RefactorScore: Evaluating Refactor Prone Code

Kevin Jesse, Christoph Kuhmuench, and Anand Sawant

***Abstract*—We propose REFACTORSCORE, an automatic evaluation metric for code. REFACTORSCORE computes the number of refactor prone locations on each token in a candidate file and maps the occurrences into a quantile to produce a score. REFACTORSCORE is evaluated across 61,735 commits and uses a model called REFACTORBERT trained to predict refactors on 1,111,246 commits. Finally, we validate REFACTORSCORE on a set of industry leading projects providing each with a REFACTORSCORE. We calibrate REFACTORSCORE's detection of low quality code with human developers through a human subject study. REFACTORBERT, the model driving the scoring mechanism, is capable of predicting defects and refactors predicted by REFDIFF 2.0. To our knowledge, our approach, coupled with the use of large scale data for training and validated with human developers, is the first code quality scoring metric of its kind.**

***Index Terms*—Refactor, automatic evaluation, machine learning, software repositories.**

I. INTRODUCTION

CODE refactoring is the process of improving the internal structure of the code while preserving the external behavior [1]. Refactoring practices are widely believed to improve code quality [2], programmer productivity [3], and is widely ingrained in development team workflows [4], [5], [6], [7]. The refactoring process is often left up to the developer, who has to assess the impact of a particular refactoring effort on their system. Additionally, such an effort often diverts developers' attention from more immediate timelines such as new customer features. Thus, developers are understandably resistant to refactoring efforts especially when the benefits are unclear. However, code quality does impact the long term success of a product and ideally should be addressed before poor quality is ingrained in the product.

Given the tradeoff of refactoring efforts, *viz.* increase time/engineering vs. decreasing internal code complexity, research such as RIPE [8] show developers the impact of a refactoring on code quality metrics prior to making such changes. The correlation of code quality metrics to needed refactorings is not universally generalizable per Nagappan et al. [9], [10] and RIPE finds correlations with a precision of 38%.

Recent works in Java, REFACTORMINER 2.0 [11] and REFDIFF 1.0 [12], and multi-language REFDIFF 2.0 [13] operate on the commit level to detect up to 40 refactoring patterns (REFACTORMINER) and in theory could prevent commits from entering the production branch; however there are three prerequisites to effectively match these patterns: 1) the existing code must perfectly match an established pattern 2) two revisions are required, e.g., a commit, *and* commit history, and 3) a Git-based version control system. We found, through randomly sampling of "refactoring commits" that they rarely, if ever, only perform a specific type of refactoring and are tangled with bug fixing and feature development. The aforementioned tools require two versions of the same code which may not be available. Finally, these tools are incompatible with C++ projects, Java for REFACTORMINER and Java/JavaScript/C for REFDIFF 2.0, and must use Git version tools. In our experience, the move to Git for some products is difficult, but the refactoring needs remain.

We also know that refactoring extends beyond the scope of specific refactoring patterns and is often entangled with code improvement reasons. We discern that an effective refactor detector must detect a mixture of distributions across a variety of topics: code smells, defects, and discrete refactoring patterns. Neural networks are universal approximators capable of learning complex relationships such as our general understanding of a refactoring, regardless of the convoluted nature of a tangled commit.

Jointly, with the widespread adoption of online software repositories like GitHub and GitLab, such platforms present an opportunity to learn the breadth of developers "refactoring" efforts as entangled as they may be.

With the capacity of fitting virtually any distribution, machine learning techniques are a modern focus for researchers. Researchers have experimented with several machine learning based methods to recommend refactors. Search based methods by Mariani et al. [14] and O'Keeffe [15], pattern mining by Bavota et al. [16] and various statistical based approaches in Aniche et al. [17]. In this article, we explore the effectiveness of large pretrained models and an extensive code quality dataset centered around refactoring operations found *in the wild*; that is with very little preprocessing effort, filtering of specific patterns, and disentanglement of coding intents. By learning from code that changes in refactoring commits, our model is better suited for detecting code quality errors that do not fit either

Manuscript received 8 December 2022; revised 20 September 2023; accepted 10 October 2023. Date of publication 16 October 2023; date of current version 14 November 2023. The work of Kevin Jesse's thesis research with UC Davis was supported by NSF CCF (SHF) under Grant 2107592. This work was done while interning at Siemens. Recommended for acceptance by N. Nagappan. (*Corresponding author: Kevin Jesse*.)

Kevin Jesse is with the Department of Computer Science, University of California Davis, Davis, CA 95616 USA (e-mail: krjesse@ucdavis.edu).

Christoph Kuhmuench is with Siemens Corporation, Princeton, NJ 08540 USA (e-mail: christoph.kuhmuench@siemens.com).

Anand Sawant is with Endor Labs, Palo Alto, CA 94301 USA (e-mail: anand@endor.ai).

Digital Object Identifier 10.1109/TSE.2023.3324613

the specific patterns or set of code features defined in existing works: REFACTORMINER [11], REFDIFF 1.0 [12]/2.0 [13], and Aniche et al. [17].

In this article, the formulation of the model's refactoring guess is similar to that of previous works. A model's guess of a refactor is a binary prediction where we can easily compare with existing methodologies. This work also mines commits, examines commit diffs, and makes predictions from versions of code. The large difference is that this work *only* requires the poor quality snippet, as predictions are conditioned on poor quality code; this differs from REFACTORMINER [11], REFDIFF 1.0 [12], REFDIFF 2.0 [13], REF-FINDER [18], REFACTORINGCRAWLER [19], and JDE-VAN [20]. Thus, our proposed model can be ran on current versions of code, without pending changes, and can determine whether code spans require a refactor. Further, this model is capable of aggregating such decisions to score entire functions, files, and projects. Within the subfield of existing works, a refactor score has yet to be proposed, especially in a manner that leverages large data and needs no adherence to specific refactor patterns.

A large focus of refactor research is based on using heuristics such as logic predicates and metrics; in this family are REFACTORMINER, REFACTORCRAWLER, JDEVAN (through UMLDIFF [21]), and REF-FINDER. Fundamentally, these methods are restricted by the same heuristics, but boast large precision and recall across *select* refactoring patterns. REFDIFF differs from the aforementioned works by employing global statistics in the form of term frequency inverse document frequency (TF-IDF). The TF-IDF weighting scheme assigns more weight to code tokens that are discriminate in the project. A limitation to generalization is the availability of overlapping tokens; this is exasperated by code's exploding vocabulary [22].

On the contrary, this article presents a model that has virtually no input constraints. Large pretrained language models (PLMs) [23], [24], [25], [26], [27] use tokenizers to break input tokens into subtokens [28], [29] which means very few words are out of vocabulary. PLMs benefit from large scale pretraining on swaths of raw data and can generalize efficiently to other contexts, not influenced by code deviations like domain specific language, identifier naming conventions, and semantic alternatives of similar functionality. The model presented in this work, REFACTORBERT, is an extension of CodeBERT, and is trained to predict poor code quality over 1.1 million open source code commits; this is approximately 55 million code refactoring token sequences. As we report in later sections, REFACTORBERT is capable of predicting refactor spans across any span of C and C++ code, grade the code quality, and predict refactors better than REFDIFF 2.0. We validate REFACTORBERT with human experts in a human subject study and confirm that the predictions by REFACTORBERT are indeed low quality code and worth refactoring. In the human subject study on code quality, we also found biases in developers when asking them to evaluate code they are familiar with, thus, emphasizing the need for automatic evaluation. This article also determines that models trained on commits *in the wild*, are capable of learning defect prediction; REFACTORBERT scores competitively on the CodeXGLUE [30] defect prediction benchmark despite not being formally trained to predict defects. Then, with REFACTORBERT, we score popular libraries providing them with a REFACTORSCORE. Finally, we examine the libraries' REFACTORSCORE over a period of 10 years and discover that code quality is degrading over time. In summary our contributions are as follows,

1) REFACTORBERT: a model capable of predicting refactors, specifically code spans of poor quality.
2) REFACTORSCORE: a score summarizing the degree of code quality.
3) A large-scale heterogenous dataset of C/C++ refactoring commits totaling over 1.1 million commits and 55 million token sequences.
4) A human subject study confirming our intuitions about entangled commits, code quality, and the effectiveness of REFACTORSCORE and REFACTORBERT

All datasets and model weights are available on Huggingface[1,2].

II. RELATED WORK

In this section, we examine domains related to predicting and prioritizing code quality through refactoring.

A. Background

Empirical studies on code quality often rely on refactoring commits to identify interesting patterns. Peruma et al. [31] explored the renaming of identifiers motivated by refactors and their commit messages. The renamings of identifiers ultimately distill functionality and domain knowledge resulting in better code.

Another study, Brown et al. [32] found long term code quality is often traded for short-term gain in the form of technical debt. This technical debt can be effectively managed in order to maintain software quality. Agile practices of refactoring are just one of the many tools to reduce technical debt and improve code quality. Fontana et al. [33] investigated the impact of refactoring on code smells and found improvement across code metrics that measure code quality and technical debt. Code smells often result in refactoring opportunities [17], [34], so it is a shared belief that technical debt such as smelly code and even defects, can be jointly addressed by refactor operations.

Mens and Tourwé [35] surveyed refactoring activities, supporting techniques, and tool support with an emphasis on requirement, design, and code refactors. This review did not comprehensively discuss specific refactoring opportunities and patterns, and since, subsequent literature reviews have been performed citing increased use of code smells and anti-patterns. Baqais and Alshayeb [36] recently surveyed automatic refactoring with most works focusing on code refactoring and only a few focused on model refactoring. Search-based refactoring (SBR) is gaining more popularity since developers can use it in a quick and efficient manner.

Mariani and Vegilio [14] discussed SBR and the increase of multi/many-objective algorithms in REF-FINDER [18] and

[1]https://huggingface.co/datasets/kevinjesse/ManyRefactors4C
[2]https://huggingface.co/kevinjesse/RefactorBERT

CODE-IMP [37]. REF-FINDER is an Eclipse plugin that can show which refactoring patterns were used from the 96 refactoring patterns of the original Fowler [1] catalog. CODE-IMP is another automated SBR tool that combines a set of refactoring patterns, metrics, and search algorithms; the developer must choose the patterns and metrics to be used in the fitness function and the search technique. Mohan and Greer [38] detailed how search based approaches have evolved over time and confirmed that evolutionary algorithms are the most commonly used search technique.

Machine learning approaches are often used to detect code smells [39], [40], [41], [42], [43], [44] and refactoring opportunities [11], [12], [13], [17], [19]. Azeem et al. [34] is a systematic literature review of machine learning algorithms for code smell prediction. Azeem et al. targeted specific code smells with various ML approaches and analyzed their performance. Azeem et al. evaluates machine learning models like decision trees, SVM, and random forest across specific refactoring patterns and discovers significant room for improvement in the context of code smell detection. This is in line with Di Nucci et al. [45], who reported that code smell machine learning models fail to perform at previously reported levels. We avoid previous pitfalls discussed in these works by carefully evaluating REFACTORSCORE on human curated evaluation sets from other leading works. We use the appropriate measures such as F1 score and validate our model with a human subject study on code quality.

B. Refactoring Models

Demeyer et al. [46] devised a technique for detecting refactors between two software versions. Demeyer et al. presented a set of heuristics for detecting refactors by applying light object-oriented metrics to successive versions of software. The heuristics aimed to discover characteristics of software systems that are symptomatic of design shifts. The authors examine refactor operations like split into super/sub class, merge with super/sub class, factor out common functionality, and moving functionality to other classes. The heuristics by Demeyer et al. are validated on three software systems and useful in providing an unbiased view of systems, interactions, and recognizing implementation skew from original design.

Antoniol et al. [47] realized an automated approach to identify class evolution discontinuities by using vector space information retrieval. Class evolution discontinuities, like the splitting of class functionality, represents a family of refactoring operations. The approach was performed over 40 releases of Java name server and the authors found that almost all refactoring operations were found and performed. Weißgerber and Diehl [48] note the approach by Antoniol has degraded performance when many changes have been performed within a class.

Weißgerber and Diehl [48] proposed a refactor detector by finding changes to method signatures. If methods are changed according to structural and local refactoring patterns, then they are classified as a refactor. The approach details conditions for both structural refactors and local refactors. Refactors must match the sets of conditions which cannot account for all possible ways developers apply refactorings.

Dig et al. [49] created REFACTORCRAWLER which leverages fast syntactic analysis to detect refactor candidates and then performs expensive semantic analysis to refine the predictions. The analysis is based on Shingles encoding, a technique borrowed from Information Retrieval. Dig et al. manually investigated the release notes in three projects to compute recall and examined source code to compute precision.

Xing and Stroulia [20] introduced JDEVAN based on design-level changes from their previous work UMLDIFF [21]. JDEVAN implements a set of queries for detecting instances of well understood design change patterns including: refactors that restructure containment and inheritance hierarchies, move features between objects, and refactor class internals. They evaluated the recall of JDEVAN on two software projects and found all known refactor operations.

Kim et al. introduced REF-FINDER [18], a refactoring tool based on logic query templates [51] and is capable of detecting most refactor types from Fowler's catalog [1]. REF-FINDER takes two program versions either from workspace snapshots or subversion repositories and extracts logic facts about the program's syntactic structure. REF-FINDER then invokes logic queries to identify program differences that match refactor types. Tsantalis et al. [11] accurately notes that some refactoring rules use a special predicate that checks if word similarity between two candidate methods is above a threshold; REF-FINDER's query templates based on, Prete [51], are validated with a low threshold and then evaluated with a different, higher threshold. In theory, this impacts generalizing to new domains where thresholds may have to be adjusted based on sampling.

Tsantalis et al. formulated RMINER/REFACTORMINER 1.0 [50], and REFACTORMINER 2.0 [11], with the latter being the most comprehensive. REFACTORMINER 2.0 is capable of identifying 40 high-level refactoring types. Similar to Xing and Stroulia [21], their algorithm performs differencing by matching statements with the AST representation. The matching of statements is bottom-up starting with leaf and proceeds to larger composite statements. The matching of AST nodes does not require any word level similarity measures to match statements and does not use similarity thresholds. Of course the algorithm does not benefit from the similarity matching techniques in other works, but without thresholds, their algorithm has better explainability and correctness as the algorithm only matches on defined conditions and conjunctions of these conditions. REFACTORMINER 2.0 matches more statements and replacement types leading to a higher precision and recall. Finally, the authors contribute an extended set of true refactoring instances and a comparison with REFDIFF. To our knowledge, this approach has the highest precision and recall on the Java dataset.

Silva and Valente [12] created REFDIFF 1.0 capable of detecting refactor operations in version histories. REFDIFF 1.0 was originally created for Java code and Git repositories. The authors contributed 448 known refactoring operations applied to seven Java systems, which served as a benchmark for other refactoring detection approaches. REFDIFF 2.0 [13], also by

TABLE I
REFACTORING APPROACHES

Tool	Input	Features	Detection	Evaluation	Language
Demeyer *et al.* [46]	Classes, methods, & attributes	Change metrics	Custom heuristics	Tools	Smalltalk
Antoniol *et al.* [47]	Identifiers	TF-IDF	Case study	dnsjava releases	Java
Weißgerber & Diehl [48]	CVS commits	Classes, methods, & fields	Signature changes	Documented refactors & sampling	Java
Dig *et al.* [49]	Code versions	Tokens	Reference similarity	Release notes	Java
Xing & Stroulia [20]	Project versions	Directed graph	UMLDiff & queries	Change documents	Java
Kim *et al.* [18]	Program versions	Syntax tree	Template logic query	jEdit Tool & examples	Java
Silva & Valente [12]	Git commits	TF-IDF	Custom rules	Seeded refactors	Java
Tsantalis *et al.* [50]	Git commits	AST features	Custom rules	Validated dataset	Java
Tsantalis *et al.* [11]	Git commits	AST features	Custom rules	Expanded dataset	Java
Silva & Valente [13]	Git Commits	TF-IDF	Custom rules	Validated dataset	Java, C++ & JavaScript
RefactorBERT	Tokens	Pretrained embeddings	Commit messages	Validated dataset [13]	C, C++

Silva and Valente, built upon the previous version and introduced multi-language refactoring with code structure trees. The language-agnostic design improved REFDIFFs precision and recall to be on par with REFACTORMINER 2.0 and was extended to two other programming languages: JavaScript and C. Our model, REFACTORSCORE, is trained for C/C++ for Siemens internal code, and thus, REFDIFF 2.0 is our primary comparison to state-of-the-art.

The aforementioned approaches are summarized in Table I.

C. Commercial Software Quality Tools

CodeQL. GitHub's CodeQL[3] is an analysis engine that automates security checks, bug detection, and other errors which are modeled as queries to a feature database. The underlying QL language introduced by De Moor [52] is the primary technology in which code is searched against known defects. CodeQL uses variant analysis by seeding known security vulnerabilities and attempts to find similar problems in target code. CodeQL focuses primarily on security threats and is not comparable to existing refactor detection works.

SonarQube. A popular platform SonarQube is a service or standalone instance that scans for bugs, vulnerabilities, code smells, coverage, and code duplication. Its comes with a graphical user interface (GUI) for interpreting the results. The tool covers great breadth of software quality including reliability, security, and maintainability. The inner workings of SonarQube are hidden due to the proprietary nature, however several empirical works have been conducted to evaluate its effectiveness. Lenarduzzi et al. [53] discovered that only few rules have low fault-proneness and many violations considered as bugs were generally not fault-prone. Consequently, the authors concluded that the fault-prediction power of the model is extremely low. The study by Lenarduzzi was motivated by real companies complaining that many rules classifying bugs are not actually faults. Marcilio et al. [54] examined static analysis violations predicted by SonarQube and found that just 13% of issues are solved in the systems, conjecturing, that just a subset of checkers actually reveal real design and coding flaws. With regard to refactor detection and REFACTORSCORE, SonarQube does not explicitly consider refactoring opportunities. With the proprietary nature of the database used in SonarQube, we are unable to control the possible leakage of the evaluation set and the query database. For these reasons, an apples to apples comparison is not possible.

In the next section, we discuss the intersection of code smells, defects, and refactoring in the context of the *refactor oracle problem* (Tsantalis et al. [11]).

D. Refactor Oracles

Refactor mining often leads to oracles that are incomplete, biased, or incorrect for a variety of reasons. Some reasons for incorrect refactor detection include flawed mining heuristics, tangled commits of refactoring and code improvement, or even plainly mislabeled commits messages. Dig et al. [49] formulated a refactor oracle with release notes. Moreno et al. [55] thoroughly inspected 990 release notes and found only 21% are refactoring operations; of these 990 release notes: 888 fixed bugs, 717 indicated new or modified features, 821 specified new or modified code components, and 206 notes mentioned refactoring operations. The reported number of release notes mentioning multiple content types shows the tangled nature of refactoring commits with bug fixing and feature development. This is exacerbated by refactoring release notes citing specific code components and public facing APIs that are affected rather than describing the operation as a refactor.

Ratzinger et al. [56] explored a predefined set of terms in commit messages to classify the commits as a refactor. Weißgerber et al. [48] created a refactoring oracle by inspecting commit messages for references to refactoring operations. Murphy-Hill [4] investigated that such commit message assumptions are often false as refactoring is an unconscious activity. Murphy-Hill also suggests that perhaps the programmer is considering the refactoring subordinate to another attitude, *e.g.*, patching, including new features, or upgrading APIs to name a few.

Synthetic oracles such as the one in REFDIFF [12] are useful in computing the precision and recall of automatic refactor tools because all refactorings are seeded. Unfortunately, synthetic refactor operations are not representative of real refactor operations and the synthetic operations are not entangled with other maintenance operations such as bug fixes and new features. To our benefit, real projects like Linux and PHP are included in the 20 project C evaluation set for REFDIFF 2.0, which positions the evaluation and comparison in a real world setting.

Automatic refactor detection tools have been used to craft refactoring oracles. Tools such as Aniche et al. [17] and Kim et al. [18] are configured by setting universal threshold values. The real world efficacy of Kim et al. (REF-FINDER) is challenged by Soares et al. [57] and Kádár et al. [58] citing only 24% recall to the reported 96%. Configuring thresholds not only leads to reduced performance but also poses a risk in mining data because these tools will likely generalize incorrectly when not

[3]https://codeql.github.com

properly tuned, propagating their statistical bias into datasets. Traditionally, finding effective universal threshold values can be challenging.

Popular neural networks boast millions, even billions, of parameters and are optimized by a loss function set with an arbitrary probabilistic threshold. For large neural networks, thresholding is not a performance oriented adjustment, but rather, a tool to weight costs between precision and recall. Due to the pre-training, generalization with a particular probability threshold is generally not a concern in these networks.

In the following sections, we formulate refactor prediction as a binary classification task and present a neural model REFACTORBERT trained on over 55 million examples of code sourced from refactoring commits.

III. METHODOLOGY

Our objective is to evaluate arbitrary code and determine whether it suffers from degraded code quality. Per the related work, previous studies primarily focus on specific code quality symptoms and metrics. We first ask if machine learning based approaches, in the realm of large language modeling, can distill refactoring patterns without the patterns being explicitly defined. To that effect we ask research question 1,

RQ1: *How effective are refactor prediction models trained on a mixture of refactoring intents?*

Once we establish the potential efficacy of our approach, we compare it to other refactor detection approaches to understand whether large BERT based models outperform traditional approaches, and what patterns they are capable of detecting across verified refactors. To this end, we ask the second research question,

RQ2: *How does REFACTORBERT compare with state-of-the-art refactor prediction model REFDIFF 2.0?*

Our training paradigm teaches our model to learn refactors as they are labeled by developers. This likely entails capturing developers fixing bugs and even refactoring mistakes. We test the models ability to detect bugs on a popular code intelligence benchmark on defect detection. We answer the following research question,

RQ3: *Do refactor aware models capture defects?*

After an evaluation of the model's ability to detect developer labeled refactorings, we wish to use REFACTORBERT's sequence prediction to score code. We wonder how our model scores projects over time and where top projects are today in their refactoring efforts. Our fourth research question asks,

RQ4: *How does code quality change with time? Does refactor size, a proxy for technical debt, increase over the project's lifetime?*

Most importantly, the grading system must align with developers' perception of bad code. Consequently, we perform a human subject study with 11 engineers, some even working on the code base. Our fifth research question is,

RQ5: *Do developers agree with REFACTORBERT's interpretation of code quality?*

Finally, we are curious how experienced developers react to code they have previously worked with. Our final RQ is,

RQ6: *Are developers resistant to changing code they are familiar with?*

To address these research questions, we design a model REFACTORBERT that predicts spans of low quality code. In the following sections, we discuss the task of refactor prediction and experiment with various modeling choices. Subsequently, we explain how the dataset was collected and the heuristics used to collect the refactors. The ensuing section evaluates our proposed models against other designs, state-of-the-art detectors, and applies the model in a broad range of settings. Finally, we conclude with a human-subject study to confirm effectiveness and discuss the implications of REFACTORSCORE for business level objectives.

A. Problem Definition

We first define our binary classification problem we call REFACTORPREDICT or RP for short. Then we define more granular objective called REFACTORSPANPREDICT or RSP. The problem is defined with respect to a program's source code or token sequences. Consider a program P composed of token sequences where $P = \langle t_0, ..., t_n \rangle$, where the tokens t are part of a vocabulary $\mathcal{V}$ and n is the number of tokens in the program. A token is a basic unit of text or code, in our case, source code words. The vocabulary consists of (sub)words and is generated from the training data in a manner that maximizes the representational power; a form of compression based on frequency and probabilities. In this work we use the CodeBERT tokenizer [23], which is trained on a combination of natural language and code. A tokenizer is a tool that delimits strings to tokens in an optimal fashion for the model; this can include breaking tokens into subtokens. Each subtoken has an aligned location, L, within the program's full token sequence where $L = \langle 0, ..., n \rangle$. In each task, RP and RSP, a binary mask $M \in \{0, 1\}^n$, represents each (sub)token in a sequence.

Given a program sequence P, the goal in the REFACTORPREDICT task is to predict if the program requires a refactor or does not. In Fig. 2(a), the sequence of tokens are passed into an embedding layer, which is a vector space R^d, designed to embed the input in a learned vector space over the training procedure. Next, the embedding layer is passed into a neural network, often a transformer network, where the network can "attend" to specific features. Finally, the networks *high level* representation is pooled or reduced into a single vector. This final vector is the sequence level vector where the network stores relevant sequence level information. A projection, or decision layer, produces a single classification value 0 or 1, which is our refactor prediction and is compared to our ground truth value from the commit.

Now, observe the REFACTORSPANPREDICT task in Fig. 2(b). The underlying architecture is the same as the RP task, but now the model must make a prediction for each token. The model

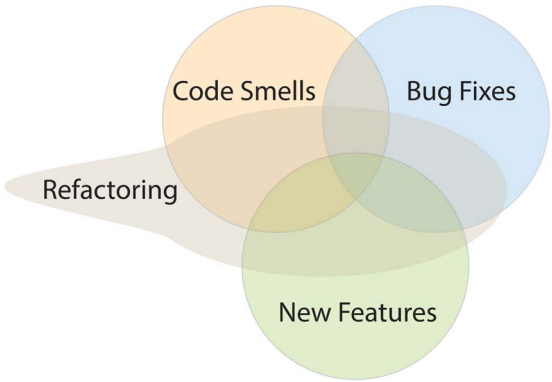


Fig. 1. Refactoring commits are often entangled with maintenance operations targeting code smells, bugs, and new features.

does not pool or reduce the sequence of hidden states, but rather makes a prediction of 0 or 1 for each token. Both models have the same underlying structure, but the calculation of the loss must be done on each and every token. On this task, RSP, the model gains the benefit of partial credit when some of the token predictions in the sequence are correct. We postulate that the model will have a greater attention to detail, as the model can determine *which* elements in what contexts contribute to the loss, as opposed to making a single general prediction across a large span as in RP.

The labels in Fig. 2(a) and 2(b) come from a refactoring commit's diff. Fig. 3 is a refactoring commit diff and illustrates the process of gathering the ground truth labels for RSP. The commits from which the labels originate from are commits that specify refactoring in the commit message. This is a common mining technique [17], [48], [56], albeit the sampling is noisy from the entanglement problem, *e.g.*, a commit that performs bug fixing and refactoring, *ref.* Fig. 1. The commit diff shows the removed items in red and added items in green. The red highlight indicates code that was removed, or for our purposes, considered incorrect with respect to a refactor. All surrounding code is considered correct if the code is not altered. The span of removed code is "highlighted" by crafting a mask of 1s and 0s around the changed and unchanged portions. During training, the model is fed both refactored sequences and non-refactored sequences with the tasks of predicting the sequence label (RP) or the label for each token (RSP), that corresponds to whether the sequence or token was removed in a refactor. A major benefit of framing the problem in this setting, is that only the removed portions from pre-refactored code is required for prediction; unlike REFACTORMINER, REFDIFF, REF-FINDER, REFACTORINGCRAWLER, and JDE-VAN.

B. Modeling RP and RSP

With the RP and RSP tasks defined, we now define the modeling of these tasks with respect to various neural network architectures. Some of these form our baselines.

Given the subtoken sequence $S = \langle t_1, ..., t_n \rangle$, a portion of program P, the model embeds each subtoken t_i using the trainable weight matrix ϕ where each subtoken maps to an embedding. For the set of all subtokens, $\mathcal{T}$, there is a vector or embedding e, i.e., $\phi \rightarrow \mathbb{R}^d$ which results in a matrix of embeddings for each sequence, i.e.,

$$[e_1, ..., e_n] = [\phi(t_1), ..., \phi(t_n)].$$

The embedding matrix is passed to various model architectures. We evaluate Long Short-Term memory (LSTM), Gated Recurrent Units (GRU), Transformers, and Large Pretrained Transformers (CodeBERT). Included statistic baselines Naïve Bayes and Logistic regression do not use embedding tables but rather corpus statistics with TF-IDF. We will continue with RefactorBERT in place of each of the aforementioned neural models as the prediction process is similar between the models.

Following the embedding of the input, we pass the embedded input to the model,

$$[h_{cls}, h_1, ..., h_n] = \text{RefactorBERT}([e_{cls}, e_1, ..., e_n])$$

where $[h_1, ..., h_n]$ are the model hidden states for each token, and $h_i \in \mathbb{R}^d$ where $i \in [1, ..., n]$ and h_{cls} is the "pooled" representation, in other words, a special token representing the entire sequence.

In the Refactor Prediction (RP), the pooled token $h_{cls} \in \mathbb{R}^d$ is projected with a trainable weight matrix $\mathbf{W_{RP}}$ with an additive bias $\mathbf{B_{RP}}$, hence,

$$\text{P}_\text{S} = \sigma(\mathbf{W_{RP}} h_{cls} + \mathbf{B_{RP}}), \qquad \text{P}_\text{S} \in (0, 1) \tag{1}$$

where $\mathbf{W_{RP}} \in \mathbb{R}^{1 \times d}$. $\sigma(x)$ is the classic sigmoid activation function bounding a logit to a value between (0,1), *viz.* a probability, with the function $\sigma(x) = \frac{1}{1+e^{-x}}$. The model will predict a refactor, value 1, when P exceeds some probability threshold τ. Such that

$$Pred(S) = \begin{cases} 1, & \text{if P} \geq \tau \\ 0, & \text{otherwise} \end{cases}$$

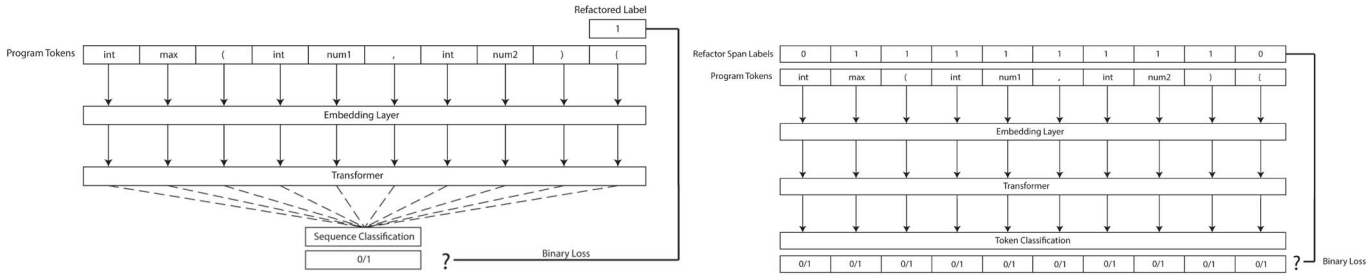
where we select $\tau = .5$ or 50%.

The Refactor Span Prediction (RSP) task requires a similarly crafted prediction but now at the token level rather than the sequence level. We can augment our neural architecture to project, or make a decision, at each token's hidden state to a binary prediction.

In RSP, the hidden state vectors, $[h_1, ..., h_n]$ or in matrix notation, $\mathbf{H} \in \mathbb{R}^{d \times n}$, are projected with a trainable weight matrix $\mathbf{W_{RSP}} \in \mathbb{R}^{1 \times d}$ with an additive bias $\mathbf{B_{RSP}}$, hence,

$$\text{P}_{\text{t}_n} = \sigma(\mathbf{W_{RSP}} \mathbf{H} + \mathbf{B_{RSP}}), \quad \forall \ > \text{t}_i \in S, \text{P}_{\text{t}_i} \in (0, 1) \tag{2}$$

where t_i is the aforementioned subtoken in sequence S and a probability P for each token $t_1, ..., t_n$. This gives a sequence of zeros and ones which are compared with our ground truth labels from the commit. Throughout many examples, the model can determine which subsequences are correlated with refactors and predict partial or complete sequences of refactors. This sequence based objective is the underpinning of our REFACTORSCORE discussed in a later section. Next we explain how REFACTORBERT is trained.



(a) Refactor Prediction (RP) task. (b) Refactor Span Prediction (RSP) task.

Fig. 2. Illustration of model architectures for sequence level refactor prediction and span level refactor prediction.

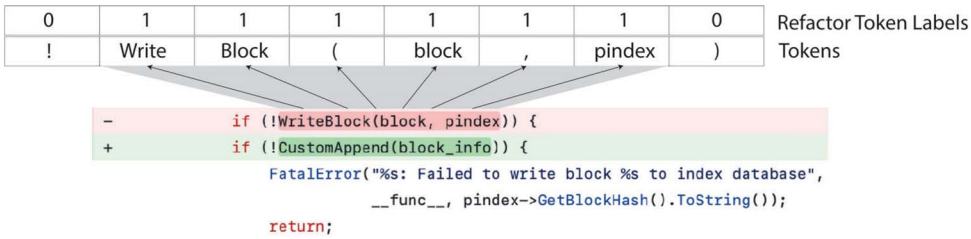


Fig. 3. How a refactoring commit is converted from diff to dataset.

C. Training the Model

We train REFACTORBERT on both RP and RSP tasks in addition to the baselines LSTM, GRU, and Transformer models. The RP and RSP training data is easily accessible with the Huggingface api hosted on Huggingface. The dataset boasts over 55 million sequences. The dataset contains parsed files from the refactor commit diff. Spans of contiguous lines are used for both positive examples of a refactor and surrounding spans are considered for negative examples. The dataset consists of more negative examples where positive examples only account for 22% of the entire dataset. Sampling down the negative examples is possible with minimal effort with the Huggingface dataset API; classes can be sampled at different probabilities. Our models are trained on all available data including negative examples.

Finally, the RP task is a reduction of the RSP task because the RP label is dependent on the presence of a refactor span in the sequence fed to the model. In all models the sequence is fixed to a length of 512 subtokens, tokenized with a CodeBERT [23] tokenizer.

The evaluation of a model's prediction is as follows. Let i denote the subtoken index of a program consisting of n subtokens in sequence S. For RP, the ground truth label is a sequence level label and for RSP the ground truth label is a token level label. Specifically the binary vectors Label_{RP} and Label_{RSP} are,

$$\text{Label}_{RP} = \begin{cases} 1, & \text{if } \exists\, t_i \in R,\ i \in [1, ..., n] \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

$$\text{Label}_{RSP}[i] = \begin{cases} 1, & \text{if } t_i \in R,\ i \in [1, ..., n], \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

where R is the set of refactored tokens. Label_{RP} is a binary valued vector corresponding to each sequence of code and Label_{RSP} is a 2D binary valued vector of corresponding to each token in the code per sequence.

To optimize the model, we train the model with a binary cross entropy loss.

$$L = \frac{1}{N}\sum_{i=1}^{N} l_i$$
$$l_i = -\left[y_i \cdot \log(P_i) + (1 - y_i) \cdot \log(1 - P_i))\right] \tag{5}$$

where N is the batch size and P is the probability from Equation 1 or Equation 2. For both tasks, the predictions are "flattened" to a single vector of predictions and labels. Finally each sequence (RP), or token location (RSP), loss is averaged for a final loss which is applied to the model with back-propagation. For the RSP model, we use both prediction tasks and compute a single loss

$$L = L_{RP} + L_{RSP}$$

where both losses are calculated from Equation 5 at sequence and token level.

Additionally, we find better performance by broadcasting the RP prediction to the RSP sequence. Namely, if the RP head indicates a refactor does not exist, it predicts 0, but the RSP head might have a few tokens it believes is a refactor. The RP value is applied to the sequence, changing the few tokens from 1 to 0, thus correcting the sequence level prediction. We found this model has the highest precision and recall. We call this the *joint* prediction model, where RP and RSP heads *jointly* impact predictions.

Evaluation of the joint model uses the same vectors of labels from Equation 4, Label_{RSP}, but computes the predictions as follows,

$$Pred_{Joint} = \langle Pred_{RP}(S) \otimes 1_n \rangle \odot Pred_{RSP}(S) \tag{6}$$

where $Pred_{RP}$ is from Equation 1 and $Pred_{RSP}$ is from Equation 2. The value from $Pred_{RP}$ is tiled into a vector and broadcast to the span prediction in $Pred_{RSP}$. We found this joint model has the highest prediction accuracy.

In the following section, we briefly discuss some modeling architectures we found less suitable for the task.

D. Less Effective Model Variants

The hidden vectors, H, can be manipulated to predict elements of the refactor prediction task. For example, each token index i in $\langle t_1 \dots t_n \rangle$ can be represented as a category or a value. If represented as a category, the beginning and end indices of a refactor can be predicted by optimizing a categorical loss. In this setup, there is no notion of "closeness" if the index was off by a little and partially correct predictions of token spans result in no credit. This setup did not result in learning refactors. This approach also does not work for multiple disjoint spans of refactors.

A more logical setup, is to predict the index as a real number. The regression output can range drastically in value from 0 to 512. While there is a notion of closeness, *e.g.*, 499 is close to 500, we found this approach to be unsuitable for the task. The model did not converge well in a large heterogenous dataset. In this setup, experimental biases such as centering code for more context means the model learns some indices are more common than others. This approach also does not work for multiple disjoint spans of refactors.

Finally, we experimented with predicting a probability distribution across 512 indexes for the start and end locations of the span. This loss is focused on maximizing the probability or confidence at a location where the refactor began and ended. Intuitively, this setup should work better than the prior two, however we did not see substantial learning progress. Like the previous two, this approach does not work for multiple disjoint spans of refactors.

The RSP task is closely aligned to sequence tagging and it was the most effective. We believe this is due to the partial credit a partially correct subsequence can achieve. Additionally, properly labeled tokens that form multiple subsequences of 1s can be positively enforced by the loss, whereas predicting a singular span explicitly forces the model to assume a bias, that bias being that all refactors occur contiguously in the token sequence.

IV. Model Evaluation

In the following sections we discuss the dataset curation, the models under test, and answer the aforementioned research questions.

A. Models Under Test

Section IV-A describes the models under test in an increasing complexity.

Random Predictor. The random predictor achieves its score by picking the most frequent class.

Naïve Bayes. Naïve Bayes is a supervised learning algorithm based on Bayes' theorem with an assumption of conditional independence between features, *hence*, naïve. Practically, Naïve bayes is an effective classifier that works in many real world scenarios. For sake of comparison, Naïve Bayes is a fast, lightweight classifier. The features for our Naïve Bayes baseline are TF-IDF vectors.

Logistic Regression. A statistical model that models the probability of a refactor taking place by the log-odds which are a linear combination of the independent variables. The log-odds are converted to probability with the logistic function. The features for this classifier are TF-IDF vectors.

Long Short-Term Memory (LSTM). A recurrent neural network that has feedback connections often used for modeling sequences like code and text.

Gated Recurrent Units (GRU). Introduced in 2014 as an alternative to LSTM. The performance of the GRU is often similar to LSTM.

Transformer. Introduced in 2017 by Vaswani et al. [59], it is the bedrock of LLMs today. Transformers operate with self-attention which is largely compute efficient *viz.* with parallel processing because the attention matrices are decomposable.

RefactorBERT. A transformer LLM pretrained on code. The initial weights are from CodeBERT [23].

B. Dataset Curation

First we identify a list of top starred repositories in C (29,021) and CPP (30,346) using GraphQL on the GitHub GraphQL API. Then on each project, we use GraphQL to gather the commit hash of each commit filtering only commits that include the word "refactor". This is an inherently noisy technique according to Murphy-Hill [4], however, at scale we found this simple approach effective.

We assess the effectiveness by sampling 500 commits each from different projects and manually label the commits as yes (true refactor), no (false refactor), and unsure. We labeled the commit as a true refactor if significant portions of code changed to reduce code complexity, such as variable renamings or method splitting, without changing program behavior. We allow multiple intents such as feature development or bug fixing if the commit still refactored elements described in the commit message. Of these sampled commits, 352 were true refactors, 60 were not refactors, and the remaining were not conclusively determined yes or no. This results in 85.44% of commits being true refactors with $> 95\%$ confidence threshold. We publish our labeling results[4]. Through manual inspection, we found many false positives were commits fixing bugs from previous refactors which reinforced our intuition that a model trained from this data would be good at detecting bugs.

With over 1.1 million commits from this heuristic, we examine the commit diffs. The commit diffs are analysed with PyDriller [62] where commit attributes are gathered including the parsed diff for each file in the commit. Within the `mod_files` field, we can extract the lines parsed in `diff_parsed` and

[4]https://github.com/kevinjesse/RefactorBERT/blob/main/sampled_refactored_commits.csv

TABLE II
REFACTOR PREDICTION (RP) RESULTS

Model	P (%)	R (%)	F1 (%)	Acc (%)	p-val	CI (95%)
Random Predictor	22.13	100.00	36.24	22.13	—	[22.10, 22.16]
Naïve Bayes	23.97	65.38	35.08	46.44	< 0.0001	[46.41, 46.47]
Logistic Regression	29.38	76.84	42.50	53.99	< 0.0001	[53.96, 54.02]
LSTM [60]	27.14	69.15	38.98	52.08	< 0.0001	[52.05, 52.11]
GRU [61]	34.61	83.76	48.98	61.38	< 0.0001	[61.35, 61.41]
Transformer [59]	31.02	79.01	44.55	56.48	< 0.0001	[56.45, 56.51]
RefactorBERT	**56.68**	**89.23**	**69.32**	**82.53**	< 0.0001	[82.50, 82.55]

Evaluation at the sequence level. P-value and confidence interval are computed with a two-tailed binomial test.

gather the `source_code_before` which we use for positive and negative sampling as well as the `source_code` field to find the refactor changes. Finally when each file source code is parsed, groups of lines deleted are sampled and labeled as positive refactoring examples and areas unchanged in the same file are labeled as non refactored changes. Across all project, commits, and files, we are left with 55 million sequences. We split this dataset across *projects* into train, test, validation with a 90%-5%-5% split; this results in a split by *examples* of 42.5, 9.09, and 3.98 million sequences for train, test, validation. We cannot perform effective deduplication as we sample commits throughout a projects' lifetime and similar code contexts make removing near duplicates very challenging. To mitigate the impact of any data leakage, we evaluate across projects and only use the evaluation set to pick a best performing model. Then, we compare the model across verified refactorings from RefDiff 2.0. The dataset is available on Huggingface[5] with over 55 million sequences source from commits citing refactoring operations.

In the next section, we explore the effectiveness of various neural architectures and ablations in detecting poor quality code.

C. RefactorBERT Performance

We compare RefactorBERT and our comparable baseline models by binary classification metrics: precision, recall, F1, accuracy across the our refactor dataset. This dataset is curated from over 1 million commits and over 55 million sequences from refactoring commits. We split the dataset by project, 90% train, 5% test, and 5% validation. We evaluated all models across the same test set. The models are trained for equal number of examples, however the dataset is so large, that we found training over a fraction of the dataset to be sufficient.

The performance of the models by architecture can be found in Table II. We measure the models' performance at the sequence level predicting whether a span of tokens will require a refactor. The precision, also called positive predictive value, is a fraction of relevant instances among the models retrieved instances. This is measured as the number of true positives over the number of positive calls, a.k.a, true positives plus false positives. The recall, or sensitivity is the total number of true positives over the number of true positives plus false negatives. F1 is the harmonic mean of the precision and recall. Accuracy is the correct prediction, positive or negative across the set.

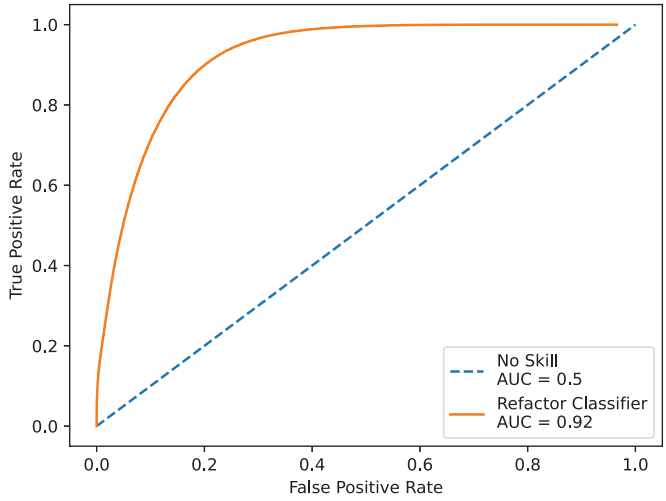


Fig. 4. Receiver operating characteristic curve of RP and RSP model. .92 AUC is considered an excellent score.

F1 is most appropriate for scoring these models. Accuracy is not appropriate as the test set is 78% negative examples and a model predicting only negatives would score 78% accuracy, but would be useless for predicting refactors. A higher precision, recall, and F1 indicates better performance. Formally, the equations for precision, recall, and F1 are as follows,

$$\text{Precision} = \frac{TP}{TP + FP} \quad \text{Recall} = \frac{TP}{TP + FN}$$

$$\text{F1} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

where TP is true positives, FP is false positives, and FN is false negatives.

Table II indicates that our model REFACTORBERT performs best in refactor prediction at the sequence level. The performance of the GRU is good, however, REFACTORBERT beats out all baselines because it can leverage its pretraining from CodeBERT [23]. Finally, observe the classifier's ROC curve in Fig. 4. The area under the curve (AUC) is 0.92 which indicates an "excellent" classifier. We use the Youden's J statistic[6] to find the optimal threshold for the test set to compare with our general purpose 50% threshold; the optimal threshold is 45% probability on 3 million validation examples so no threshold engineering is necessary as the model can generalize well (0.92 AUC). In the next section we will compare this performance with the performance of the RSP trained model.

D. Learning Spans of Refactors (RSP)

The Refactor Span Prediction task is a granular version of the Refactor Prediction task. In this section, we evaluate REFACTORBERT on learning specific tokens that must be changed. By narrowing the refactor to a specific set of tokens, the model can learn which spans correlate with refactors. We find that the model learning two objectives, REFACTORBERT$_{RP+RSP}$ in Table III, performs marginally

[5]https://huggingface.co/datasets/kevinjesse/ManyRefactors4C

[6]https://en.wikipedia.org/wiki/Youden%27s_J_statistic

TABLE III
PREDICTING SEQUENCE LEVEL VS. SPANS LEVEL PREDICTION (RP VS. RSP)

Model	P (%)	R (%)	F1 (%)	Acc (%)	p-val	CI (95%)
RefactorBERT_{RP}	56.68	89.23	69.32	82.53	< 0.0001	[82.50, 82.55]
$\text{RefactorBERT}_{RP+RSP}$	50.25	94.46	65.60	78.08	< 0.0001	[78.05, 78.10]

Model trained on both RP and RSP is still evaluated with RP prediction head only at sequence level. P-value and confidence interval are computed with a two-tailed binomial test.

TABLE IV
JOINT PREDICTION RESULTS (RP × RSP)

Model	P (%)	R (%)	F1 (%)	Acc (%)	p-Val	CI (95%)
Random Predictor	3.81	100.00	7.33	3.81	—	[3.809, 3.811]
$\text{RefactorBERT}_{RSP}$	43.35	**81.25**	56.54	95.246	< 0.0001	[95.245, 95.247]
$\text{RefactorBERT}_{RP \times RSP}$	**43.94**	80.83	**56.93**	**95.3456**	< 0.0001	[95.3450, 95.3462]

Evaluation across all tokens in RSP. Broadcasting RP on RSP predictions improves performance. P-value and confidence interval are computed with a two-tailed binomial test.

worse on the sequence level but gains the ability to specify which tokens contribute to its decision with minimal performance impact. This trade-off is worthwhile in practice because the sequence of suspect tokens can be searched among known refactors while providing an explanation to the developer (other than yes or no).

Table IV has the span level performance $\text{REFACTORBERT}_{RSP}$ across the sequences of tokens. This table measures the precision, recall, F1 and accuracy of each token in the sequence requiring a refactor or not. Scoring at the token level means partial credit is given to subsequences correctly classified, even if the entire sequence is not predicted correctly. We see that the token level performance is quite good, given that it is a more difficult task i.e. predicting each tokens' membership, and the random predictor scores very poorly. $\text{REFACTORBERT}_{RSP}$ is able to get over 81% of refactor prone tokens, and scores a 43% precision. We note that the ROC curve was nearly identical as the RP performance and also scored a 0.92 AUC.

The effectiveness of a model at a token level with a high AUC binary classifier translates to accurately annotating virtually any C/C++ code base, regardless of the version control platform (Siemens uses a variety of platforms), and the model can confidently lead developers to code segments that require additional focus.

RQ1: REFACTORBERT significantly outperforms the random predictor and is a strong binary classifier for refactors *in the wild* with an AUC of 0.92 and F1 of 69.32%.

Next we experiment with a joint prediction from both predictors for even better performance.

E. Joint Prediction RP × RSP

Naturally, we want to investigate if there is mutually exclusive information learned by the sequence level hidden state in addition to each token. Since the RP model performs better, we expected a projection of the RP value on the sequence of predictions (RSP) would lead to an improved effect. The RP objective could correct errors in the RSP prediction, namely, where tokens were flagged incorrectly. In this scenario, projecting a 0 across a sequence of 1s would correct a misprediction and the joint accuracy would increase.

Table IV shows the performance of broadcasting the RP prediction head to the RSP prediction head. We observe an improvement in performance specifically precision, F1, and accuracy. The joint model, formulated in Equation 6 is the best performing span prediction model.

RQ1 cont.: REFACTORBERT impressively scores 56.93% F1 score on the token level prediction compared to the random predictor's 7.33%.

In the next section we compare REFACTORBERT to REFDIFF 2.0 on the REFDIFF 2.0 evaluation set.

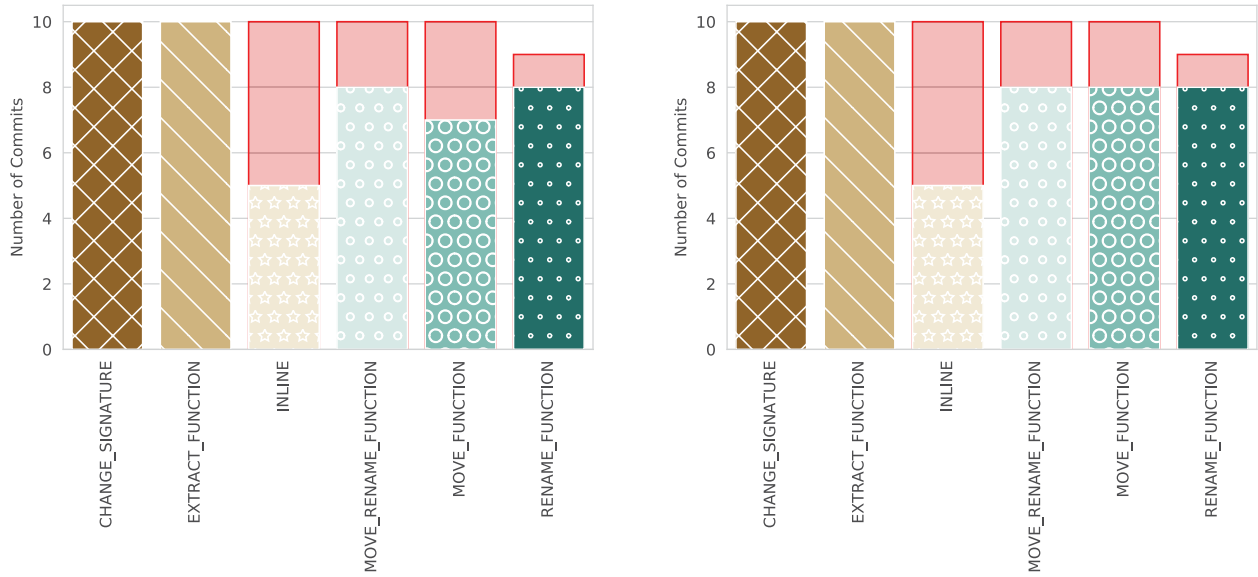
F. Comparison With REFDIFF 2.0

RefDiff 2.0 is a multiple language refactoring detection tool [13]. The original version, employed a combination of heuristics based on static analysis and code similarity to detect 13 well known refactoring types for Java [12]. REFDIFF 2.0 expands to C and JavaScript and introduces novel refactoring detection algorithms relying on code structure trees. A comparison with REFDIFF 2.0 calibrates REFACTORBERT against other state-of-the-art detection tools and helps to identify which refactoring types REFACTORBERT is capable of detecting. The evaluation data in REFDIFF 2.0 are manually verified across the *analyzed refactoring* (precision) and *documented refactoring* (recall) spreadsheets[7]. Our comparison is conducted as follows.

First we remove file level refactorings like file renamings since REFACTORBERT classifies source code directly. This results in a slightly smaller evaluation set of 60 examples in both precision and recall; both tables have differing sets of commits prior and coincidentally have the same number of examples after. Next, we remove 1 commit from each table (precision/recall) because two commit hashes exist in REFACTORBERT's dataset of refactoring commits. The evaluation is across 59 commits for both precision and recall. We checkout each commit and use the commit diff to get the exact token sequence under consideration. Then across the file, REFACTORBERT predicts whether the span requires a refactor. If any span is flagged, the commit must be a refactoring commit. The commit level prediction is a positive classification for a refactor. We compare our binary classifier results with those of REFDIFF 2.0 for precision and recall according to the published results for REFDIFF 2.0 C evaluation tables.

Table V shows the performance comparison between REFACTORBERT and REFDIFF 2.0. The models achieve similar precision, but REFACTORBERT performs much better in recall. It is our belief that REFACTORBERT is more sensitive to general refactoring patterns due to the sheer volume of data REFACTORBERT is trained on. Alternatively, the code structure tree in REFDIFF 2.0 might be less flexible to the various project domains and namespaces, resulting in less positive predictions and thus lower recall. The improved recall boosts the F1 score

[7]https://github.com/aserg-ufmg/RefDiff



(a) RefactorBERT (b) RefDiff 2.0

Fig. 5. Precision comparison of RefactorBERT and RefDiff 2.0 on RefDiff 2.0 C evaluation data. Red indicates incorrect predictions from the maximum possible total.

TABLE V
Comparision With RefDiff 2.0

Model	Precision (%)	Recall (%)	F1 (%)
RefDiff 2.0 [13]	83.05	86.44	84.71
RefactorBERT	81.36	**94.92**	**87.62**

higher than RefDiff 2.0, making it a stronger classifier over RefDiff 2.0 by nearly 3%.

Fig. 5(a) is a categorical visualization of the precision score in Table V. By refactor type, RefactorBERT misses only one more in `MOVE_FUNCTION`. Likewise, Fig. 6(a) is a categorical breakdown of recall on the commits in Table V. RefactorBERT performs much better than RefDiff by only missing three commits, one in `CHANGE_SIGNATURE`, `INLINE`, and `MOVE_RENAME_FUNCTION`. For more information on the refactor types, we direct the reader to the original article [13] and our published results[8]. High performance across verifiable refactors, in a variety of refactoring patterns, means RefactorBERT can be generally used to predict refactorings in *at least* these categories, likely much more, and not to mention RefactorBERT can be used in defect prediction which we discuss next.

RQ2: RefactorBERT detects more refactors than RefDiff 2.0 with nearly identical precision.

G. Refactor Models for Defect Prediction

Bug fixing and behavior changing code changes are often disguised under refactoring in tangled commits. A tangled commit is a commit where a developer bundles multiple code changing operations that address different problems; unfortunately many of these bug fixing commits are disguised as a refactor. There is a field of research dedicated to identifying tangled commits [69], [70], [71], and even specifically related to refactoring [72]. With a model trained on commits labeled as a "refactor", we are interested in how well RefactorBERT does at defect prediction.

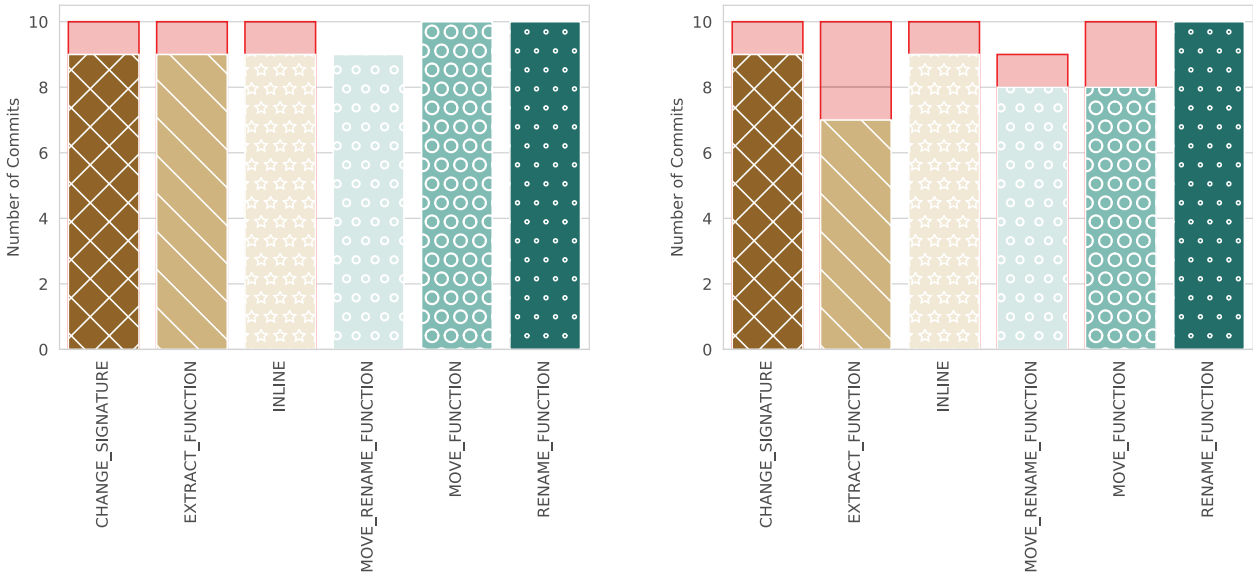
To test the hypothesis that RefactorBERT will be capable of predicting some defects, we employ an industry recognized suite of datasets and leaderboards called CodeXGLUE [30]. CodeXGLUE is a benchmark dataset and open challenge for code intelligence models. We refer the reader to the leaderboard[9] where RefactorBERT scores #3 only behind C-BERT [64] by 0.37%.

What makes RefactorBERT performance impressive is that most models on the leaderboard are specifically trained on defect prediction prior to being trained and evaluated on the benchmark. RefactorBERT out performs VulBERTa [65] which was trained on six vulnerability datasets: Vuldeepecker, Draper, REVEAL, muVuldeepecker, Devign, and D2A. Vuldeepecker and muVuldeepecker samples from two additional datasets: National Vulnerability Database (NVD) and Software Assurance Reference Dataset (SARD). RefactorBERT is not trained on explicitly labeled defects, but benefits from the occasional bug fix, or span of suspect code which in turn is easily tune-able in the benchmark training. See Table VI for the defect prediction results in CodeXGLUE.

RQ3: RefactorBERT captures defect related patterns as it performs competitively in CodeXGLUE's defect prediction task.

[8]https://github.com/kevinjesse/RefactorBERT

[9]https://microsoft.github.io/CodeXGLUE



(a) REFACTORBERT (b) REFDIFF 2.0

Fig. 6. Recall comparison of REFACTORBERT and REFDIFF 2.0 on REFDIFF 2.0 C evaluation data. Red indicates incorrect predictions from the maximum possible total.

TABLE VI
CODEXGLUE DEFECT PREDICTION RESULTS

Model	Accuracy (%)
CoText [63]	66.62
C-BERT [64]	65.45
RefactorBERT	**65.08**
VulBERTa-MLP [65]	64.75
VulBERTa-CNN [65]	64.42
PLBART [66]	63.18
code2vec [67]	62.48
CodeBERT [23]	62.08
RoBERTa [68]	61.05
TextCNN [30]	60.69
BiLSTM [30]	59.37

REFACTORBERT serves as the foundation for predicting refactors by evaluating spans of code. We expand REFACTORBERT into a code quality evaluator, by aggregating the identified suspect code and scoring it. We call this REFACTORSCORE.

V. APPLICABILITY OF REFACTORSCORE

REFACTORSCORE in its initial formulation is determined by the predicted cardinality of the refactor. Formally,

$$C = \sum_{i=1}^{N} \text{Pred}_{RSP}[i]$$

where C is the count of tokens that are predicted to require a refactor. The sequence value C is then mapped into one of five quantiles determined from the training set. The quantiles are determined by discretizing $C_{\text{Ground Truth}}$ of the training set into equal-sized buckets based on five sample quantiles. We direct the reader to Fig. 7, which are histograms of refactor length across each set of data. The similarity across various sampling shows that any quanitization in one set will generalize.

The quantiles determined by the training set are as follows,

$$\text{REFACTORSCORE} = \begin{cases} \text{A}, & 0 \leq C < 25 \\ \text{B}, & 25 \leq C < 57 \\ \text{C}, & 57 \leq C < 87 \\ \text{D}, & 87 \leq C < 121 \\ \text{F}, & 121 \leq C < 512 \end{cases} \quad (7)$$

We choose to generate the quantiles on the training set because it consists of 42.5 million sequences, where the validation set only has 3.98 million sequences; the test set has 9.09 million sequences. For the training set, the average refactor length is 80.53 tokens and the median is 73 tokens. We found these statistics to be similar to validation and test set.

In the next section, we evaluate the discretization on the test set.

A. Accuracy of RefactorScore

For REFACTORSCORE to work, two generalizations must occur: (1) The model REFACTORBERT must be capable of generalizing to the test set and accurately predicting spans of refactors, and (2) the quantiles from the 42.5 million training examples must be generally applicable to a test set. With strong performance in the former, see Table III, and statistically similar distributions in the latter, Fig. 7, we can reasonably expect good performance in REFACTORSCORE.

The accuracy across all test set examples is 79.56%, meaning, that the grade A-F provided by REFACTORSCORE is the same label as the ground truth for both positive and negative outcomes. The accuracy across the subset of examples requiring a refactor is 71.18%. Thus we note that the negative outcomes are not

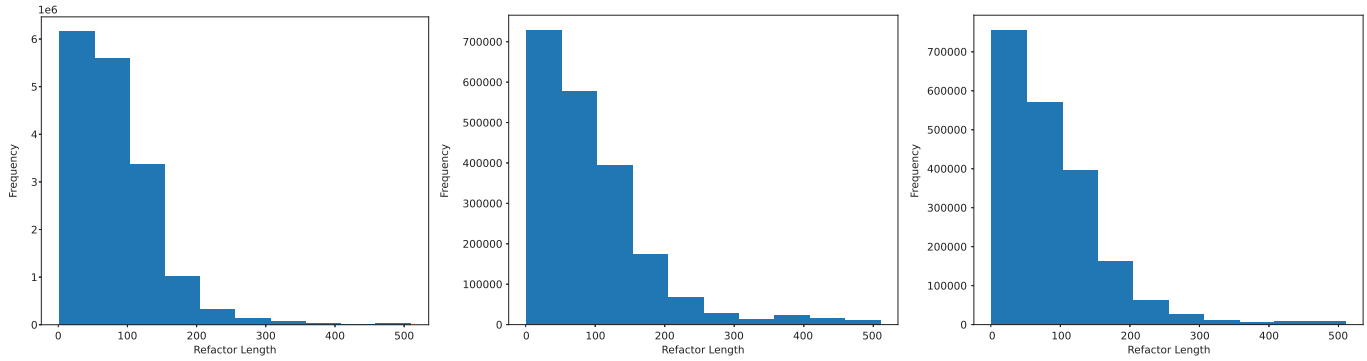


Fig. 7. Histogram of refactor length.

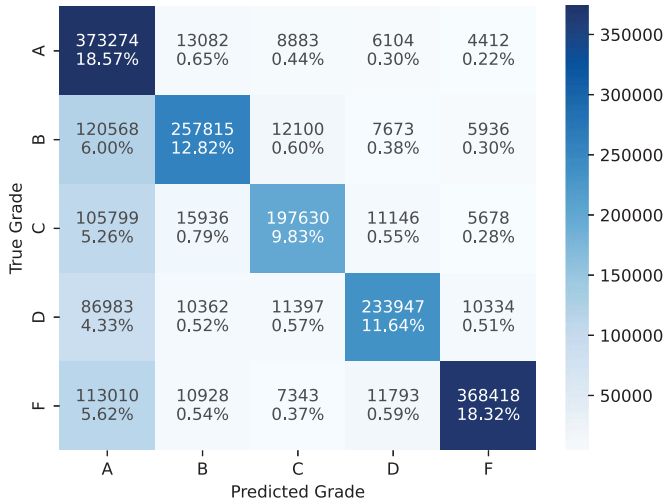


Fig. 8. REFACTORSCORE confusion matrix measuring accuracy by grade A - F (A is minor refactor by refactor length).

confounding the total accuracy score. Lastly, REFACTORSCORE is not correlated by file length. Across the 9.09 million examples in the test set, the correlation between code length and REFACTORSCORE's refactor length is -.325 Pearson score indicating there is a slight negative correlation.

Finally, we direct the reader to the REFACTORSCORE confusion matrix (Fig. 8). The confusion matrix calculates the accuracy of each class and shows where REFACTORSCORE makes correct and incorrect predictions. Fig. 8 is consistent with a strong classifier, namely, where the ground truth is correctly predicted (diagonal). The model incorrectly predicts a minor refactor when more substantial refactoring is the correct answer. The uniform underfitting across all categories in column A is indicative of aleatoric uncertainty, likely, from the sampling of commit histories. In other words, unchanged code in a prior commit might ultimately be changed in a later commit, thus making the prior commits' labels noisy. The changing nature of code makes modeling repositories with their commit history a challenging task. For now, it is beyond the scope of this article.

In the following section, we apply a trained REFACTORSCORE to gain insights about open-source repositories' code quality.

B. Scoring Open Source Projects

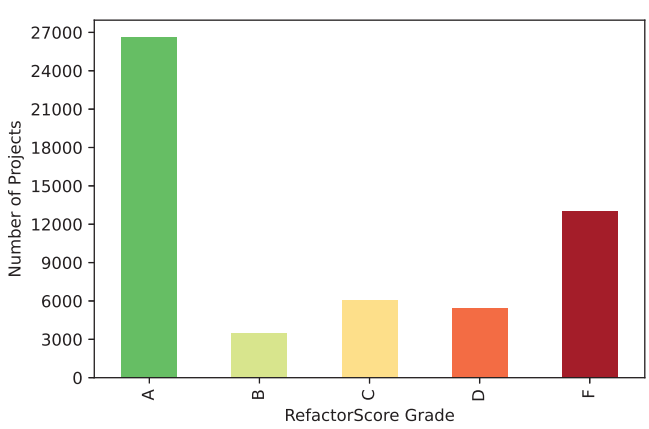
In this section we use REFACTORSCORE to grade open source projects. First we apply REFACTORSCORE at a project level on the test set and then examine several top projects current HEAD commit on GitHub.

1) Snippets From Test Set: Starting with the many preprocessed projects in our test set, the projects are scored with REFACTORSCORE (Fig. 9). Fig. 9(a) is a bar plot of REFACTORSCORE on project source files. The median score is reported at the project level giving an overall project score. Notice that most projects score quite well. However approximately 15,000 projects fail with a score of an F accounting for about 25% of projects. This means, across all code snippets in the project, the median is a failing grade indicating over 121 tokens in each window require a refactor. This amounts to at least 23.6% of tokens (or higher) need to be refactored. In summary, Fig. 9(a) shows how projects in the test set generally score.

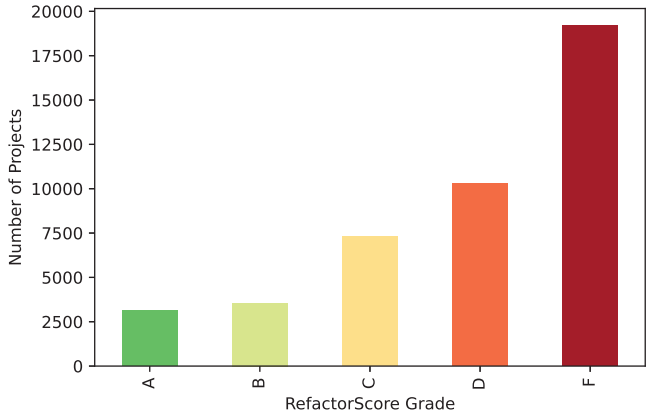
Given that a refactor is required in the span, Fig. 9(b) demonstrates how bad that refactor is expected to be. Across all the spans requiring a refactor, the goal in this plot is to see how bad on average these refactors are according to REFACTORSCORE. Fig. 9(b) shows that the median refactors across a singular project is more then often in the C-F range. This shows that REFACTORSCORE can be used to target specific projects, directories, and files to gauge quality.

2) Top Starred Projects: Next we apply REFACTORSCORE to a list of top 60 GitHub projects found here[10]. We checkout the current HEAD on the main branch and feed the source code to REFACTORSCORE. The individual project scores can be viewed in Fig. 10. The average refactor span length is 26.08 with the projects on average scoring a B; the average computed at project level. Within these projects there were many third party libraries. We filter these directories and perform REFACTORSCORE on both the third party library directories and the remainder of the project. Third party software scores a bit worse than the remainder of the code at 32.16 refactor span length (681,391 snippets) where as the remainder of the project is closer to the 60 project average of 25.89 span length

[10]https://github.com/EvanLi/Github-Ranking/blob/master/Top100/CPP.md



(a) Grades of Projects in Test



(b) The median grade, by project, of code tokens when a refactor is required (a.k.a how smelly is the refactor?)

Fig. 9. REFACTORSCORE across projects in test set.

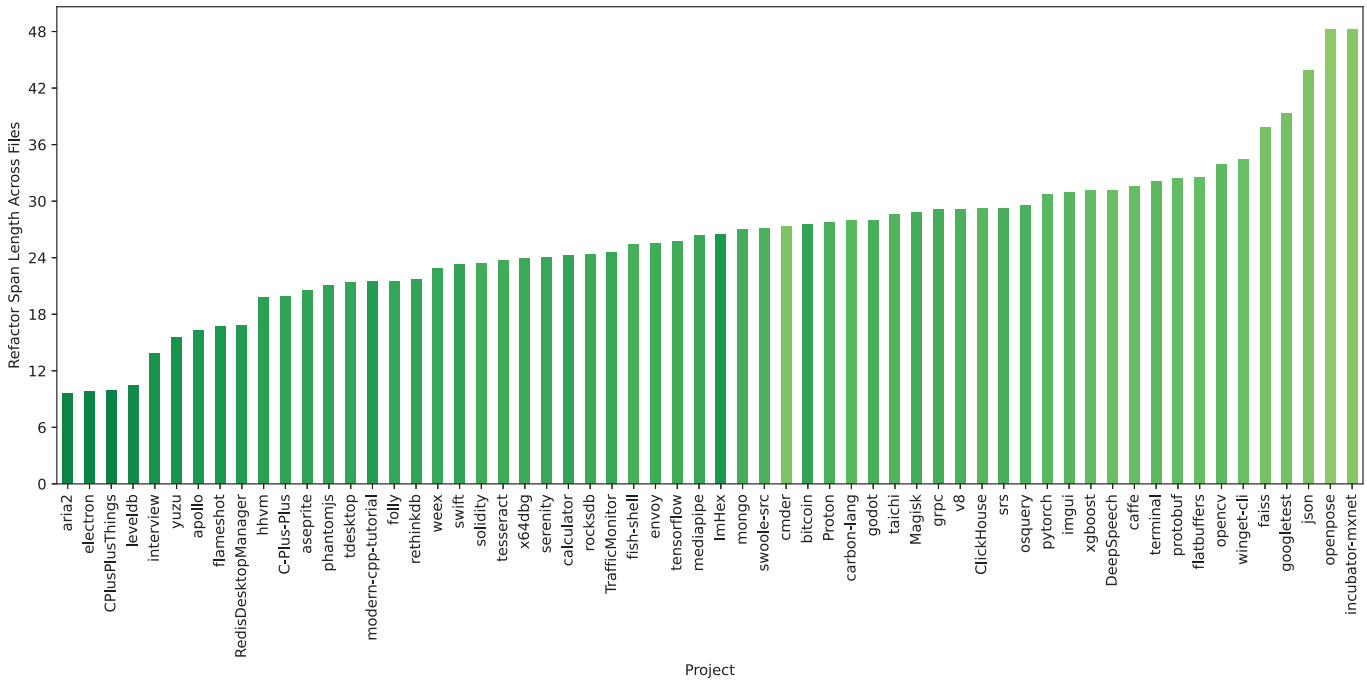


Fig. 10. Grading the most starred projects with REFACTORSCORE @ HEAD commit. The color represents the average grade across all code spans and the bar high indicates the average span length. Notice the grade decreases as the refactor is more severe.

(871,505 snippets). The color of the columns correspond to an average grade for the repository.

Additionally, we investigate how projects change over time. Fig. 11 shows the mean refactor span grade over the past 10 years for the top 60 starred projects. The span length of less than 25 is deemed an A and above 25 is considered a B. The general trend over the last 10 years are an increasing amount of code snippets that should be refactored. This is intuitive as many more pull requests for specific features and issues, often results in code that must be reworked later.

Finally, we observe how REFACTORSCORE changes across the most popular subset of projects in each year. Fig. 12 plots the average span length for the ten most popular projects. While some projects stay the same or decrease (TensorFlow and Electron), the remaining projects increase in code requiring a refactor.

RQ4: REFACTORSCORE demonstrates a degrade in code quality across top open source projects from refactor length increasing; ultimately this indicates an rise in technical debt.

C. *REFACTORSCORE in Out-of-Domain Setting*

In this section, we evaluate REFACTORSCORE on Siemens proprietary code. Siemens proprietary code is considered out-of-domain for REFACTORBERT because the code is domain

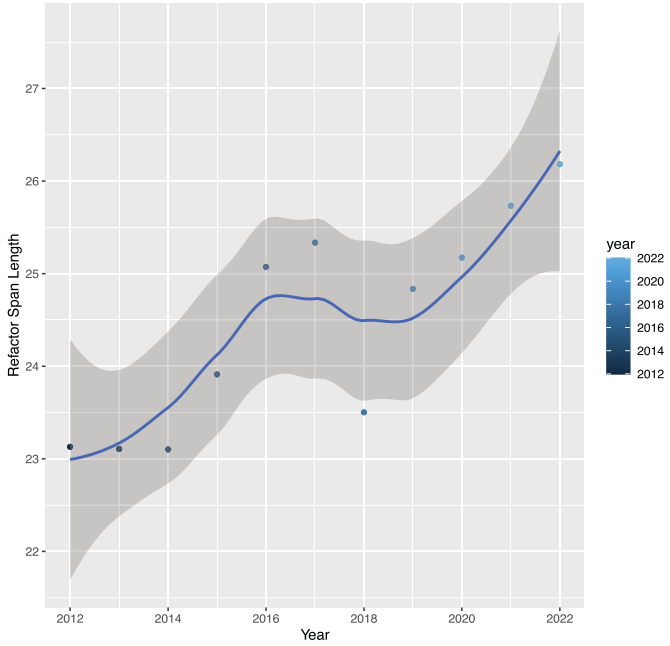


Fig. 11. REFACTORSCORE over the past 10 years averaged by project and time interval.

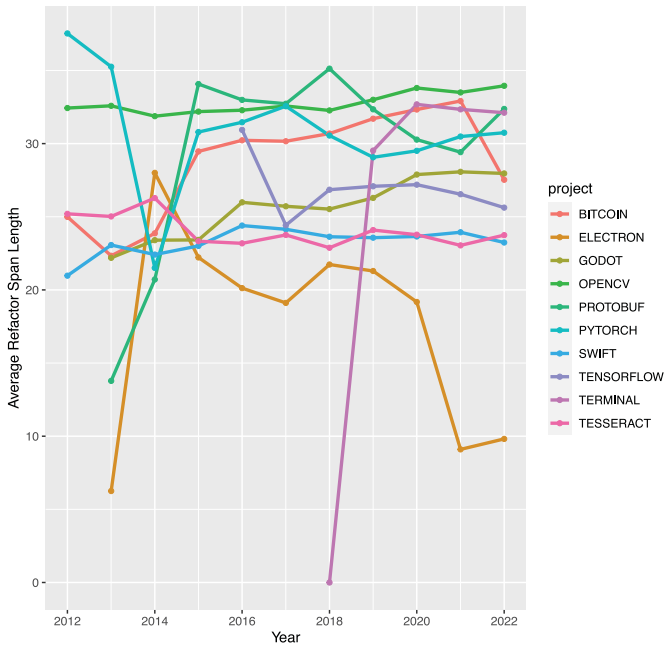


Fig. 12. A subset of the top open source projects plotting the average refactor span length over the last 10 years.

specific and inherits little from open source. Arguably, proprietary code presents a challenge to REFACTORBERT and REFACTORSCORE.

In the following paragraphs, we detail the validation of REFACTORSCORE with human developers by showing them ten snippets of code they may or may not be familiar with. Five of these snippets are graded by REFACTORSCORE to require a sizeable refactoring and the other five are negative examples as predicted by REFACTORSCORE. The snippets are selected by

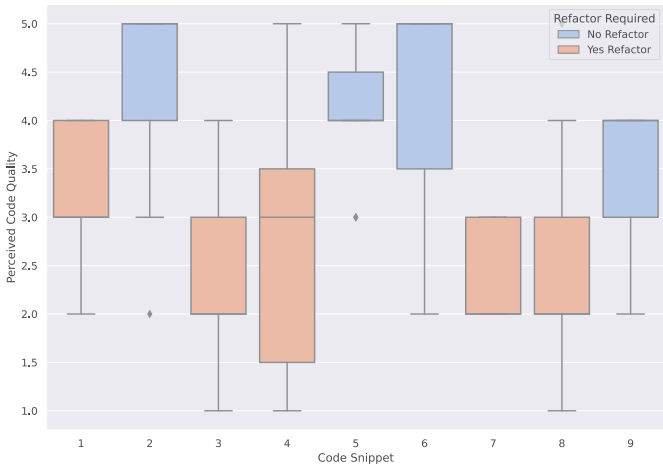


Fig. 13. Code quality across all code snippets ranked by 11 developers. Low scoring code requires a refactor and high scoring code does not.

the worst and best grades from REFACTORBERT. We ask the developers the following questions:

1) Does this code require a refactor?
2) If so, where do you suspect the refactor should take place?
3) What would you rate the quality of this code snippet? (1-5, 5 being best)
4) If this snippet was flagged, would it have led to a useful refactoring, provided the required refactor effort?

As a result of this study, we gauge the usefulness of REFACTORSCORE in a new domain of code. Fig. 13 is the results of "What would you rate the quality of this code snippet?". Code quality prediction from REFACTORSCORE matches developers' understanding code quality. Code that does not require a refactor per developers, was not predicted to need a refactor by REFACTORSCORE. The result of this study can be explained in the Spearman Correlation between the developers' recommendation to refactor, stated code quality, useful refactor, and model prediction (Table VII). The developers' recommendation to refactor (Question 1) and useful refactor (Question 4) have a correlation of 0.527 to the model's prediction; this is a moderate positive correlation coefficient. Likewise, the model's prediction relative to the code quality is a moderate negative correlation. Finally, the developer recommends refactoring when code exhibits low quality and the refactoring would be worthwhile of the development effort.

RQ5: Developers agree with REFACTORBERT's interpretation of code quality as segments flagged by REFACTORBERT have a .527 Spearman Correlation with developers.

Finally, during the developer survey, subjects that had been involved with the development of the code often rationalized certain design decisions. Some of the justifications were clearly biased from previous constraints, like code compliance and logging standards, imposed on the developers during the development lifecycle. While the developers acknowledge the poor code quality after being told where the

TABLE VII
SPEARMAN CORRELATION FROM DEVELOPER STUDY

	Developer	Code Quality	Useful Refactor?	Model
Developer	1.000	-0.821	0.833	0.527
Code Quality	-0.821	1.000	-0.746	-0.557
Useful Refactor?	0.833	-0.746	1.000	0.527
Model	0.527	-0.557	0.527	1.000

TABLE VIII
DEVELOPER RESULTS ON INTERNAL PROJECT. OBSERVE THE HIGHER SCORES FROM OUTSIDE DEVELOPERS

Developer	Precision	Recall	Accuracy	Experience?
1	0.667	0.8	0.667	False
2	0.800	0.8	0.778	False
3	0.625	1.0	0.667	True
4	0.833	1.0	0.889	False
5	0.800	0.8	0.778	False
6	1.000	1.0	1.000	False
7	1.000	1.0	1.000	False
8	0.833	1.0	0.889	False
9	0.600	0.6	0.556	True
10	0.667	0.8	0.667	True
11	0.667	0.4	0.556	True

TABLE IX
INTERRATER AGREEMENT (FLEISS KAPPA)

Question	Fleiss Kappa
Does this code require a refactor?	0.26 (Fair Agreement)
What would you rank the quality of this code snippet? (1–5)	0.12 (Slight agreement)
Would this be an useful refactoring?	0.24 (Fair Agreement)

problem was, they exhibited resistance despite the obvious need to change the code. This poses a serious challenge for balancing technical debt.

We also found that developers that worked on the project scored significantly worse on average than fresh developers. Table VIII illustrates this phenomena quantitatively. Again we saw reasoning and justification for bad code and more elevated scoring universally across bad examples. By automatically scoring the code with REFACTORSCORE, we can bypass some potential internal bias exhibited by the developers.

RQ6: Developers that have experience on the code base perform markedly lower in survey precision and accuracy when identifying poor code quality.

From this developer survey we can takeaway some insights,

1) REFACTORSCORE predictions are indicative of low quality code.
2) Segments predicted poorly by REFACTORSCORE are worth refactoring.
3) Refactoring low-quality code from REFACTORSCORE are good investments towards the improvement of code quality.
4) Automatic metrics like REFACTORSCORE can circumvent developer implicit bias when deciding on code quality and refactoring needs.

In the following section, we discuss how REFACTORSCORE can be weighed or adjusted to address business and development needs.

D. Deriving Alternative REFACTORSCORES

Deriving alternative scoring techniques with REFACTORBERT is possible. Such augmentations permit organizations to prioritize refactor prone hotspots in different ways.

Code smells, *viz.* problematic coding structures, such as complex classes, dead code, duplicate code, unnecessary loops, etc. can be incorporated into REFACTORSCORE. Corpus statistics of such design problems can weigh the token sequences that match both design patterns and REFACTORBERT refactors. This would give the organization the ability to prioritize particular refactor prone code, specifically at the design level.

At a product level, there might be critical components that should be targeted above all others. By weighting vital classes or specific features that run with performance constraints, the business unit can more proactively fix expensive refactor prone code, thus optimizing for the organizations budget and customer expectations. Code that consistently monopolizes developers time, requires the most developers, or cost the most over the products lifecycle are valid means of weighting REFACTORBERT.

Finally, previous refactors and bugs are effective predictors for future defects. Previous research has cited the effectiveness of considering previously defective code elements [73], [74], [75], [76]. This promising line of work can be integrated into REFACTORSCORE by automatic labeling and fine-tuning of REFACTORBERT further, or by using global statistics from previous commits and superimposing across REFACTORBERT predictions. We plan to explore this in future work.

VI. THREATS TO VALIDITY

The focus of this work implies that the main threats to its validity are external.

A. External Validity

In this article we assess the ability of a sequence based machine learning model to properly identify refactoring commit spans. The projects are sampled according to their popularity which often correlates with the size of the project both in commit history and absolute source code tokens; this is a widely adopted practice. Generalization in machine learning models on software engineering tasks are convoluted by code clones and code reuse. Our findings show that REFACTORBERT works for over 18,000 projects, however, all of these projects are publically accessible. Further studies are needed to confirm the models capability across code that has little or no influence from the open source community. This threat could be ameliorated in a variety of ways: by further intra-project training on commit history, training on products that share a specific domain, incorporating product level knowledge such as performance requirements or specific known system issues such as timeouts.

B. Internal Validity

A challenge in this study was the variance of the interpretation of a refactoring commit. In a classical sense, RefactorBERT should not care about unreliable or frequently changed code, but only the adoption of design patterns that reduce internal complexity and preserve functionality. In practice, developers often confound bug fixes, security updates, and functional changes as refactors. Fundamentally, software quality is formulated beyond design patterns and often includes code correctness and security requirements, all of which RefactorBERT trained on. We believe that a variation of RefactorBERT trained with a tool like RefactorMiner [11] or RefDiff [13] could identify specific problematic design patterns. As such, the performance of such a model would be limited to the existing performance of the pattern matcher. Concluding this thought, we aim to take advantage of large scale data to generally grade software quality and specific refactoring practices is not comprehensive to this goal, especially for refactors *in the wild*.

Another challenge is the varying interpretation of code quality for the developer survey. The four questions for each code snippet, gave the developer an opportunity to reason about their code quality score. No answers were provided to the developers as that could bias future decisions. Some code snippets had outlier code quality values, but over the course of 11 developers with varying experience, we establish a strong trend in the inter-quartile range. As such, we are confident that our developers responses are consistent with a broader set of subjects.

Coding languages, libraries, and practices evolve over time. The generalization capability of machine learning models is a challenge. RefactorBERT was trained on project commit history making the time series experiments biased towards data the model has seen. Empirically, the RefactorScore across the 60 projects and the respective subset was not sampled at a refactoring commit, but at each yearly interval and evaluated across each token of code base. Thus, the effect of data bias for a particular span would be minimal in the final results, even if a few spans are properly graded from data leakage.

Finally, code duplication can confound performance scores. We only sample changing code on a parent of the refactoring commit in regions that are part of the commit diff. Then performance is measured with precision and recall so that duplicate negative examples are not impacting reported performance. Moreover, performance is likely hindered from negative sampling where code that is changed at a later date is sampled as a negative example in an earlier commit. Refactoring changes adopted widespread across projects is confounder to performance. At the span level, there are minimal duplicates as we sampled only 242 out of 1.3M or 0.018% with a cosine similarity of greater than 0.9.

Lastly, code deduplication tools are largely not useful for this body of work, as overlapping contexts around a diff area throughout the file's commit histories leads to a high number of false positives.

VII. Future Work

As discussed in Section V-D, alternative formulations of RefactorScore have the ability to impact business units by concentrating refactoring efforts. Siemens has many products at various stages of the software development lifecycle that benefit from differing interpretations of RefactorScore. Products in development benefit long term from fixing design problems early to avoid long term maintainability issues, technical complexity, and scaling limitations. Software sustainment teams can benefit from recommended third party refactors, bug fixes, and code simplification where code is given a poor RefactorScore.

Lastly, we intend to improve the refactor explanations. To search through the 70 million code documents, we index the files with the commit diff, commit message, project and file level information. With the ElasticSearch engine, developers can walk through paginated commits that are similar to the span in question by RefactorBERT. In future work, we plan to extend the service to include natural language explanations in addition to analogous code examples. This likely entails a form of code summarization between the commit features and source code.

Finally, we plan on expanding the RefDiff 2.0 C evaluation set to include C++. We also plan to mine more refactoring patterns in both languages so that we can discover additional patterns RefactorBERT is capable of detecting.

VIII. Conclusion

In this work, we present an automatic evaluation metric for code. RefactorScore is different from existing tools because it scores spans of code with only existing code, needs no commit history, and is version control invariant. The scores are human interpretable grades where the underlying model, RefactorBERT, highlights the refactor prone tokens that drive the overall grade. As a code quality tool, RefactorScore shows high accuracy ($\sim$ 80%) on grading code A-F and these grades correlate with 11 human developers interpretation of code quality. Developers agree that the code presented by RefactorBERT and scored by RefactorScore should be refactored *and* that the refactor would be a worthwhile refactoring. We found that training models on *in the wild* refactorings, meaning many entangled maintenance operations, did not impact the model's ability to detect refactoring patterns competitive tools can detect; in fact we expect many other patterns to be captured by our model.

RefactorBERT outperforms the state of the art comparison for C programming language, RefDiff 2.0 by $\sim$ 3% F1-score for a total score of 87.62%. RefactorBERT achieves $\sim$95% recall meaning that RefactorBERT detects on average 5 more refactors than RefDiff 2.0 on refactor patterns specifically selected for RefDiff 2.0. Moreover, the thresholding problem [11], which plagues other machine learning and statistical models in refactor detection, does not affect RefactorBERT as shown by the excellent ROC curve. We test RefactorBERT on the defect prediction task in CodeXGLUE [30] and find that

it scores among the top models; we infer this is because of the *in the wild* training.

How do we envision the use of REFACTORSCORE and REFACTORBERT?

1) Developers can use REFACTORBERT in live coding environments to highlight suspect lines of code with the corresponding REFACTORSCORE grade.
2) Developers can use REFACTORSCORE to ensure code quality prior to code being committed into the code base.
3) Developers should use REFACTORSCORE retroactively to fix outstanding code quality issues.
4) REFACTORSCORE can be weighted for organizational needs such as targeting performance constraints.

ACKNOWLEDGMENT

The authors like to thank the participants of the code quality survey. The authors especially appreciate Christoph Brand for his support of this work.

REFERENCES

[1] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Reading, MA, USA: Addison-Wesley, 2018.
[2] S. Kannangara and W. Wijayanayake, "An empirical evaluation of impact of refactoring on internal and external measures of code quality," 2015, *arXiv:1502.03526*.
[3] R. Leitch and E. Stroulia, "Assessing the maintainability benefits of design restructuring using dependency analysis," in *Proc. 5th Int. Workshop Enterprise Netw. Comput. Healthcare Industry (IEEE Cat. No. 03EX717)*, Piscataway, NJ, USA: IEEE Press, 2004, pp. 309–322.
[4] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 5–18, Jan./Feb. 2012.
[5] N. Tsantalis, V. Guana, E. Stroulia, and A. Hindle, "A multidimensional empirical study on refactoring activity," in *Proc. Conf. Center Adv. Stud. Collaborative Res. (CASCON)*, 2013, pp. 132–146.
[6] M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits," in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, 2012, pp. 1–11.
[7] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? Confessions of github contributors," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2016, pp. 858–870.
[8] O. Chaparro, G. Bavota, A. Marcus, and M. Di Penta, "On the impact of refactoring operations on code quality metrics," in *Proc. IEEE Int. Conf. Softw. Maintenance Evolution*, Piscataway, NJ, USA: IEEE Press, 2014, pp. 456–460.
[9] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proc. 28th Int. Conf. Softw. Eng.*, 2006, pp. 452–461.
[10] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proc. 27th Int. Conf. Softw. Eng.*, 2005, pp. 284–292.
[11] N. Tsantalis, A. Ketkar, and D. Dig, "RefactoringMiner 2.0," *IEEE Trans. Softw. Eng.*, vol. 48, no. 3, pp. 930–950, Mar. 2020.
[12] D. Silva and M. T. Valente, "RefDiff: Detecting refactorings in version histories," in *Proc. IEEE/ACM 14th Int. Conf. Mining Softw. Repositories (MSR)*, Piscataway, NJ, USA: IEEE Press, 2017, pp. 269–279.
[13] D. Silva, J. P. da Silva, G. Santos, R. Terra, and M. T. Valente, "RefDiff 2.0: A multi-language refactoring detection tool," *IEEE Trans. Softw. Eng.*, vol. 47, no. 12, pp. 2786–2802, Dec. 2021.
[14] T. Mariani and S. R. Vergilio, "A systematic review on search-based refactoring," *Inf. Softw. Technol.*, vol. 83, pp. 14–34, Mar. 2017.
[15] M. O'Keeffe and M. Ó. Cinnéide, "Search-based refactoring: An empirical study," *J. Softw. Maintenance Evolution, Res. Pract.*, vol. 20, no. 5, pp. 345–364, 2008.
[16] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring," *J. Syst. Softw.*, vol. 107, pp. 1–4, Sep. 2015.
[17] M. Aniche, E. Maziero, R. Durelli, and V. Durelli, "The effectiveness of supervised machine learning algorithms in predicting software refactoring," *IEEE Trans. Softw. Eng.*, vol. 48, no. 4, pp. 1432–1450, Apr. 2022.
[18] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit, "Ref-Finder: A refactoring reconstruction tool based on logic query templates," in *Proc. 18th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2010, pp. 371–372.
[19] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automatic detection of refactorings for libraries and frameworks," in *Proc. Workshop Object Oriented Reengineering (WOOR)*, 2005.
[20] Z. Xing and E. Stroulia, "The JDEvAn tool suite in support of object-oriented evolutionary development," in *Proc. Companion 30th Int. Conf. Softw. Eng.*, 2008, pp. 951–952.
[21] Z. Xing and E. Stroulia, "UMLDiff: An algorithm for object-oriented design differencing," in *Proc. 20th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2005, pp. 54–65.
[22] R.-M. Karampatsis, H. Babii, R. Robbes, C. Sutton, and A. Janes, "Big code!= big vocabulary: Open-vocabulary models for source code," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2020, pp. 1073–1085.
[23] Z. Feng et al., "CodeBERT: A pre-trained model for programming and natural languages," 2020, *arXiv:2002.08155*.
[24] D. Guo et al., "GraphCodeBERT: Pre-training code representations with data flow," 2020, *arXiv:2009.08366*.
[25] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," 2021, *arXiv:2109.00859*.
[26] M. Chen et al., "Evaluating large language models trained on code," 2021, *arXiv:2107.03374*.
[27] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *Proc. 6th ACM SIGPLAN Int. Symp. Mach. Program.*, 2022, pp. 1–10.
[28] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," 2015, *arXiv:1508.07909*.
[29] T. Kudo and J. Richardson, "Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing," 2018, *arXiv:1808.06226*.
[30] S. Lu et al., "CodeXGLUE: A machine learning benchmark dataset for code understanding and generation," 2021, *arXiv:2102.04664*.
[31] A. Peruma, M. W. Mkaouer, M. J. Decker, and C. D. Newman, "Contextualizing rename decisions using refactorings, commit messages, and data types," *J. Syst. Softw.*, vol. 169, Nov. 2020., Art. no. 110704.
[32] N. Brown et al., "Managing technical debt in software-reliant systems," in *Proc. FSE/SDP Workshop Future Softw. Eng. Res.*, 2010, pp. 47–52.
[33] F. A. Fontana, V. Ferme, and S. Spinelli, "Investigating the impact of code smells debt on quality code evaluation," in *Proc. 3rd Int. Workshop Manag. Tech. Debt (MTD)*, Piscataway, NJ, USA: IEEE Press, 2012, pp. 15–22.
[34] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis," *Inf. Softw. Technol.*, vol. 108, pp. 115–138, Apr. 2019.
[35] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Trans. Softw. Eng.*, vol. 30, no. 2, pp. 126–139, Feb. 2004.
[36] A. A. B. Baqais and M. Alshayeb, "Automatic software refactoring: A systematic literature review," *Softw. Qual. J.*, vol. 28, no. 2, pp. 459–502, 2020.
[37] I. H. Moghadam and M. Ó Cinnéide, "Code-Imp: A tool for automated search-based refactoring," in *Proc. 4th Workshop Refactoring Tools*, 2011, pp. 41–44.
[38] M. Mohan and D. Greer, "A survey of search-based refactoring for software maintenance," *J. Softw. Eng. Res. Develop.*, vol. 6, no. 1, pp. 1–52, 2018.
[39] F. Pecorelli, F. Palomba, D. Di Nucci, and A. De Lucia, "Comparing heuristic and machine learning approaches for metric-based code smell detection," in *Proc. IEEE/ACM 27th Int. Conf. Program Comprehension (ICPC)*, Piscataway, NJ, USA: IEEE Press, 2019, pp. 93–104.
[40] F. A. Fontana and M. Zanoni, "Code smell severity classification using machine learning techniques," *Knowl.-Based Syst.*, vol. 128, pp. 43–58, Jul. 2017.
[41] F. Arcelli Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Softw. Eng.*, vol. 21, no. 3, pp. 1143–1191, 2016.
[42] F. A. Fontana, M. Zanoni, A. Marino, and M. V. Mäntylä, "Code smell detection: Towards a machine learning-based approach," in *Proc. IEEE*

Int.Conf. Softw. Maintenance, Piscataway, NJ, USA: IEEE Press, 2013, pp. 396–399.
[43] N. Pritam et al., "Assessment of code smell for predicting class change proneness using machine learning," *IEEE Access*, vol. 7, pp. 37414–37425, 2019.
[44] F. Pecorelli, D. Di Nucci, C. De Roover, and A. De Lucia, "A large empirical assessment of the role of data balancing in machine-learning-based code smell detection," *J. Syst. Softw.*, vol. 169, 2020, Art. no. 110693.
[45] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia, "Detecting code smells using machine learning techniques: Are we there yet?" in *Proc. IEEE 25th Int. Conf. Softw. Anal., Evolution Reengineering (SANER)*, Piscataway, NJ, USA: IEEE Press, 2018, pp. 612–621.
[46] S. Demeyer, S. Ducasse, and O. Nierstrasz, "Finding refactorings via change metrics," *ACM SIGPLAN Notices*, vol. 35, no. 10, pp. 166–177, 2000.
[47] G. Antoniol, M. Di Penta, and E. Merlo, "An automatic approach to identify class evolution discontinuities," in *Proc. 7th Int. Workshop Princ. Softw. Evolution*, Piscataway, NJ, USA: IEEE Press, 2004, pp. 31–40.
[48] P. Weissgerber and S. Diehl, "Identifying refactorings from source-code changes," in *Proc. 21st IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Piscataway, NJ, USA: IEEE Press, 2006, pp. 231–240.
[49] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated detection of refactorings in evolving components," in *Proc. Eur. Conf. Object-Oriented Program.*, Berlin, Germany: Springer-Verlag, 2006, pp. 404–428.
[50] N. Tsantalis, M. Mansouri, L. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2018, pp. 483–494.
[51] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Piscataway, NJ, USA: IEEE Press, 2010, pp. 1–10.
[52] O. De Moor et al., "Keynote address:.QL for source code analysis," in *Proc. 7th IEEE Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*, Piscataway, NJ, USA: IEEE Press, 2007, pp. 3–16.
[53] V. Lenarduzzi, F. Lomio, H. Huttunen, and D. Taibi, "Are sonarQube rules inducing bugs?" in *Proc. IEEE 27th Int. Conf. Softw. Anal., Evolution Reengineering (SANER)*, Piscataway, NJ, USA: IEEE Press, 2020, pp. 501–511.
[54] D. Marcilio, R. Bonifácio, E. Monteiro, E. Canedo, W. Luz, and G. Pinto, "Are static analysis violations really fixed? A closer look at realistic usage of sonarqube," in *Proc. IEEE/ACM 27th Int. Conf. Program Comprehension (ICPC)*, Piscataway, NJ, USA: IEEE Press, 2019, pp. 209–219.
[55] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, A. Marcus, and G. Canfora, "ARENA: An approach for the automated generation of release notes," *IEEE Trans. Softw. Eng.*, vol. 43, no. 2, pp. 106–127, Feb. 2017.
[56] J. Ratzinger, T. Sigmund, and H. C. Gall, "On the relation of refactorings and software defect prediction," in *Proc. Int. Work. Conf. Mining Softw. Repositories*, 2008, pp. 35–38.
[57] G. Soares, R. Gheyi, E. Murphy-Hill, and B. Johnson, "Comparing approaches to analyze refactoring activity on software repositories," *J. Syst. Softw.*, vol. 86, no. 4, pp. 1006–1022, 2013.
[58] I. Kádár, P. Hegedűs, R. Ferenc, and T. Gyimóthy, "A manually validated code refactoring dataset and its assessment regarding software maintainability," in *Proc. 12th Int. Conf. Predictive Models Data Analytics Softw. Eng.*, 2016, pp. 1–4.
[59] A. Vaswani et al., "Attention is all you need," in *Proc. 31st Int. Conf. Adv. Neural Inf. Process. Syst. (NIPS)*, vol. 30, 2017, pp. 6000–6010.
[60] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.
[61] K. Cho, B. Van Merriënboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches," 2014, *arXiv:1409.1259*.
[62] D. Spadini, M. Aniche, and A. Bacchelli, "PyDriller: Python framework for mining software repositories," in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2018, pp. 908–911.
[63] L. Phan et al., "CoTexT: Multi-task learning with code-text transformer," 2021, *arXiv:2105.08645*.
[64] L. Buratti et al., "Exploring software naturalness through neural language models," 2020, *arXiv:2006.12641*.
[65] H. Hanif and S. Maffeis, "VulBERTa: Simplified source code pre-training for vulnerability detection," 2022, *arXiv:2205.12424*.
[66] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," 2021, *arXiv:2103.06333*.
[67] D. Coimbra, S. Reis, R. Abreu, C. Păsăreanu, and H. Erdogmus, "On using distributed representations of source code for the detection of C security vulnerabilities," 2021, *arXiv:2106.01367*.
[68] Y. Liu et al., "RoBERTa: A robustly optimized BERT pretraining approach," 2019, *arXiv:1907.11692*.
[69] K. Herzig, S. Just, and A. Zeller, "The impact of tangled code changes on defect prediction models," *Empirical Softw. Eng.*, vol. 21, no. 2, pp. 303–336, 2016.
[70] M. Dias, A. Bacchelli, G. Gousios, D. Cassou, and S. Ducasse, "Untangling fine-grained code changes," in *Proc. IEEE 22nd Int. Conf. Softw. Anal., Evolution, Reengineering (SANER)*, Piscataway, NJ, USA: IEEE Press, 2015, pp. 341–350.
[71] H. Kirinuki, Y. Higo, K. Hotta, and S. Kusumoto, "Hey! Are you committing tangled changes?" in *Proc. 22nd Int. Conf. Program Comprehension*, 2014, pp. 262–265.
[72] A. Bagheri and P. Hegedűs, "Is refactoring always a good egg? Exploring the interconnection between bugs and refactorings," in *Proc. IEEE/ACM 19th Int. Conf. Mining Softw. Repositories (MSR)*, Piscataway, NJ, USA: IEEE Press, 2022, pp. 117–121.
[73] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in *Proc. 7th IEEE Work. Conf. Mining Softw. Repositories (MSR)*, Piscataway, NJ, USA: IEEE Press, 2010, pp. 31–41.
[74] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller, "Predicting faults from cached history," in *Proc. 29th Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2007, pp. 489–498.
[75] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the location and number of faults in large software systems," *IEEE Trans. Softw. Eng.*, vol. 31, no. 4, pp. 340–355, Apr. 2005.
[76] A. E. Hassan and R. C. Holt, "The top ten list: Dynamic fault prediction," in *Proc. 21st IEEE Int. Conf. Softw. Maintenance (ICSM)*, Piscataway, NJ, USA: IEEE Press, 2005, pp. 263–272.

Kevin Jesse received the B.S. degree in computer engineering and computer science from the University of California, Santa Cruz, CA, USA, and the M.S. degree in computer science from the University of California Davis, Davis, CA, USA, where he is currently working toward the Ph.D. degree. His research includes the areas of machine learning, natural language processing, and software engineering.

Christoph Kuhmuench received the Ph.D. degree in computer science from the University of Mannheim, Germany and has published extensively. He is a Software Architect with Siemens and holds several patents.

Anand Sawant received the Ph.D. degree from the Delft University of Technology. He is a Research Scientist with Endor.ai and previously worked in software engineering research with Siemens. He served as Committee Member in the Artifacts-track of ESEC/FSE 2020 and 2021. He was Online Co-Chair organizing committee for ESEC/FSE 2021.