P. Eddy, L. H. Etzkorn, and N. A. Kraft, "Configuring latent dirichlet allocation based feature location," *Empirical Software Engineering*, vol. 19, pp. 465–500, 2014.

[22] The Authors, "Online Appendix," https://bitbucket.org/easelab/featracer/, 2023.

[23] H. Abukwaik, A. Burger, B. K. Andam, and T. Berger, "Semi-automated feature traceability with embedded annotations," in *ICSME*, 2018.

[24] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki, "A study of variability models and languages in the systems software domain," *IEEE Transactions on Software Engineering*, vol. 39, no. 12, pp. 1611–1640, 2013.

[25] D. Nešić, J. Krüger, S. Stănciulescu, and T. Berger, "Principles of feature modeling," in *ESEC/FSE*, 2019.

[26] A. Wąsowski and T. Berger, *Domain-Specific Languages: Effective Modeling, Automation, and Reuse*. Springer Nature, 2023, ch. Feature Modeling, pp. 437–457.

[27] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki, "Reverse engineering feature models," in *ICSE*, 2011.

[28] P. Franz, T. Berger, I. Fayaz, S. Nadi, and E. Groshev, "Configfix: Interactive configuration conflict resolution for the linux kernel," in *ICSE/SEIP*, 2021.

[29] J. Krüger, M. Mukelabai, W. Gu, H. Shen, R. Hebig, and T. Berger, "Where is my feature and what is it about? a case study on recovering feature facets," *Journal of Systems and Software*, vol. 152, pp. 239–253, 2019.

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

[31] T. Schwarz, W. Mahmood, and T. Berger, "A common notation and tool support for embedded feature annotations," in *SPLC*, 2020.

[32] J. Martinson, H. Jansson, M. Mukelabai, T. Berger, A. Bergel, and T. Ho-Quang, "Hans: Ide-based editing support for embedded feature annotations," in *SPLC*, 2021.

[33] S. Entekhabi, A. Solback, J.-P. Steghöfer, and T. Berger, "Visualization of feature locations with the tool featuredashboard," in *SPLC (2)*, 2019.

[34] B. Andam, A. Burger, T. Berger, and M. R. V. Chaudron, "Florida: Feature location dashboard for extracting and visualizing feature traces," in *VaMoS*, 2017.

[35] A. Bergel, R. Ghzouli, T. Berger, and M. R. V. Chaudron, "Featurevista: Interactive feature visualization," in *SPLC*, 2021.

[36] J. Krüger, J. Wiemann, W. Fenske, G. Saake, and T. Leich, "Do you remember this source code?" in *ICSE*, 2018.

[37] L. Passos, R. Queiroz, M. Mukelabai, T. Berger, S. Apel, K. Czarnecki, and J. Padilla, "A study of feature scattering in the linux kernel," *IEEE Transactions on Software Engineering*, 2018.

[38] T. Savage, M. Revelle, and D. Poshyvanyk, "Flat3: Feature location and textual tracing tool," in *ICSE*, 2010.

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

[40] A. Razzaq, A. Le Gear, C. Exton, and J. Buckley, "An empirical assessment of baseline feature location techniques," *Empirical Software Engineering*, vol. 25, no. 1, pp. 266–321, 2020.

[41] D. Poshyvanyk and A. Marcus, "Combining formal concept analysis with information retrieval for concept location in source code," in *ICPC*, 2007.

[42] 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*, vol. 33, no. 6, pp. 420–432, 2007.

[43] M. P. Robillard, "Automatic generation of suggestions for program investigation," in *ESEC/FSE*, 2005.

[44] M. Trifu, "Improving the dataflow-based concern identification approach," in *CSRM*, 2009.

[45] F. Asadi, M. Di Penta, G. Antoniol, and Y.-G. Guéhéneuc, "A heuristic-based approach to identify concepts in execution traces," in *CSMR*, 2010.

[46] M. Revelle, B. Dit, and D. Poshyvanyk, "Using data fusion and web mining to support feature location in software," in *ICPC*, 2010.

[47] C. S. Corley, K. Damevski, and N. A. Kraft, "Exploring the use of deep learning for feature location," in *ICSME*, 2015.

[48] A. C. Marcén, J. Font, Ó. Pastor, and C. Cetina, "Towards feature location in models through a learning to rank approach," in *SPLC*, 2017.

[49] A. C. Marcén, F. Pérez, Ó. Pastor, and C. Cetina, "Enhancing software model encoding for feature location approaches based on machine learning techniques," *Software and Systems Modeling*, vol. 21, no. 1, pp. 399–433, 2022.

[50] M. Ballarín, A. C. Marcén, V. Pelechano, and C. Cetina, "On the influence of model fragment properties on a machine learning-based approach for feature location," *Information and Software Technology*, vol. 129, p. 106430, 2021.

[51] R. Koschke and J. Quante, "On dynamic feature location," in *ASE*, 2005.

[52] 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.

[53] M. P. Robillard and G. C. Murphy, "Representing concerns in source code," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 1, Feb. 2007.

[54] S. Maro, J. Steghöfer, and M. Staron, "Software traceability in the automotive domain: Challenges and solutions," *J. Syst. Softw.*, vol. 141, pp. 85–110, 2018.

[55] M. Seiler and B. Paech, "Documenting and exploiting software feature knowledge through tags," in *SEKE*, 2019.

[56] T. Berger and J. Guo, "Towards system analysis with variability model metrics," in *VaMoS*, 2014.

[57] L. Passos, K. Czarnecki, S. Apel, A. Wąsowski, C. Kästner, and J. Guo, "Feature-Oriented Software Evolution," in *VaMoS*, 2013.

[58] M. Mukelabai, C. Derks, J. Krüger, and T. Berger, "To share, or not to share: Exploring test-case reusability in fork ecosystems," in *ASE*, 2023.

[59] M. Mukelabai, D. Nešić, S. Maro, T. Berger, and J.-P. Steghöfer, "Tackling combinatorial explosion: a study of industrial needs and practices for analyzing highly configurable systems," in *ASE*, 2018.

[60] E. Engström and P. Runeson, "Software product line testing—a systematic mapping study," *Information and Software Technology*, vol. 53, no. 1, pp. 2–13, 2011.

[61] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake, "A classification and survey of analysis strategies for software product lines," *ACM Computing Surveys*, vol. 47, no. 1, pp. 6:1–6:45, Jun. 2014.

[62] A. F. Blackwell and T. R. Green, "Investment of attention as an analytic approach to cognitive dimensions," in *PPIG-11*, 1999.

[63] T. R. G. Green and M. Petre, "Usability analysis of visual programming environments: a 'cognitive dimensions' framework," *Journal of Visual Languages & Computing*, vol. 7, no. 2, pp. 131–174, 1996.

[64] A. Blackwell and M. Burnett, "Applying attention investment to end-user programming," in *HCC*, 2002.

[65] C. Brown and C. Parnin, "Nudging students toward better software engineering behaviors," in *BotSE*, 2021.

[66] R. Balebako and L. Cranor, "Improving app privacy: Nudging app developers to protect user privacy," *IEEE Security & Privacy*, vol. 12, no. 4, pp. 55–58, 2014.

[67] M. Seiler and B. Paech, "Using tags to support feature management across issue tracking systems and version control systems," in *REFSQ*, 2017.

[68] K. Bąk, Z. Diskin, M. Antkiewicz, K. Czarnecki, and A. Wąsowski, "Clafer: unifying class and feature modeling," *Software & Systems Modeling*, vol. 15, no. 3, pp. 811–845, 2016.

[69] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich, "Featureide: An extensible framework for feature-oriented software development," *Science of Computer Programming*, vol. 79, pp. 70–85, 2014.

[70] D. Beuche, "Modeling and building software product lines with pure::variants," in *SPLC*, 2008.

[71] J. Sincero and W. Schröder-Preikschat, "The linux kernel configurator as a feature modeling tool." in *SPLC (2)*, 2008.

[72] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki, "The variability model of the linux kernel," in *VaMoS*, 2010.

[73] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wąsowski, "Evolution of the linux kernel variability model," in *SPLC*, 2010.

[74] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *ICSE*, 2013.

[75] S. Strüder, M. Mukelabai, D. Strüber, and T. Berger, "Feature-oriented defect prediction," in *SPLC*, 2020.

[76] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *ICSE*, 2013.

[77] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *ICSE*, 2008.

[78] R. Queiroz, L. Passos, M. T. Valente, C. Hunsen, S. Apel, and K. Czarnecki, "The shape of feature code: an analysis of twenty c-preprocessor-based systems," *Software & Systems Modeling*, vol. 16, no. 1, pp. 77–96, 2017.

[79] M. Antkiewicz, K. Bak, A. Murashkin, R. Olaechea, J. Hui, and K. Czarnecki, "Clafer tools for product line engineering," in *SPLC Workshops*, 2013.

[80] T. Fogdal, H. Scherrebeck, J. Kuusela, M. Becker, and B. Zhang, "Ten years of product line engineering at danfoss: lessons learned and way ahead," in *SPLC*, 2016.

[81] H. P. Jepsen, J. G. Dall, and D. Beuche, "Minimally invasive migration to software product lines," in *SPLC*, 2007.

[82] G. Tsoumakas, E. Spyromitros-Xioufis, J. Vilcek, and I. Vlahavas, "Mulan: A java library for multi-label learning," *Journal of Machine Learning Research*, vol. 12, pp. 2411–2414, 2011.

[83] N. SpolaôR, E. A. Cherman, M. C. Monard, and H. D. Lee, "A comparison of multi-label feature selection methods using the problem transformation approach," *Electronic Notes in Theoretical Computer Science*, vol. 292, pp. 135–151, 2013.

[84] H. Liu and H. Motoda, *Computational methods of feature selection*. CRC Press, 2007.

[85] N. Spolaôr, M. C. Monard, G. Tsoumakas, and H. D. Lee, "A systematic review of multi-label feature selection and a new method based on label construction," *Neurocomputing*, vol. 180, pp. 3–15, 2016.

[86] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *J. Mach. Learn. Res.*, vol. 3, no. null, p. 993–1022, mar 2003.

[87] J. Rubin and M. Chechik, "A Survey of Feature Location Techniques," in *Domain Engineering*. Springer, 2013, pp. 29–58.

[88] "Gensim LDA," https://radimrehurek.com/gensim/, 2022.

[89] Stephen W. Thomas, "Lucene-LDA repository," https://github.com/stepthom/lucene-lda, 2012.

[90] F. Charte, A. J. Rivera, M. J. del Jesus, and F. Herrera, "Addressing imbalance in multilabel classification: Measures and random resampling algorithms," *Neurocomputing*, vol. 163, pp. 3–16, 2015.

[91] F. Charte, A. Rivera, M. J. del Jesus, and F. Herrera, "Concurrence among imbalanced labels and its influence on multilabel resampling algorithms," in *HAIS*, 2014.

[92] F. Charte, A. J. Rivera, M. J. del Jesus, and F. Herrera, "Mlsmote: Approaching imbalanced multilabel learning through synthetic instance generation," *Knowledge-Based Systems*, vol. 89, pp. 385–397, 2015.

[93] C. Kästner, A. Dreiling, and K. Ostermann, "Variability mining with leadt," *Tec. Rep., Philipps Univ. Marburg*, 2011.

[94] F. W. Warr and M. P. Robillard, "Suade: Topology-based searches for software investigation," in *ICSE*, 2007.

[95] A. Pleuss, R. Rabiser, and G. Botterweck, "Visualization techniques for application in interactive product configuration," in *SPLC (2)*, 2011.

[96] C. Kästner, S. Trujillo, and S. Apel, "Visualizing software product line variabilities in source code." in *SPLC (2)*, 2008.

[97] L. Linsbauer, F. Schwaegerl, T. Berger, and P. Gruenbacher, "Concepts of variation control systems," *Journal of Systems and Software*, vol. 171, p. 110796, 2021.

[98] W. Mahmood, D. Strueber, T. Berger, R. Laemmel, and M. Mukelabai, "Seamless variability management with the virtual platform," in *ICSE*, 2021.

Mukelabai Mukelabai is a Lecturer in Computer Science at The University of Zambia. He received his PhD in Software Engineering at the University of Gothenburg, Sweden, in 2022. His research focuses on empirical software engineering and supporting feature-oriented quality assurance in low-maturity variant-rich systems.

Kevin Hermann is currently working towards a PhD degree at Ruhr University Bochum, Germany, where he is part of the Chair for Software Engineering, Faculty of Computer Science. He received his Masters degree in Applied Computer Science at the Ruhr University Bochum. His research focuses on the traceability of security features in software systems, and empirical software engineering.

Thorsten Berger is a Professor in Computer Science at Ruhr University Bochum in Germany. After receiving the PhD degree from the University of Leipzig in Germany in 2013, he was a Postdoctoral Fellow at the University of Waterloo in Canada and the IT University of Copenhagen in Denmark, and then an Associate Professor jointly at Chalmers University of Technology and the University of Gothenburg in Sweden. He received competitive grants from the Swedish Research Council, the Wallenberg Autonomous Systems Program, Vinnova Sweden (EU ITEA), and the European Union. He is a fellow of the Wallenberg Academy—one of the highest recognitions for researchers in Sweden. He received two *best-paper* and two *most-influential-paper* awards. His service was recognized with distinguished reviewer awards at the tier-one conferences ASE 2018 and ICSE 2020, and at SPLC 2022. His research focuses on model-driven software engineering, program analysis, empirical software engineering, and AI engineering.

Jan-Philipp Steghöfer is a Senior Researcher at XITASO Gmbh IT & Software Solutions in Germany and a former Associate Professor of Software Engineering at Chalmers University of Technology and the University of Gothenburg. His research interests include software and systems traceability, agile development of safe and secure systems, model-driven development, agile requirements engineering, and the engineering of AI applications with a focus on the healthcare domain.