Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Deep learning with class-level abstract syntax tree and code histories for detecting code modification requirements[☆]

O.O. Büyük[a], A. Nizam[b,*]

[a] Department of Computer Engineering, Fatih Sultan Mehmet Vakif University, Turkey
[b] Department of Software Engineering, Fatih Sultan Mehmet Vakif University, Turkey

ARTICLE INFO

Article history:
Received 7 January 2022
Received in revised form 17 May 2023
Accepted 8 September 2023
Available online 15 September 2023

Keywords:
Refactoring
Code smell
Recurrent neural network
Abstract syntax tree
Code embedding

ABSTRACT

Improving code quality is one of the most significant issues in the software industry. Deep learning is an emerging area of research for detecting code smells and addressing refactoring requirements. The aim of this study is to develop a deep learning-based system for code modification analysis to predict the locations and types of code modifications, while significantly reducing the need for manual labeling. We created an experimental dataset by collecting historical code data from open-source project repositories on the Internet. We introduce a novel class-level abstract syntax tree-based code embedding method for code analysis. A recurrent neural network was employed to effectively identify code modification requirements. Our system achieves an average accuracy of approximately 83% across different repositories and 86% for the entire dataset. These findings indicate that our system provides higher performance than the method-based and text-based code embedding approaches. In addition, we performed a comparative analysis with a static code analysis tool to justify the readiness of the proposed model for deployment. The correlation coefficient between the outputs demonstrates a significant correlation of 67%. Consequently, this research highlights that the deep learning-based analysis of code histories empowers software teams in identifying potential code modification requirements.

1. Introduction

The improvement in code quality is one of the main contributors to successful software systems. A significant amount of source code makes it difficult to ensure and control the code quality of GitHub like huge code repositories (Xue et al., 2019). Code smell detection and refactoring are important methods for providing and controlling quality attributes (Lacerda et al., 2020). Code quality issues, defined as code smells, suggest or reveal the possibility of refactoring (Martin, 2018). Refactoring is defined as "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior" (Martin, 2018). Determining the type and location of refactoring is a major challenge in software engineering research (Martin, 2018; Mens and Tourwé, 2004; Lacerda et al., 2020). Similarly, bug localization that automates the task of locating the potential buggy files, supports developers to focus on crucial files and allocate their resources more efficiently (Lam et al., 2017).

Manually identifying refactoring opportunities in the source code is a time-consuming process (Kurbatova et al., 2020). Numerous machine learning-based models have recently been developed to detect software quality problems (Shi et al., 2020). However, they require significant preprocessing and substantial effort to extract features from the source code and create barriers to their practical adaptations and applications (Sharma et al., 2021). Furthermore, existing tools do not consider the context that plays an important role in deciding whether a reported smell is an important quality issue. In addition, missing values can reduce the detection performance of (traditional) machine learning classifiers (Alazba and Aljamaan, 2021). Thus, the application of machine learning algorithms requires further research to facilitate code smell detection (Al-Shaaby et al., 2020) and refactoring opportunities (Alenezi et al., 2020).

Deep neural network (DNN) based code analysis techniques demonstrate better performance than handcrafted features (Shi et al., 2020). These techniques require effective code representations and large training datasets based on annotated codes. Recently, the success of Natural Language Processing (NLP) based embedding techniques have created new opportunities to develop effective code representations, called code embedding; thereby, low-dimensional vector representations of codes support

☆ Editor: Burak Turhan.
* Corresponding author.
E-mail address: ali.nizam@fsm.edu.tr (A. Nizam).

the application of existing machine-learning techniques to various code analysis tasks (Sui et al., 2020). Code embedding has been applied to solve many problems, such as method naming (Alon et al., 2019), bug localization (Liang et al., 2019), and legacy code migration (Chen et al., 2018). However, existing approaches have the limitations of intraprocedural embedding (Sui et al., 2020). Almost all existing embedding techniques are intraprocedural (Alon et al., 2019; Chen et al., 2018). This prevents the preservation of data flow across methods and isolates the analyzed code from the context. In addition, DNN analysis requires tedious efforts to collect huge amounts of refactoring data and map them to code snippets, such as classes and methods (Kádár et al., 2016). Experts have to verify a large sample of changes to define their type in the large modern defect datasets (Da Costa et al., 2017).

History-based smell detection methods that involve collecting and employing source code evolution information (Palomba et al., 2015; Fu and Shen, 2015) create an alternative solution for gathering labeled data. Learning from developers' past refactoring to create future refactoring recommendations establishes a new research domain (Bavota et al., 2015). Such methods extract structural information for different code versions, reflecting changes in the code development process (Sharma et al., 2021). Historical data contains the details of refactoring that existing studies on refactoring practices have not investigated; thus, analyzing the commit history of a project helps to infer rules for automatic detection to develop refactoring recommendation systems, identify the root cause and develop decisions behind specific refactoring types (Silva et al., 2016). In addition, it can be useful to understand how source code changes over time instead of using source code snapshots that demonstrate only instantaneous source code metrics and other information (Palomba et al., 2015). Oliveira et al. (2019) claimed that most developers evaluate the refactoring requirements using their experience instead of following refactoring recommendations in papers, books, and sites.

This study aims to detect source code modification requirements for a specific version of a class by designing and implementing a deep learning system based on historical code data and a class-level abstract syntax tree (AST) based vectorization. To the best of our knowledge, this study is the first to analyze historical code using deep-learning techniques to detect code modification requirements without derived features. Some studies have evaluated only earlier and later versions of modified source files (Sivaraman et al., 2022); however, we employed multiple versions of the same file in the learning and testing processes. The proposed system supports solving the problem of detecting the code modification type and location to improve the code quality for software teams using their past refactoring actions.

The inputs to our model are a code snippet and a corresponding refactoring tag obtained from the code history. We have developed a class-based vector representation that includes the class definition as an additional level of the AST to remove the limitations of intraprocedural embedding and strengthen the relationship between the refactoring location and code content. The AST of the entire class provides to exploration of the paths between methods and class-level variables to create a better code representation. The recurrent neural network (RNN) is designed to create a probabilistic model for identifying the code modification type required in a class. The output of the model is the code modification type for a class version that is used to predict future refactoring locations and types. Thus, if the entire code repository is represented as an input, the proposed system can detect the locations where and what type of refactoring is required. We demonstrated the effectiveness of our approach using three Java project repositories on GitHub as an experimental dataset. We compared our solution with similar vectorization solutions, such as Code2Vec (Alon et al., 2019), a variation of Code2Vec (Kurbatova et al., 2020), and text-based Word2Vec. A classic machine learning (Naïve Bayes classifier) technique is also used to evaluate the effectiveness of the deep learning-based system after the vectorization phase. To assess the readiness of our approach for deployment, we conducted a correlation analysis between the output of the proposed model and the results obtained from the metric-based PMD tool on our code-history dataset.

Our study differs from existing code analysis methods and makes the *following contributions*: (1) A new code embedding method (Class2Vector or abbreviated Class2Vec) to better represent source code integrity by adding the class level to the AST. To address the significant challenge posed by the massive size of class-level AST, we designed a DNN configuration using a cache model that stores subtrees and manages their connections to class-level AST. (2) Automatic detection of code modification requirements in the source code using a deep learning model with historical code data. (3) We created an experimental and large code-refactoring dataset that includes code and the type of change in code directly collected from changes in version histories of GitHub repositories. Based on these contributions, we have defined our research questions as follows:

RQ1: Is the proposed approach using deep learning methods with code history data capable of accurately detecting the code modification requirements in a software project? To answer this question, we investigated the effectiveness of our model in detecting refactoring locations and types. We compared our model with similar vector-based DNN approaches

RQ2: Is the proposed approach capable of accurately detect code modification requirements between different projects? Therefore, we aimed the generalization of the results by applying our model to the combined dataset from various repositories because changes in a single repository are heavily influenced by the coding style of a software development team.

RQ3: How does class-level vectorization improve the overall performance of detecting code modification requirements with a DNN? This question focuses on comparing our approach to other approaches based on word-based, intraprocedural vectorization and the naïve Bayes model. We used the Student's t-test to check for any significant differences to confirm the hypothesis of this question.

The rest of this paper is organized as follows. Section 2 introduces related work from the software literature. Section 3 presents our methodology for detecting refactoring points in the code. Section 4 presents the result obtained using DNN on the dataset. Section 5 answers the research questions and discusses threats to validity. Finally, Section 6 presents conclusions, implications, and future work on deep learning and refactoring.

2. Related work

In this section, we present the background and current state of code analysis by comparing several existing approaches. In addition, we investigate applications of deep learning to understand how DNN-based models have been applied to problems of code quality, code modification, and especially refactoring point detection problems.

2.1. Code analysis, refactoring, and code smell detection techniques

The code analysis and smell detection methods in software engineering literature are as follows:

- *Metric-based methods* compute source code metrics and apply a threshold to detect defects (Bigonha et al., 2019) or predict defects (He et al., 2015). Code smells reflect inappropriate design qualities in object-oriented systems, such as method cohesion and complexity, correct placement of methods in classes, and polymorphic problems, such as long method, feature envy, and state checking (Tsantalis and Chatzigeorgiou, 2011). A combination of software metrics constructs metrics-based rules to quantify defects; however, identifying the best threshold values is difficult for these metrics (Maddeh et al., 2021).

- *Rule-based*: Define rules to detect code smells using source code metrics and the decision tree method to formalize design defects (Sharma and Spinellis, 2018) and code smells (Moha et al., 2010). Domain experts need to manually generate certain rules to define each code smell (Alazba and Aljamaan, 2021).
- *History-based* methods utilize information on source code evolution (Fu and Shen, 2015). Information regarding past code changes related to code smells constitutes a reliable source for prioritizing refactoring suggestions (Tsantalis and Chatzigeorgiou, 2011). Such methods can be used to extract information regarding code structures and changes in versions to feed a detection model that infers smells in the code (Kádár et al., 2016). CCFINDER (code clone detection tool) was used to detect each refactoring candidate (Weißgerber and Diehl, 2006).
- *Machine learning* utilizes the code smell detection problem that requires data selection, formalization of the input and output, and algorithms that link prior knowledge to data to analyze the statistical properties of the code context (Allamanis et al., 2018). Many techniques have been employed in this area including neural network (Alazba and Aljamaan, 2021), Bayesian belief network (Khomh et al., 2009), and binary logistic regression (Bryton et al., 2010). The machine learning process includes the following steps: collecting a large repository; extracting metrics from systems at the class, method, package, and project levels; determining advisors for their detection; applying the chosen advisors on the systems, and manually labeling candidate smell for each class and method; and training and testing supervised classifiers (Arcelli Fontana et al., 2016).
- *Deep learning* automatically creates abstract features without specific metric-set specifications (Sharma et al., 2021). These methods such as Convolutional Neural Network and RNN methods are used to create code embedding (Alon et al., 2019), source code generation (Le et al., 2014), software vulnerability detection (Zhuang et al., 2019), and smell detection (Sharma et al., 2021; Liu et al., 2018).

Code-smell detection methods are largely based on the investigation of their negative side effects on software development activities, violation of object-oriented design principles, and granularity-based smell classification based on source code representation (Sharma and Spinellis, 2018). Code refactoring methods can be used to detect and eliminate code smells; conversely, code smell detection techniques identify where refactoring should be applied by developers (Martin, 2018). They allow developers to detect code fragments that should be restructured to improve the quality of the system (Vidal et al., 2016; Arcelli Fontana et al., 2016) and decide when to start and stop the refactoring mechanism (Martin, 2018).

The code modification process can be studied at different levels, such as class, method, block, and code line in the literature. There are many studies on identifying or predicting class-level code modification requirements. Different algorithms were applied to extracted software metrics computed from object-oriented software systems for refactoring prediction at the class, level such as ensemble learning (Panigrahi et al., 2022), a GRU-based system (Alenezi et al., 2020), string alignment algorithms (Moghadam et al., 2021), and support vector machines with optimization algorithms (Akour et al., 2022). Thus, developing deep learning systems that detect code modification requirements at the class level is an important research area.

2.2. Source code representation models

Code embedding aims to represent code semantics through distributed vector representations for source code analysis (Sui et al., 2020). The meanings of an element are distributed across multiple vector components while mapping similar objects to close vectors to assign paths and path elements to their corresponding real-valued vector representations or embeddings (Alon et al., 2019). Vectors support a variety of program analysis tasks such as code summarization and semantic labeling (Sui et al., 2020).

Many previous efforts in code embedding using Word2Vec (Mikolov et al., 2013) have treated the source code as textual tokens (Sui et al., 2020). Word2Vec uses hierarchical SoftMax in which the vocabulary is represented as a Huffman binary tree that assigns short binary codes to frequent words to reduce the number of evaluated output units and the complexity (Mikolov et al., 2013). It is possible to train high-quality word vectors using Word2Vec (Mikolov et al., 2013); however, software source code is different from natural language (Shi et al., 2020). Code written in a general-purpose programming language is a relatively new research area for existing machine learning and NLP techniques; code is executable and has formal syntax and semantics (Allamanis et al., 2018). The semantic structure of code and software programs is a collection of structured combinations of reserved words and identifiers (Shi et al., 2020).

Code embedding is based on graph embedding that converts a graph into a low-dimensional space while preserving the graph information (Sharma and Spinellis, 2018). The different semantic units in the AST represent the complex structure of the source code (Wang et al., 2020). After decomposing the code components into a collection of AST paths, the neural network learns the atomic representation of each path and the aggregation patterns between a set of paths (Alon et al., 2019).

There are many code representation models, such as Code2Vec (Alon et al., 2019), PathPair2Vec (Shi et al., 2020), and Code2seq (Tsantalis and Chatzigeorgiou, 2011). Code2Vec converts a code snippet into a single fixed-length code vector to predict semantic properties (Alon et al., 2019). It was used to predict the name of an unobserved method from the vector representation of its body after training a DNN with known method names (Alon et al., 2019). A variant of Code2Vec, which we call averaged-Code2Vec, uses a weighted average of all combined context vectors to recommend the move method refactoring by finding similarities between vector representations (Kurbatova et al., 2020).

PathPair2Vec (Tsantalis and Chatzigeorgiou, 2011) proposes a short-path concept that extracts short-path pairs between a terminal node and an internal node by constraining the length and span of the extracted path pairs. It uses two separate AST levels: one is based on the AST of the entire software program, and the other represents program's methods using sub-ASTs. The effectiveness of PathPair2Vec was evaluated using a software defect dataset to predict defective files.

Code2seq follows an encoder-decoder architecture. The encoder converts a given code snippet to a set of compositional paths over its AST, where the decoder compresses each path to a fixed-length vector using long short-term memory (LSTM) system. It assigns different weighted averages to the path vectors to produce output tokens to identify methods having different sequential representations but the same functionality. Code2seq applied code summarization to predict a method's name from its body and code captioning to predict a natural language sentence describing code snippets (Yahav and Levy, 2019).

Existing studies on code embedding focus on subparts such as methods or procedures (Sui et al., 2020). This prevents the complete representation of the refactoring context. In addition, there is a lack of a labeled dataset that supports research on the quality improvement or refactoring point detection of source code.

2.3. Deep learning and code analysis

Deep learning is a subfield of machine learning composed of multiple processing layers to learn data representations by decomposing and describing complicated mappings into different layers (Goodfellow et al., 2016). Mainstream tools use rules, metrics, histories, or machine learning approaches to detect smells (Sharma and Spinellis, 2018); whereas, deep learning research on code evaluation provides important advantages.

The main advantages of deep learning over traditional machine learning methods for code analysis are reduced preprocessing requirements and learning directly from the context. DNN techniques require less human labor and show better accuracy than traditional methods because their abstract high-dimensional features provide them with better representation capabilities (Liang et al., 2019). It has been suggested that the deep learning-based automatic selection of textual and other features significantly improves accuracy and outperforms well-known methods for detecting code smells (Liu et al., 2018). Thus, DNN can detect smells using only the code context without specific feature set specifications, such as metrics and rules (Sharma et al., 2021).

Code embedding-based deep learning solutions have generally been implemented using RNN. RNN learning algorithms store representations of recent input events in the form of activations of feedback connections (Hochreiter and Schmidhuber, 1997). RNNs can learn to retain only relevant information and forget others to make predictions by evaluating the importance of information. The textual processing ability of DNN reveals the opportunities to evaluate code histories to locate and detect code modification requirements. Historical code changes reflect human intuition, which is the fundamental source for defining refactoring points, and precise criteria and metrics where or when refactoring is overdue (Martin, 2018). Thus, previous changes in code can inform the developer to identify which code smells should be fixed (Vidal et al., 2016).

Numerous training data are required for a DNN system to learn the context. However, a large training dataset for training models on a large scale is not available and researchers have used training data generated by existing tools as an initial step to assess the feasibility of deep learning techniques for smell detection (Sharma et al., 2021). In addition, refactoring point detection requires the detection of a correspondence between the entire content of the changed method and a label for the reason behind the change. However, programming language datasets suffer from scarce annotations, owing to the time and expertise required to label them (Jain et al., 2020). Collecting large amounts of refactoring data and mapping them to low-level source code elements, such as classes and methods require substantial effort (Kádár et al., 2016). To overcome this difficulty, synthetic auto-generated labels have been used for method naming (Alon et al., 2019; Yahav and Levy, 2019; Yonai et al., 2019) and bug detection (Ferenc et al., 2018; Pradel and Sen, 2018; Benton et al., 2019). However, synthetic code datasets suffer from duplication issues (Habchi et al., 2021) and biases (Shin et al., 2019) that degrade generalization. Moreover, auto-generated data do not cover the diverse program behaviors encountered in the real world (Jain et al., 2020).

3. Methodology

This section describes our research methods by providing an overview, explaining the data curation process, discussing the code vectorization method, and describing the architecture of the deep learning models. The analysis activities of refactoring are (1) identifying code modification requirements in code at the class level and (2) determining the type of code modification required for the identified location (Mens and Tourwé, 2004).

Fig. 1 provides a simplified overview of the proposed methodology and its phases. The inputs are the code snippet, corresponding change type, version, and subsequent version. Let C be the code snippet and U be the corresponding change type. Therefore, our model attempts to learn the change-type distribution conditioned on the code: $P(U|C)$. The main phases are:

(1) Building a dataset using GitHub annotation with the Git Desktop and Git History tools.

(2) Vectorize the source code using GRU on class-level AST.

(3) Evaluate the probability of code modification and detect the code modification type and location using LSTM which takes the code vector and code modification type as inputs.

3.1. Construct experimental datasets

We have created an experimental dataset that contained AST-code modification type label pairs to address the refactoring locations and types. To the best of our knowledge, no code-history-based dataset is currently available for code modifications. The existing datasets reflect analyzes based on an instantaneous situation of a code base in the topics of method naming (http://learnbigcode.github.io/.), the analysis result of the GitHub repository smell detection tools (Sharma et al., 2021), or classifying code by functionalities (Mou et al., 2016). Some refactoring databases contain labels, types, and information about code changes, not the whole code (Géron, 2019), only object-oriented metrics about refactoring (https://github.com/dspinellis/awesome-msr), or only focus on mapping changed lines to bugs (Williams and Spacco, 2008).

Previous works on the dataset creation process for code analysis are based on qualitative analysis and static analysis tools (Rahman et al., 2019), transforming likely correct code into likely incorrect code by injecting synthetic smell statements (Pradel and Sen, 2018; Borovits et al., 2022). Because we aim to examine the refactoring and code modification behavior of software developers while creating code versions, we preferred to collect only the actual code history data from GitHub repositories without any augmentation. GitHub repositories provide tracking changes to investigate large amounts of data and allow other researchers to reach the same code base for future work.

The repository selection criteria are the popularity, strategy of labeling code, and the number of contributions to the project because an adequate number of release versions and code modifications between two adjacent releases (Kádár et al., 2016) are required for the RNN to learn the code modification types for a class. In addition, the license models of the selected projects are legitimate for analysis without legal restrictions. We found 8 GitHub repositories that fulfilled the above criteria.

We collected the change type, source code, and version information to build the dataset. Commit message types are essential sources of change history revealing code changes along with many other aspects of software development (AlOmar et al., 2021), such as file change-proneness and bug-proneness (Xia et al., 2016). Internal or external documentation supports the understanding of how the developer's intent to refactor codes leads to adequate corrective actions (AlOmar et al., 2021). We first

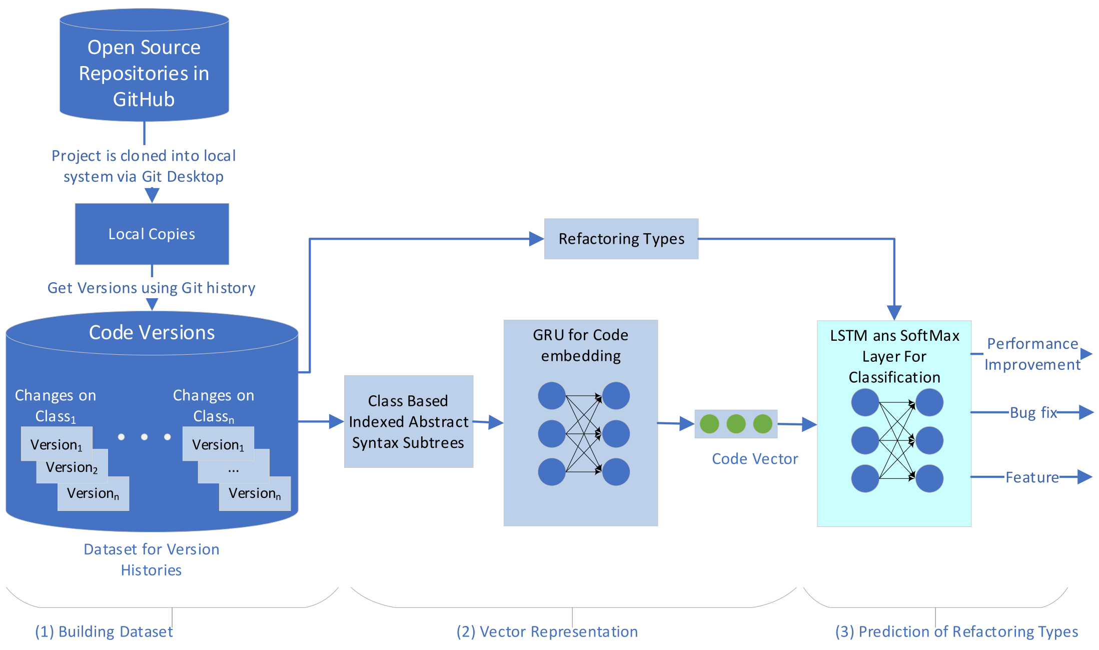

Open Source Repositories in GitHub
Project is cloned into local system via Git Desktop
Local Copies
Get Versions using Git history
Code Versions
Changes on Class1
Changes on Classn
Version1
Version2
Versionn
Dataset for Version Histories
Refactoring Types
Class Based Indexed Abstract Syntax Subtrees
GRU for Code embedding
Code Vector
LSTM ans SoftMax Layer For Classification
Performance Improvement
Bug fix
Feature
(1) Building Dataset
(2) Vector Representation
(3) Prediction of Refactoring Types

Fig. 1. The overall representation of our methodology.

identified the change type from the commit types of each repository and then related the lines that were modified in the corresponding action as the bug-fixing changes point to the locations of a bug (Xia et al., 2016).

A mapping schema is required for some repositories because they track the subtypes of code changes. Developing a mapping schema is a one-time-only operation that requires minimal manual effort. For example, sub-defect types of Hazelcast, such as *defect*, *performance defect*, and *fuzzy test defect*, were assigned to the most comprehensive term as *defect*. As a repository always uses the same commit type, a simple schema definition at the beginning is sufficient for the mapping process.

Generating training and evaluation data requires refactoring information and a path to the folder where the code fragments corresponding to the refactoring label are stored. We employed the GitHub Desktop tool to clone repositories into the local system to store and visualize the changes in each class. An open-source project called GitHistory was integrated with GitHub Desktop and customized to replace code changes in a structured folder tree.

Fig. 2 illustrates the storage structure that demonstrates the development history and relationship between the old and new versions of a class. All changes in a class are stored in the same folder. The version number and prefix label representing the changes in a class are appended to the end of the file name to audit the content of the developer's commit. The code modification types and their prefixes are *bugfix* (BF), *feature* (FE), and *performance enhancement* (PE). Version indices act as special identifiers for the AST of a class version.

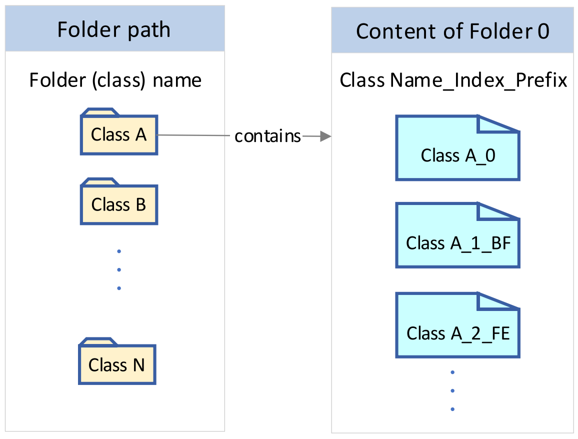


Fig. 2. Overview of the dataset.

3.2. Proposed code vectorization approach

The AST is converted into a single fixed-length code vector representing a code snippet to predict the semantic properties of the snippet (Allamanis et al., 2018). The vectorization creation process includes four steps: extracting paths from the AST, encoding the terminal node in the path, encoding the internal control node in the path, and combining path features into an overall representation of the source code (Allamanis et al., 2018).

The code is first decomposed into a collection of paths using its AST (Alon et al., 2019). The AST of method-level code snippet m is defined (Alon et al., 2019) as a tuple $\langle N, T, X, s\rangle$; it contains the sets of nonterminal nodes as N, terminal nodes as T, associated values as X, and $s \in N$ is the root node. An AST path of length k is a sequence of the form: $n_1 d_1 \ldots n_k d_k n_{n+1}$, where $n_1, n_{n+1} \in T$ are terminals, for $i \in [2..k]$: $ni \in N$ are nonterminals, and for $i \in [1..k]$ d_i is the up or down directions representing movement in the tree. All lists of children contain every node exactly once, whereas the root appears in more than one list.

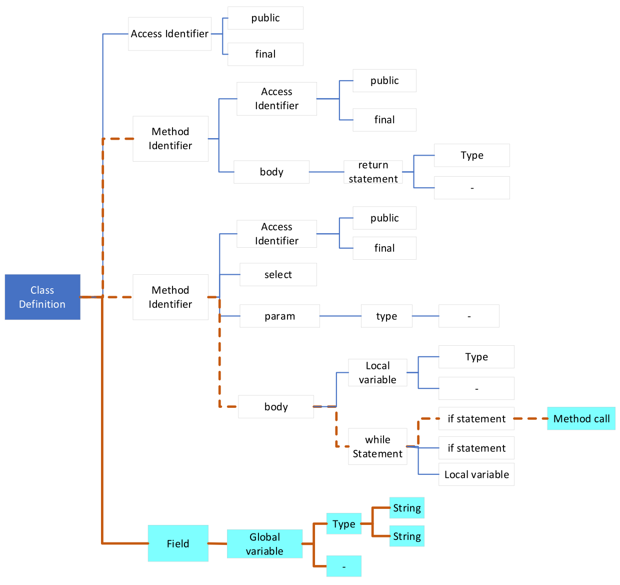
We prefer to use all path contexts of the AST in learning because it improves the performance of a neural network by identifying the importance of multiple path contexts and aggregating them (Alon et al., 2019). We add the class definition at the top level of our AST to represent the full path of the code to build a compact and comprehensive code representation instead of an intraprocedural AST definition. The class statement and method statement are defined as special statement nodes; therefore, Eq. (1) is formulated as

$$\langle S, (c, m)\rangle\ c : ClassStatement,\ m : Method\ Statement \quad (1)$$

The formal explanation extends the path definition by adding a class as a special terminal root node, n_c. The AST representation

```
class HasAttributeNodeSelector implements NodeSelector {
  private final String key;
  private final String value;
  public void getValue() {
     return value;
  }
  public void select(Iterable<Node> nodes) {
     Iterator<Node> itr = nodes.iterator();
     while (itr.hasNext()) {
        Map<String, List<String>> attributes =
itr.next().getAttributes();
        if (allAttributes == null) continue;
        List<String> values = attributes.get(key);
        if (values == null || false == values.contains(getValue())) {
           itr.remove();
        }
     }
  }
```

a) Java code fragment and its class level AST



b) Representation AST of a Class

Fig. 3. Class and Its AST format.

is constructed using the statements in the class and methods as formulated in Eq. (2).

$$n_c \times p_m = p_c \qquad : n_c \in s \qquad \therefore n_c \times AST_m = AST_c \tag{2}$$

Adding the class definition to AST allows for the creation of an effective code representation, including class-level parameters and method calls inside the class boundaries. This allows the DNN to learn from the entire class context not only its subparts. The class-level AST includes the additional path definition generated using class-level parameters and method transitions. This supports the extraction of substantial structural information regarding code, flow, and context-sensitive value flows to achieve improved results (Sui et al., 2020). The additional path definitions are as follows:

$$n_1^c d_1^c \ldots n_k d_k n_{n+1}, \text{ where } n_1^c d_1^c \text{ are class-level parameters} \tag{3}$$

$$n_1^{m1} d_1^{m1} \ldots n_k^{m2} d_k^{m2} n_{n+1}^{m2} \; where \; n_1^{m1} d_1^{m1} \text{ are method 1 and } d_k^{m2} n_{n+1}^{m2} \text{ method 2 parameters} \tag{4}$$

Fig. 3 shows a class and its representation in AST format. It contains the class definition, access identifier, method identifier, field, body, param, variable type, and statement label. Additional class-level paths are highlighted in different colors. Comments, Java documents, notations, and blank lines are not part of AST because the main focus is on the structure of the code and its changes.

3.3. Learning with RNN

To retrieve all symbols in ASTs as input for training, the pre-order traversal of the trees is implemented. Unsupervised vectors of symbols are learned using an approach extended by Code2Vec. The GRU performs encode and decode operations to obtain vector representation and code embedding. LSTM-based hidden layers take a single vector input representing indexed subtrees to learn features.

After the creation of ASTs, we employed the generative recurrent unit (GRU) that is a specialized version of RNN to encode and decode each subtree and representation index as a vector. RNNs can learn to retain only relevant information and forget others to make predictions by evaluating the importance of information. RNN builds a composition mechanism that can aggregate a bag of AST path contexts to generalize and represent code for analysis tasks (Alon et al., 2019).

Class-level vectorization creates a very large AST structure containing redundant or irrelevant information that requires a massive amount of system resources that could negatively impact the performance of the RNN. We have used subtrees of an AST rather than giving each time the entire AST as an input to decrease the system workload. An AST tree that represents a class includes a list of AST subtree nodes. A subtree can be one or more methods in a class and each element in the AST node binds to a node or subtree as shown in Fig. 4. Resolution binding in ASTs reveals the relationship between the overall structure of its content and leverages the naturalness of method statements (Zhang et al., 2019a).

A cache model is used in a gated recurrent unit (GRU) to store and access subtrees of an abstract syntax tree (AST) that is accessed multiple times. A cache model is a type of memory that stores frequently used data in a fast-access location, such as the GRU's memory, so that it can be retrieved quickly when needed. The node and its main reference are stacked in the GRU unit via embedding. Class-level vectorization holds information about each AST node binding with its *SimpleName* as an identifier which is known as an index that uniquely identifies the class of the source code. If an AST node and another AST node belong to the same class, each of them has a binding with this class index.

The first step involves the pre-trained embedding parameters which are represented in Eq. (3). For a given statement class tree T, let n represents a non-leaf node, C represents the number of its branched nodes, and W_e represents the encoded weight matrix.

$$W_e \in \mathrm{R}^{|V| x d} \tag{5}$$

The vector of the node n can be retrieved by Eq. (6) where V is the code vector size of the source code and d indicates the embedding dimension of symbols:

$$v_n = W_e^T x_n \tag{6}$$

where x_n is the one-hot representation of the statement type of symbol n and v_n denotes the code embedding. Next, the following

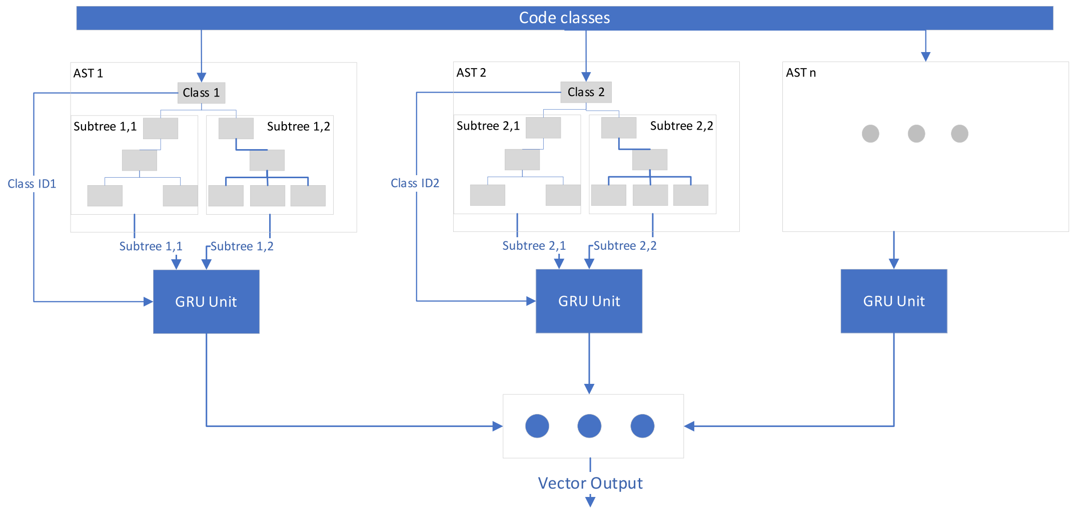


Fig. 4. Detailed representation of our vectorization model.

equations compute the vector representation of the node n as follows:

$$x_n = W_n^{\mathrm{T}} v_n \tag{7}$$

$$h_s = \sum_{i \in [1,C]} h_i \tag{8}$$

$$h = \sigma(w + h_s + b_n) \tag{9}$$

where $W_n \in R^{\mathrm{dxk}}$ represents the weight matrix and its multiplication with the vector presentation of node n reveals each vector's weight represented as w. The encoding dimension is k. The bias term is declared as b_n. The hidden state for each child in class and method statement is h_i where h denotes the updated hidden state. h_s is the sum of h. σ symbolizes the activation function. In addition, to determine the most important features of the node vectors, all nodes in the class and method statements are placed into a stack and then sampled by the encoder–decoder embeddings.

The GRU empowers the specification capability of the recurrent unit and obtains the dependency information. The hidden states in both directions of the unit are enumerated to form new states. The new form of the hidden states is defined by the following equations:

$$h_t^{\rightarrow} = GRU^{\rightarrow}(e_t), t \in [1, T] \tag{10}$$

$$h_t^{\leftarrow} = GRU^{\leftarrow}(e_t), t \in [T, 1] \tag{11}$$

$$h_t = [h_t^{\rightarrow}, h_t^{\leftarrow}], t \in [1, T] \tag{12}$$

The learnable parameters are matrices W_f, W_i, W_o, W_c and vectors b_f, b_i, b_o, b_c. t index corresponds to the tth character of the input sequence. The operator o represents the element-wise Hadamard product (Shi et al., 2020). The objective is to determine the values of w and b that minimize the output error of the loss function. The sequence of each statement in the RNN (Géron, 2019) can be obtained using the following equations:

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z) \quad \text{update gate} \tag{13}$$

$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r) \quad \text{reset gate} \tag{14}$$

$$\sim h_t = tanh(W_h x_t + r_t x(U_h h_{t-1}) + b_h) \quad \text{candidate state} \tag{15}$$

$$h_t = (1 - z_t)\, x h_{t-1} + z_t x \sim h_t \quad \text{current state} \tag{16}$$

where r_t is the reset gate to check the dependency of the previous state, the update gate is represented as z_t which provides a combination of the previous and new incoming information, and $\sim h_t$ is the upcoming state and is used to perform a linear interpolation together with the previous state h_{t-1} to determine the current state h_t. The symbols of W_r, W_z, W_h, U_r, U_z, $U_h \in R^{k \times m}$ define weight matrices, and b_r, b_z, b_h are bias terms. After checking the next iteration for computing hidden states of all time steps, the sequential representation of the statement can be retrieved (Zhang et al., 2019b)

The mathematical expressions for using subtrees of an AST as input to a gated recurrent unit (GRU) with a cache model are defined using $V_{AST} = \{v_1, v_2, \ldots, v_n\}$ as the input and h_t as the output where V_{AST} is a set of vectors representing the subtrees of the AST, and h_t is the hidden state output by the GRU at time step t. To process the input, the cache model first checks if the subtree represented by each vector v_i is stored in the cache. If the subtree is not in the cache, it is added to the cache, and its corresponding hidden state h_i is calculated using the GRU. If the subtree is already in the cache, its corresponding hidden state h_i is retrieved directly. The final hidden state h_t is then calculated using the hidden states of all the subtrees as inputs:

$$h_t = GRU(h_1, h_2, \ldots, h_n) \tag{17}$$

Algorithm 1 summarizes in pseudocode the process of using subtrees as inputs to the GRU with a cache model. The cache model is used to store and retrieve the hidden states of the subtrees that are accessed multiple times.

Algorithm 1 Subtree with cache model

```
1  for each subtree vector $v_i$ in $V_{AST}$ loop
2     CHECK if $v_i$ is in the cache.
3        If $v_i$ is not in the cache:
4           ADD $v_i$ to the cache.
5           CALCULATE $h_i = GRU(v_i)$.
6        If $v_i$ is in the cache:
7           RETRIEVE $h_i$ from the cache.
8  end for
9  CALCULATE $h_t = GRU(h_1, h_2, \ldots, h_n)$.
```

Trained embeddings are passed as initial parameters into the LSTM unit (Dey and Salemt, 2017). The input of the recommended method is a tuple of class-based AST as $\langle AST_c \rangle$. This provides a performance advantage accomplished by in-memory storage of predefined embeddings and the use of caches for incoming test vectors. All leaf nodes of AST that represents the lexical information, such as expressions, are also combined into the leaf nodes of the statement class trees (Zhang et al., 2019b). The designed encoder traverses the tree and recursively receives the symbol of the current node with the hidden states of its branched nodes as a new input for the calculation.

We have adopted the algorithm in Code2Vec style embeddings and compared them with the result of the proposed method. The input of Code2Vec is a class label plus method-based AST as $\langle c, AST_m \rangle$. Averaged-Code2Vec creates a vector for each method using Code2Vec and averages them to represent the corresponding class (Kurbatova et al., 2020). The input of this method is $\langle c, AST_{avg} \rangle$.

In addition, to evaluate the difference between AST and text-based analyses of code, we used Word2Vec as the base model. The input of Word2Vec is $\langle c, WordVector \rangle$. The Word2Vec uses two different models Continuous Bag Of Words (CBOW) and Skip-gram (Mikolov et al., 2013). CBOW predicts the target terms from a given set of context terms and Skip-gram predicts the context terms from a target term. CBOW is faster to train than Skip-gram and slightly more accurate for frequent words (Mikolov et al., 2013) because the code contains repeated keywords. We chose the CBOW model for the code analysis.

As the proposed method enhances Code2Vec which uses deep learning in the code vectorization phase, it is difficult to make direct comparisons with classical machine learning methods. However, after vectorization, the Bernoulli Naïve Bayes classifier is used to compare deep learning models with normal machine learning. Bernoulli Naïve Bayes is implemented for discrete variables on the binary concept of whether the term occurs (Ullah et al., 2022). In this study, the feature vector represents the class. The target class label determines the target category.

We used stratified cross-validation to validate our machine learning classifier (Kohavi, 1995) by partitioning our dataset into 70% for training, 15% for validation, and 15% for testing sets. The validation step is used to improve the training performance and tune the parameters of the classifier to ensure that all samples in the dataset are used in training. We repeated the process five times using different folds and averaged the detection performance of the iterations to obtain the final estimate.

3.4. Static code analysis

We employed the PMD static analysis tool to identify the quality issues in our dataset for the purpose of conducting a comparative analysis. The PMD source code analyzer is designed to detect common programming flaws, such as unused variables, loose coupling, and unnecessary object creation (PMD, 2023). PMD is a highly cited tool due to its good overall performance (Lacerda et al., 2020). To evaluate the correlation between the output of PMD and Class2vec, we followed a systematic procedure as outlined below:

1. Perform an analysis on a particular class using its version history, which includes all the labeled changes made to that class. We excluded files that contain syntax errors, as they prevent the static code analysis process.
2. Filter the version history to include only the sections of code that have been refactored, focusing on performance enhancements, since the primary function of PMD tool is to detect code quality problems rather than bugs and future enhancements.
3. Compute the class vector and method sub-vectors for each version of the class, as they have already been stored.
4. Compare the method sub-vectors between consecutive versions of the class and determine whether any changes or additions represent performance enhancements in the methods.
5. Calculate the total number of times performance enhancements have been made to the class throughout its history by aggregating the number of method performance enhancements across all versions of the class.

We used the *correlation coefficient* term because the output of the quality analysis tool may not precisely capture or match intents of actions performed by the development team. The correlation coefficient is calculated as *(number of correctly identified performance enhancements)/ (total number of performance enhancements)*. This formula measures how effectively the proposed system can identify performance enhancements compared to the total number of performance enhancements provided by quality analysis tools.

4. Results

This section introduces the properties of our dataset and provides quantitative analysis results by comparing the actual output with the predicted output. We investigated the results of Class2Vec, Code2Vec, averaged-Code2Vec, Word2Vec, and naïve Bayes methods.

4.1. Dataset

We have selected open-source projects Elasticsearch, Hazelcast, Mockito, Apache Dubbo, Guice, Hystrix, Jenkins, and Netty as repositories to collect the source code and corresponding refactoring progress information. The refactoring-based changes are mined between releases, although the changes between releases are rather low. The dataset contains the fine-grained refactoring and source code of eight open-source Java systems and an average of 37 classes from each repository. We attempted to ensure a balanced distribution of the dataset by collecting a similar number of classes from each repository. The full set of code history data are available at https://github.com/FSMVU/CodeModificationHistories for further research. Table 1 summarizes the code modification labels and their numbers in each repository. The number of modifications and corresponding class versions vary for each repository.

Table 2 provides details about the projects, their names, the labeling strategies for code modification types, the number of classes, class version histories, and lines of codes. The dataset contains 1537 source code units and 425,448 code lines. We categorized and labeled the modified code by obtaining the developer's declarations in the git commits of projects.

The hyperparameters were determined using the random search method by experimenting with and evaluating different values to identify the best combination of accuracy, training time, and GPU memory. First, the bounds for creating hyperparameter sets were identified. Random values of the hyperparameter set were then tested, and the performance of the model was observed. The definitions and optimal values of the hyperparameters are presented in Table 3. For convenience, we use the same values for the compared models.

The network has four input layers to handle different repository features. The hidden layers are configured with a layer size of four, each with 25 neurons, and the total neuron size is 100. The output layer contains three output nodes for each code modification type. The ReLU activation function is used to train the network to converge more quickly and reliably (Parhi and Nowak, 2020). The optimization algorithm of AdaMax with the

Table 1
Change type list.

Repository	Label of modification type			Total
	Bug It is a flaw or fault in a software code.	Feature - Only newly added implementation. - This may be change requests as well.	Performance Making the existing mechanism robust	
Elasticsearch	75	46	41	162
Hazelcast	67	29	44	140
Mockito	43	34	39	116
Apache Dubbo	52	118	58	228
Guice	77	103	21	201
Hystrix	42	107	17	166
Jenkins	83	124	22	229
Netty	95	112	88	295

Table 2
The statistics for source code from the selected GitHub repositories.

Name of source project code	Used labeling mapping strategies	Base class numbers	Class version numbers	Number of code lines (excluding blank lines)
Elasticsearch	Bug, refactoring, enhancement, performance	50	162	24,991
Hazelcast	Defect (defect, performance defect, fuzzy test defect), enhancement, performance	50	140	23,361
Mockito	Bug, refactoring, enhancement (new feature)	50	116	24,738
Apache Dubbo	Bug, refactor (code-quality), enhancement (feature, proposal, faq, dependency-upgrade), performance	29	228	33,367
Guice	Bug, type-task (new feature), performance (type-code quality)	33	201	41,232
Hystrix	Bug, enhancement, performance	22	166	98,615
Jenkins	Bug (regression-fix, major-bug), rfe (enhancement, major-rfe, minor-rfe), performance (leak)	28	229	49,465
Netty	Bug (defect), feature, improvement (performance)	32	295	129,679
Total numbers		294	1537	425,448

Table 3
The definition of the hyperparameters and their values.

Type of hyperparameters	Value
Input layer	4
Hidden layer	4
Epoch	10
Size of mini-batch	64
Name of optimizer	Adamax (Learning Rate: 0.002, ß1 = 0.9, ß2 = 0.999)
Size of encoding	128
Activation function	ReLU
Neuron size	100
Dropout	2

defined values is advantageous for sparse parameters in code and word embeddings. The encoding of the code and word vector size is 128. The mini-batch size for obtaining a subset of data during an iteration is 64. The epoch value indicates that the model passes through the training set ten times. The dropout value provides randomly dropped units during training to reduce overfitting and improve generalization error (Srivastava et al., 2014).

4.2. Quantitative evaluation

In this section, we utilized standard statistical measures such as accuracy, precision, recall, and F-score to evaluate and compare the performance and effectiveness of different approaches. The classes from the repositories were categorized based on their change history and used for training, validating, and testing the model. The results presented here are the average of five repetitions for each dataset.

Fig. 5a–h shows the test accuracy over the number of iterations for each of the evaluated models. It can be observed that all tested models achieved their best results around the 40th iteration of the validation process. The overall saturation rate of each model exhibited similar characteristics. Naïve Bayes does not use epoch logic; therefore, a graphical representation is not presented visually.

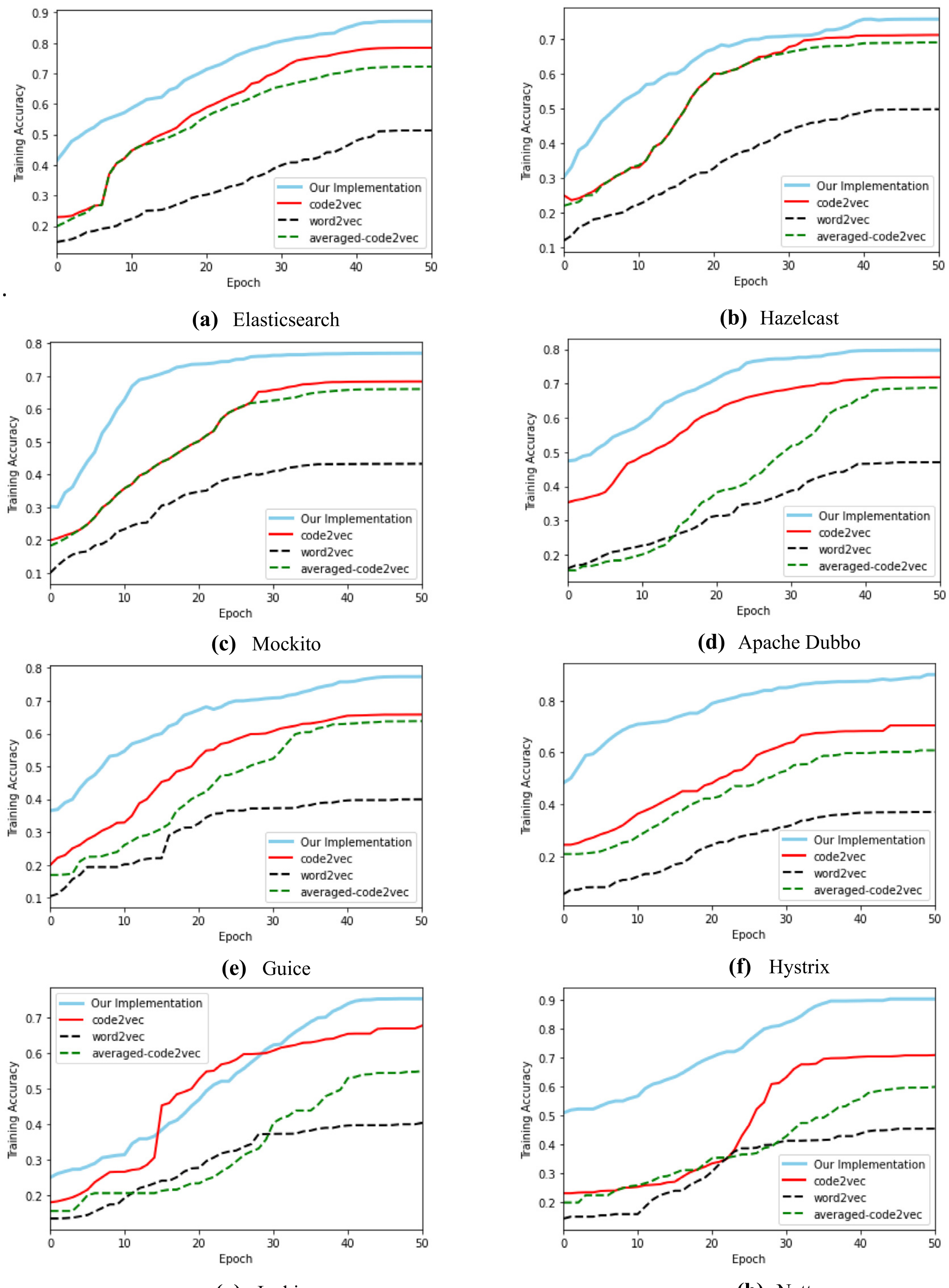
Table 4 presents confusion matrices of the combined dataset to provide more precise evidence of the performance of the proposed method (Ullah et al., 2020). Confusion matrices for each repository are provided in Appendix A Supplementary data, so as not to overextend the article.

Table 5 summarizes the correct and missed classification values to demonstrate the possibility of an imbalanced classification problem. We have summarized the $Success\,(S) = TN + TP$ and $Failure\,(F) = FN + FP$ to limit the size of the table.

Table 6 presents a summary of the results obtained from applying the proposed approach to each repository. The precision and recall metrics were calculated as a means of evaluating the classification accuracy of the approach. These metrics provide insight into the potential impact of false positives and false negatives on the predictions made by the approach.

Table 7 presents the correlation between the outputs of the static analysis tools and Class2Vec by showing the number of outputs of the static analysis tools accurately identified by Class2Vec for each repository. *The raw outputs of static analysis tools* is given in Appendix A Supplementary data. In addition, Appendix A Supplementary data also includes *the correlation coefficient between the output of Class2Vec and static analysis tools for each repository* and class version to allow a detailed investigation of researchers.

To eliminate the effect of the difference in the number of classes in the examined repositories, the normalized averages were calculated by multiplying the number of class versions in different repositories by the accuracy, precision, and recall values. The Class2Vec approach consistently provided better accuracy values than Code2Vec, averaged-Code2Vec, and Word2Vec on all datasets. The normalized accuracy of our model was improved



(a) Elasticsearch **(b)** Hazelcast **(c)** Mockito **(d)** Apache Dubbo **(e)** Guice **(f)** Hystrix **(g)** Jenkins **(h)** Netty

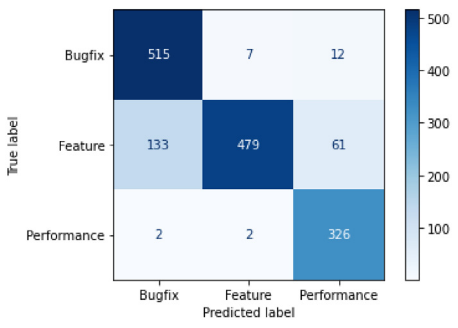
Fig. 5. Accuracy comparison of different repositories.

by approximately 12% for Code2Vec; 39% for Word2Vec; and 20% for averaged-Code2Vec. For the combined dataset, Class2Vec increased the average accuracy rate by 13% over Code2Vec, 37% over Word2Vec, and 17% over averaged-Code2Vec. The performance of the naïve Bayes classifier was too low to be considered in the evaluation. The Class2Vec, Code2Vec, and Word2Vec algorithms provided the same high recall values, indicating that they correctly identified most of the positive code modification types.

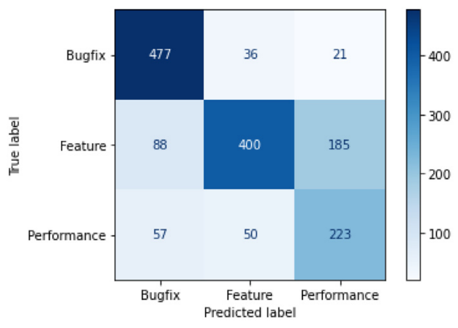
We investigated whether the AST definition of Class2Vec could also help detect code modification requirements for different repositories. For the collective evaluation, we combined all data into a single dataset and selected the training and test data as shown in Fig. 6.

We conducted Student's Paired t-test for significance testing to determine the improvement observed in the performance produced by different models (Kak, 2019; Akbar and Kak, 2019) as

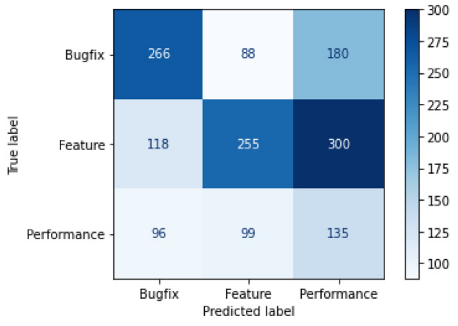
Table 4
Confusion matrices for the combined dataset.



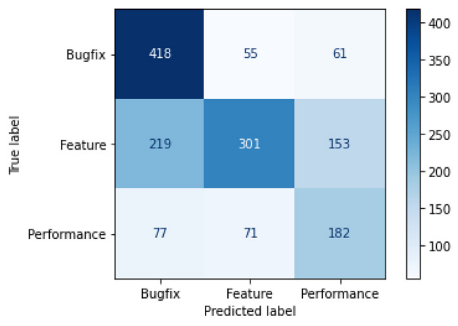
(a) Class2Vec



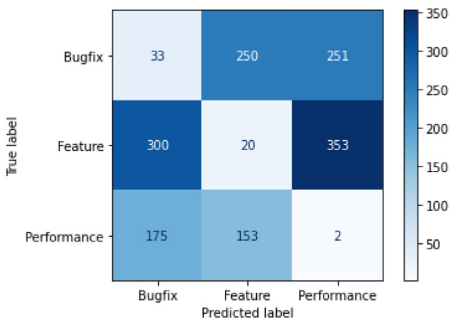
(b) Code2Vec



(c) Word2Vec



(d) Averaged-Code2Vec



(e) Naïve-Bayes

Table 5
Measuring success and failures for labels of different source code vector methods.

Type of label	Method	Name of dataset								
Success (S): N+TP, Failure (F): N+FP		Elasticsearch	Hazelcast	Mockito	Apache Dubbo	Guice	Hystrix	Jenkins	Netty	Combined
BugFix (S, F)	Class2Vec	96, 8	70, 14	40, 7	25, 7	20, 4	16, 5	20, 5	20, 4	20, 8
	Code2Vec	50, 12	42, 18	35, 14	20, 11	19, 8	15, 9	19, 7	39, 5	48, 21
	Word2Vec	46, 37	34, 24	19, 16	21, 16	19, 17	18, 3	10, 18	22, 25	19, 18
	Averaged-Code2Vec	51, 33	38, 26	28, 19	12, 8	17, 10	15, 9	30,21	36, 9	39, 19
	naïve-Bayes	11, 85	3, 59	3, 49	3, 24	5, 29	8, 29	18, 50	20, 44	10, 22
Feature (S, F)	Class2Vec	28, 7	30, 8	34, 7	150, 10	124, 3	122, 10	161, 20	220, 26	1440, 30
	Code2Vec	31, 8	37, 10	32, 10	120, 51	103, 26	97, 29	130, 30	160, 59	1299, 95
	Word2Vec	20, 14	21, 20	15, 28	60, 55	69, 51	63, 56	97, 60	110, 109	700, 712
	Averaged-Code2Vec	20, 7	26, 10	25, 15	110, 68	90, 33	70, 49	110, 45	164, 49	1220, 185
	naïve-Bayes	9, 20	3, 40	2, 34	20, 150	19, 92	30, 59	31, 109	80, 80	144, 1300
Performance Enh. (S, F)	Class2Vec	17, 6	12, 6	16, 12	29, 7	44, 6	9, 4	18, 5	20, 5	25, 14
	Code2Vec	46, 15	32, 7	11, 11	20, 6	30, 15	10, 6	30, 13	28, 4	55, 29
	Word2Vec	18, 27	14, 27	17, 21	41,35	25, 20	13, 13	22, 23	12, 17	44, 44
	Averaged-Code2Vec	46, 5	32, 8	23, 11	19, 11	30, 21	18, 5	19, 4	29, 8	45, 25
	naïve-Bayes	5, 22	4, 34	2, 26	4, 27	19, 37	4, 36	4, 16	15, 50	11, 50

Table 6
The summary of the results.

Method		Elastic-search	Hazelcast	Mockito	Apache Dubbo	Guice	Hystrix	Jenkins	Netty	Normalized averages	Combined dataset
Class2Vec	Accuracy	0.87	0.80	0.77	0.80	0.90	0.77	0.75	0.90	0.83	0.86
	Precision	0.93	0.90	0.89	0.93	0.91	0.84	0.86	0.91	0.90	0.90
	Recall	0.93	0.84	0.79	0.84	0.97	0.88	0.83	0.96	0.89	0.95
	f-score	0,93	0,87	0,84	0,88	0,94	0,86	0,84	0,93	0,89	0,92
Code2Vec	Accuracy	0.79	0.72	0.69	0.72	0.70	0.66	0.68	0.71	0.71	0.73
	Precision	0.85	0.81	0.85	0.74	0.63	0.73	0.74	0.73	0.75	0.76
	Recall	0.86	0.83	0.72	0.71	0.93	0.82	0.81	0.85	0.82	0.94
	f-score	0,85	0,82	0,78	0,72	0,75	0,77	0,77	0,79	0,78	0,84
Word2Vec	Accuracy	0.52	0.49	0.43	0.47	0.37	0.40	0.40	0.46	0.44	0.49
	Precision	0.70	0.54	0.50	0.35	0.40	0.37	0.60	0.45	0.48	0.47
	Recall	0.63	0.55	0.54	0.56	0.38	0.74	0.44	0.67	0.56	0.95
	f-score	0,66	0,54	0,52	0,43	0,39	0,49	0,51	0,54	0,52	0,63
Averaged-Code2Vec	Accuracy	0.72	0.69	0.66	0.68	0.60	0.64	0.55	0.60	0.63	0.69
	Precision	0.63	0.45	0.46	0.71	0.70	0.59	0.73	0.61	0.63	0.74
	Recall	0.45	0.50	0.51	0.72	0.67	0.85	0.56	0.71	0.64	0.84
	f-score	0,53	0,47	0,48	0,71	0,68	0,70	0,63	0,66	0,63	0,79
Naïve Bayes	Accuracy	0.15	0.05	0.06	0.08	0.09	0.07	0.11	0.08	0.09	0.10
	Precision	0.37	0.04	0.06	0.05	0.07	0.09	0.10	0.06	0.10	0.06
	Recall	0.08	0.05	0.05	0.10	0.09	0.07	0.10	0.13	0.09	0.11
	f-score	0,13	0,04	0,05	0,07	0,08	0,08	0,10	0,08	0,10	0,08

Table 7
The correlation coefficient between the output of Class2Vec and static analysis tools.

	ApacheDubbo	Elasticsearch	Guice	Hazelcast	Hystrix	Jenkins	Mockito	Netty	Total
The correlation coefficient	66.00	52.78	59.36	65.18	77.64	78.46	70.24	81.63	67.35

Table 8
The correlation coefficients between the output of Class2Vec and the PMD tool.

	Code2Vec		Averaged-Code2Vec		Word2Vec	
	t-value	*p*-value	*t*-value	*p*-value	*t*-value	*p*-value
Class2Vec	4.91	0.000078	6.6	<.00001	14.62212	<.00001

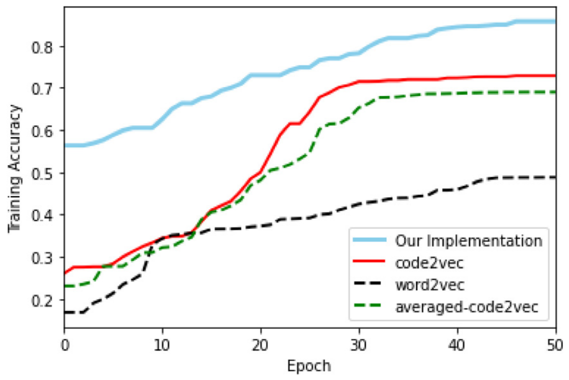


Fig. 6. Accuracy comparison on the combined dataset.

listed in Table 8. These results are significant at p < .05. Therefore, we accept the hypothesis that class-level AST usage attains higher scores than other DNN-Based vectorization methods for code modification types and locations at the class level.

We found low correlations between accuracy and the number of classes, class versions, lines of code, and average line of code in methods as 0.11, 0.41, 0.24, and 0,09 respectively. This suggests that repository structure is more important than its size in terms of accuracy.

From the performance perspective, the training times for the Class2Vec, Code2Vec, Word2Vec, averaged-Code2Vec, and Naïve Bayes methods were approximately 108 h, 96 h, 36 h, 78 h, and 2 h respectively. Class2Vec uses a complex AST structure that increases its runtime duration. However, considering the increase in accuracy, a slightly longer duration compared to other vector-based methods is acceptable.

5. Discussion

In this section, we address the implications of our work in relation to with the research questions and compare our findings with those of previous studies.

5.1. Answering the research questions

The evaluation of the results reveals the answers to the research questions and the applicability of the proposed approach.

RQ1: Is the proposed approach using deep learning methods with code history data capable of accurately detecting the code modification requirements in a software project?

Answer 1: Our results suggest that using historical information in the deep learning-based system can identify the refactoring location at the class level with a high accuracy of 80% on average for the same repository. These results provide a new perspective on previous studies by indicating that refactoring activities can be estimated using previous refactoring activities.

RQ2: Is the proposed approach capable of accurately detecting code modification requirements between different projects?

Answer 2: The accuracy of the combined dataset is slightly lower (2.25%) than that of the repository-based calculation. Such performance loss is a typical result of combining repositories with different structures. However, the overall results obtained on the combined dataset confirm that we can generalize our refactoring detection approach to different projects.

RQ3: How does class-level vectorization improve the overall performance of detecting code modification requirements with DNN?

Answer 3: The unique and combined dataset results reveal that adding the class definition to the AST vectorization process increases the detection performance of the deep learning-based system. It can be observed that a comparison of our approach to Word2Vec and Code2Vec over the same dataset demonstrates an improvement over Code2Vec ($>$11%), averaged-Code2Vec (18%), and Word2Vec ($>$35%). The improvement compared to Word2Vec is an expected outcome due to its inability to define software code-specific constructs. The results demonstrate the ability of the Class2Vec method to provide improvements over other vectorization approaches even for combined datasets also.

Class2Vec provided the best accuracy results for bugfix and performance refactoring, according to the values listed in Table 5. Considering the combined dataset, the bugfix accuracy rate was approximately 3.5 times better than Code2Vec, 5 times better than averaged-Code2Vec, and 4.3 times better than Word2Vec. The main reason behind these accuracy differences is that these code modification types cause a code change that produces a similar AST. However, when new code was added to the refactoring process (adding features), Code2Vec and its variant exhibited the best accuracy and outperformed Class2Vec. Word2Vec provided lower accuracy in all aspects compared to the other methods except naïve Bayes. The deep-learning-based methods exhibited similar saturation characteristics as shown in Figs. 5 and 6. The accuracy of the naive Bayes algorithm was very low for code vectors.

In-depth examination of the study results showed that when using small lines of code (fewer than 100 lines per class), all methods except for word2vec and Naive Bayes performed well. However, as the size of the class increased and the changes in the code structure were not clearly visible, the performance scores for accuracy decreased for all methods. Despite this decrease in accuracy, Class2vec still performed the best, especially in cases where the class size was large (500 lines of code or more) compared to other methods. When comparing the accuracy, as shown in Fig. 5, it can be seen that Class2vec had the best learning results, particularly when more lines of code were trained, as seen in Table 2. This was due in part to the development of Netty, which had more meaningful labels and changes that affected the code vectorization structure. In addition, the success of the classifier can be attributed to the quality and diversity of the training data, which were well matched, optimized, and tested on various repositories.

Code-based classifiers provide high accuracy in code refactoring analysis. Improving the performance of Code2Vec implies that class-based AST provides a better code representation than the method-based representation. In addition, the performance improvement of Class2Vec against Code2Vec for the combined dataset is slightly higher than the average improvements of the unique datasets. This indicates that class-level AST promises better discrimination capabilities for large and combined codebases in deep learning studies.

Based on the comparative analysis results presented in Table 7, it can be observed that approximately 67% of the classes demonstrated a significant correlation between the recommended system output and the code changes suggested by the analysis tool. However, approximately 25% of code quality issues did not align with the results generated by the proposed system. This finding highlights the need for further investigation and research in this area.

The results of our study are comparable to the existing metric-based tools in the literature. RefDetect and RMiner achieved f-scores of 92.1% and 86.9%, respectively, for class-level refactorings on a code history dataset containing 514 commits from 185 Java applications, and 5058 true refactoring instances (Atwi et al., 2021). Although it is not possible to directly compare the results, as the dataset used in that work was not accessible, we obtained a high f-score of 92.43% for the combined dataset without using any predetermined metrics.

5.2. Threat to validity

Although the experiments demonstrated improvements provided by our approach, there are still some potential threats to the validity of our study.

Determining the optimal hyperparameters for different models is a difficult process for achieving *internal validity*. These parameters were calibrated by computing the accuracy of each model using different parameter values to select the best combination before comparing models. In addition, we observed the activation function outputs of each algorithm over iterations until the saturation point was reached, to select an appropriate iteration number.

Our study found that using a limited set of software projects from GitHub resulted in an accuracy of approximately 80%. While our method allows for the examination of a larger codebase with DNN, license restrictions have slowed the collection of historical data from GitHub, resulting in a relatively small sample size. Recently, some open-source projects have converted their fully usable code (with Apache v2 licenses) to paid features. Generally, increasing the training data improves the accuracy of DNN systems, but as more code is transformed into AST and stored in memory, our model may encounter overflow issues. To address this, we will need to optimize the model for larger datasets or run garbage collection tasks on stored units

For external validity, the several factors may limit the generalization of our results. The first factor is the representativeness of our dataset which was built by sampling classes written in Java using the experiment conducted over eight applications. This makes it difficult to generalize our results to proprietary software or non-Java software.

In this study, we attempt to identify class-level code modifications. The class may seem like a huge structure to investigate the modification location; however, determining which class needed what kind of modfication is very important in a project with thousands of classes. In addition, many studies investigated class-level refactoring requirements in the literature (Panigrahi et al., 2022; Alenezi et al., 2020; Moghadam et al., 2021; Akour et al., 2022). After identifying the class, the development team further investigates the code to address quality problems.

The proposed model using source code metrics has better performance than other DNN vectorization techniques at the class level. However, the line-based and block measurement capability of existing code smell detection tools provides them an advantage over DNN systems in real-world applications. The comparative analysis results with the PMD tool also indicate that the proposed system has a relatively low success rate in identifying the recommendation of static code analysis tools. Thus, we believe that the recommended class-level system can be used as a supporting system for existing code smell tools rather than as a direct alternative to them. It creates a potential to integrate deep learning techniques into static code analysis tools or develop code quality control tools directly based on deep learning.

The accuracy and code smell detection capability of the PMD system used in this study is reported to be relatively low (Lacerda et al., 2020). The results of the proposed system can be compared with those other code quality analysis tools, such as Sonarqube and Designite for a more comprehensive evaluation. These tools often require the compilation of entire source code to detect errors and analyze code quality. However, during our attempts to compile the entire GitHub repository for each file version, we encountered numerous errors. While this posed challenges in conducting a more comprehensive analysis, we believe that addressing these errors and performing a thorough evaluation would provide valuable insights for future studies.

The other factor that influences the sampling procedure is the selection of code modification types when building the datasets. Our AST-based method can address any refactoring location in the code and can be implemented easily to detect the locations of different code modification types. However, the type selection involves uncertainty because refactoring is interleaved with other development tasks, such as updating functionalities and bug fixes (AlOmar et al., 2021), whereas bug fixes necessitate refactoring actions (Pantiuchina et al., 2020). To generalize our results, the recommended method should be tested with other refactoring types, such as rename field, inline method, encapsulate field, pull up field/method, and push down field/method (Oliveira et al., 2019).

6. Conclusion

The principal contribution of this research is a refactoring detection system based on deep learning and code histories. We collected the code modification types and locations using the GitHub commit histories, created a class-level vectorization with GRU, and then employed a LSTM-based RNN system to predict future refactoring. The results indicate that the recommended system can identify code modification requirements with almost no manual labeling.

Automatic refactoring detection reduces the time and effort required for refactoring and facilitates the code maintenance. The recommended method supports the development of tools that suggest refactoring operations according to the developer's previous decisions, creates an alternative solution for static code analysis, and decreases the effort required to create manually labeled datasets for machine learning algorithms. In addition, it can be employed for a wide range of programming language tasks, such as code classification and tagging, code similarity analysis, or clone detection.

The integrated evaluation of data without preprocessing is a widely used approach in deep learning. For example, convolutional neural networks can recognize objects without preprocessing an image. Our class-based code analysis method can provide the application of this approach to code analysis.

The practical implication of this study is to facilitate knowledge sharing among the software team members. All stakeholders in a software development team using a code-hosting platform for version control and collaboration should share the same coding culture to build quality products. However, the coding culture reveals itself to the leading members of the team and continues when delegating to new members of the team through the transfer of expertise. Adapting to a coding culture requires time for new developers and creates problems in testability, naming, alignment, performance, and coding. Moreover, when critical team members leave the team, the code culture information cannot be passed on to subsequent members. Having a project-specific and standardized understanding of code refactoring using historical data, which is suggested in this study, provides significant benefits to corporate memory and code culture.

Although we believe that the results of this study will provide useful insights for detecting code modification requirements, there are many areas for further research in deep learning and code analysis. The first direction would be to combine the outcomes of static code analysis tools with historical data to build a comprehensive perspective for identifying code modification requirements. In addition, other deep learning methods, such as transfer or reinforcement learning, can be used to investigate the factors behind code modifications.

CRediT authorship contribution statement

O.O. Büyük: Conceptualization, Methodology, Software, Formal analysis, Data curation, Writing – review & editing, Visualization. **A. Nizam:** Conceptualization, Methodology, Software, Validation, Formal analysis, Supervision, Project administration, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data is available with apache licence.

Fundings

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.jss.2023.111851.

References

Akbar, S.A., Kak, A.C., 2019. SCOR: Source code retrieval with semantics and order. In: IEEE International Working Conference on Mining Software Repositories, 2019-May(5). pp. 1–12. http://dx.doi.org/10.1109/MSR.2019.00012.
Akour, M., Alenezi, M., Alsghaier, H., 2022. Software refactoring prediction using SVM and optimization algorithms. Processes 10 (8), 1–10. http://dx.doi.org/10.3390/pr10081611.
Al-Shaaby, A., Aljamaan, H., Alshayeb, M., 2020. Bad smell detection using machine learning techniques: A systematic literature review. Arab. J. Sci. Eng. 45 (4), 2341–2369. http://dx.doi.org/10.1007/s13369-019-04311-w.
Alazba, A., Aljamaan, H., 2021. Code smell detection using feature selection and stacking ensemble: An empirical investigation. Inf. Softw. Technol. 138 (August 2020), 106648. http://dx.doi.org/10.1016/j.infsof.2021.106648.
Alenezi, M., Akour, M., Al Qasem, O., 2020. Harnessing deep learning algorithms to predict software refactoring. Telkomnika (Telecommun. Comput. Electr. Control) 18 (6), 2977–2982. http://dx.doi.org/10.12928/TELKOMNIKA.v18i6.16743.
Allamanis, M., Barr, E.T., Devanbu, P., Sutton, C., 2018. A survey of machine learning for big code and naturalness. ACM Comput. Surv. 51 (4).
AlOmar, E.A., Mkaouer, M.W., Ouni, A., 2021. Toward the automatic classification of Self-Affirmed Refactoring. J. Syst. Softw. 171, 110821. http://dx.doi.org/10.1016/j.jss.2020.110821.
Alon, U., Zilberstein, M., Levy, O., Yahav, E., 2019. Code2Vec: Learning distributed representations of code. In: Proceedings of the ACM on Programming Languages, 3(POPL). pp. 1–29. http://dx.doi.org/10.1145/3290353.
Arcelli Fontana, F., Mäntylä, M.V., Zanoni, M., Marino, A., 2016. Comparing and experimenting machine learning techniques for code smell detection. Empir. Softw. Eng. 21 (3), 1143–1191. http://dx.doi.org/10.1007/s10664-015-9378-4.
Atwi, H., Lin, B., Tsantalis, N., Kashiwa, Y., Kamei, Y., Ubayashi, N., Bavota, G., Lanza, M., 2021. P Y R EF: Refactoring detection in python projects, pp. 136–141. http://dx.doi.org/10.1109/SCAM52516.2021.00025.
Bavota, G., De Lucia, A., Di Penta, M., Oliveto, R., Palomba, F., 2015. An experimental investigation on the innate relationship between quality and refactoring. J. Syst. Softw. 107, 1–14. http://dx.doi.org/10.1016/j.jss.2015.05.024.

Benton, S., Ghanbari, A., Zhang, L., 2019. Defexts: A curated dataset of reproducible real-world bugs for modern JVM languages. In: Proceedings - 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion, ICSE-Companion 2019. pp. 47–50. http://dx.doi.org/10.1109/ICSE-Companion.2019.00035.

Bigonha, M.A.S., Ferreira, K., Souza, P., Sousa, B., Januário, M., Lima, D., 2019. The usefulness of software metric thresholds for detection of bad smells and fault prediction. Inf. Softw. Technol. 1152018, 79–92. http://dx.doi.org/10.1016/j.infsof.2019.08.005.

Borovits, N., Kumara, I., Di Nucci, D., Krishnan, P., Palma, S.D., Palomba, F., Tamburri, D.A., Heuvel, W.J., van den, 2022. FindICI: Using machine learning to detect linguistic inconsistencies between code and natural language descriptions in infrastructure-as-code. Empir. Softw. Eng. 27 (7), http://dx.doi.org/10.1007/s10664-022-10215-5.

Bryton, S., Brito E Abreu, F., Monteiro, M., 2010. Reducing subjectivity in code smells detection: Experimenting with the long method. In: Proceedings - 7th International Conference on the Quality of Information and Communications Technology, QUATIC 2010, 3. pp. 337–342. http://dx.doi.org/10.1109/QUATIC.2010.60.

Chen, X., Berkeley, U.C., Song, D., 2018. Tree-To-Tree Neural Networks for Program Translation. Nips.

Da Costa, D.A., McIntosh, S., Shang, W., Kulesza, U., Coelho, R., Hassan, A.E., 2017. A framework for evaluating the results of the SZZ approach for identifying bug-introducing changes. IEEE Trans. Softw. Eng. 43 (7), 641–657. http://dx.doi.org/10.1109/TSE.2016.2616306.

Dey, R., Salemt, F.M., 2017. Gate-variants of gated recurrent unit (GRU) neural networks. In: Midwest Symposium on Circuits and Systems, 2017-Augus(2). pp. 1597–1600. http://dx.doi.org/10.1109/MWSCAS.2017.8053243.

Ferenc, R., Tóth, Z., Ladányi, G., Siket, I., Gyimóthy, T., 2018. A public unified bug dataset for Java. In: ACM International Conference Proceeding Series. pp. 12–21. http://dx.doi.org/10.1145/3273934.3273936.

Fu, S., Shen, B., 2015. Code bad smell detection through evolutionary data mining. In: International Symposium on Empirical Software Engineering and Measurement, 2015-Novem. pp. 41–49. http://dx.doi.org/10.1109/ESEM.2015.7321194.

Géron, A., 2019. HandS-on Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques To Build Intelligent Systems. O'Reilly Media.

Goodfellow, I., Bengio, Y., Courville, A., 2016. Deep Learning. MIT Press.

Habchi, S., Moha, N., Rouvoy, R., 2021. Android code smells: From introduction to refactoring. J. Syst. Softw. 177, 110964. http://dx.doi.org/10.1016/j.jss.2021.110964.

He, P., Li, B., Liu, X., Chen, J., Ma, Y., 2015. An empirical study on software defect prediction with a simplified metric set. Inf. Softw. Technol. 59, 170–190. http://dx.doi.org/10.1016/j.infsof.2014.11.006.

Hochreiter, S., Schmidhuber, J., 1997. Long short-term memory. Neural Comput. 1780, 1735–1780.

Jain, P., Jain, A., Zhang, T., Abbeel, P., Gonzalez, J.E., Stoica, I., 2020. Contrastive code representation learning. In: ICLR 2021 Review.

Kádár, I., Hegedüs, P., Ferenc, R., Gyimóthy, T., 2016. A code refactoring dataset and its assessment regarding software maintainability. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, 2016-Janua. pp. 599–603. http://dx.doi.org/10.1109/SANER.2016.42.

Kak, A., 2019. Evaluating information retrieval algorithms with significance testing based on randomization and student ' s paired t-test. (Issue March).

Khomh, F., Vaucher, S., Guéehéeneuc, Y.G., Sahraoui, H., 2009. A bayesian approach for the detection of code and design smells. In: Proceedings - International Conference on Quality Software. pp. 305–314. http://dx.doi.org/10.1109/QSIC.2009.47.

Kohavi, R., 1995. A study of cross-validation and bootstrap for accuracy estimation and model selection. In: International Joint Conference of Artificial Intelligence, 2001.

Kurbatova, Z., Veselov, I., Golubev, Y., Bryksin, T., 2020. Recommendation of move method refactoring using path-based representation of code. In: Proceedings - 2020 IEEE/ACM 42nd International Conference on Software Engineering Workshops, ICSEW 2020. pp. 315–322. http://dx.doi.org/10.1145/3387940.3392191.

Lacerda, G., Petrillo, F., Pimenta, M., Guéhéneuc, Y.G., 2020. Code smells and refactoring: A tertiary systematic review of challenges and observations. J. Syst. Softw. 167, http://dx.doi.org/10.1016/j.jss.2020.110610.

Lam, A.N., Nguyen, A.T., Nguyen, H.A., 2017. Bug localization with combination of deep learning and information retrieval. In: 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC). pp. 218–229. http://dx.doi.org/10.1109/ICPC.2017.24.

Le, T.H.M., Chen, H.A.O., Babar, M.A.L.I., 2014. Deep learning for source code modeling and generation: Models, applications and challenges. pp. 1–37.

Liang, H., Sun, L., Wang, M., Yang, Y., 2019. Deep learning with customized abstract syntax tree for bug localization. IEEE Access 7, 116309–116320. http://dx.doi.org/10.1109/ACCESS.2019.2936948.

Liu, H., Xu, Z., Zou, Y., 2018. Deep learning based feature envy detection. In: ASE 2018 - Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. pp. 385–396. http://dx.doi.org/10.1145/3238147.3238166.

Maddeh, M., Ayouni, S., Alyahya, S., Hajjej, F., 2021. Decision tree-based design defects detection. IEEE Access 9, 71606–71614. http://dx.doi.org/10.1109/ACCESS.2021.3078724.

Martin, F., 2018. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional.

Mens, T., Tourwé, T., 2004. A survey of software refactoring. IEEE Trans. Softw. Eng. 30 (2), 126–139. http://dx.doi.org/10.1109/TSE.2004.1265817.

Mikolov, T., Chen, K., Corrado, G., Dean, J., 2013. Efficient estimation of word representations in vector space. In: 1st International Conference on Learning Representations, ICLR 2013 - Workshop Track Proceedings. pp. 1–12.

Moghadam, I.H., Cinnéide, M.Ó., Zarepour, F., Jahanmir, M.A., 2021. RefDetect : A multi-language refactoring detection tool based on string alignment. IEEE Access 9, http://dx.doi.org/10.1109/ACCESS.2021.3086689.

Moha, N., Guéhéneuc, Y.G., Duchien, L., Le Meur, A.F., 2010. DECOR: A method for the specification and detection of code and design smells. IEEE Trans. Softw. Eng. 36 (1), 20–36. http://dx.doi.org/10.1109/TSE.2009.50.

Mou, L., Li, G., Zhang, L., Wang, T., Jin, Z., 2016. Convolutional neural networks over tree structures for programming language processing. In: 30th AAAI Conference on Artificial Intelligence, AAAI 2016. pp. 1287–1293. http://dx.doi.org/10.1609/aaai.v30i1.10139.

Oliveira, J., Gheyi, R., Mongiovi, M., Soares, G., Ribeiro, M., Garcia, A., 2019. Revisiting the refactoring mechanics. Inf. Softw. Technol. 1102018, 136–138. http://dx.doi.org/10.1016/j.infsof.2019.03.002.

Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., Poshyvanyk, D., De Lucia, A., 2015. Mining version histories for detecting code smells. IEEE Trans. Softw. Eng. 41 (5), 462–489. http://dx.doi.org/10.1109/TSE.2014.2372760.

Panigrahi, R., Kuanar, S.K., Misra, S., Kumar, L., 2022. Class-level refactoring prediction by ensemble learning with various feature selection techniques. Appl. Sci. (Switz.and) 12 (23), 1–29. http://dx.doi.org/10.3390/app122312217.

Pantiuchina, J., Zampetti, F., Scalabrino, S., Piantadosi, V., Oliveto, R., Bavota, G., Di Penta, M., 2020. Why developers refactor source code: A mining-based study. ACM Trans. Softw. Eng. Methodol. 29 (4), 1–31. http://dx.doi.org/10.1145/3408302.

Parhi, R., Nowak, R.D., 2020. The role of neural network activation functions. IEEE Signal Process. Lett. 27, 1779–1783. http://dx.doi.org/10.1109/LSP.2020.3027517.

PMD, 2023. PMD. https://pmd.github.io/.

Pradel, M., Sen, K., 2018. A replication of DeepBugs: A learning approach to name-based bug detection. In: ACMProgram. Lang. 2, OOPSLA, 2(November). http://dx.doi.org/10.1145/3276517.

Rahman, A., Parnin, C., Williams, L., 2019. The seven sins: Security smells in infrastructure as code scripts. In: Proceedings - International Conference on Software Engineering, 2019-May. pp. 164–175. http://dx.doi.org/10.1109/ICSE.2019.00033.

Sharma, T., Efstathiou, V., Louridas, P., Spinellis, D., 2021. Code smell detection by deep direct-learning and transfer-learning. J. Syst. Softw. 176 (110936).

Sharma, T., Spinellis, D., 2018. A survey on software smells. J. Syst. Softw. 138, 158–173. http://dx.doi.org/10.1016/j.jss.2017.12.034.

Shi, K., Lu, Y., Chang, J., Wei, Z., 2020. PathPair2Vec: An AST path pair-based code representation method for defect prediction. J. Comput. Lang. 59 (May), 100979. http://dx.doi.org/10.1016/j.cola.2020.100979.

Shin, R., Kant, N., Gupta, K., Bender, C., Trabucco, B., Singh, R., Song, D., 2019. Synthetic datasets for neural program synthesis. In: ICLR. pp. 1–16.

Silva, D., Tsantalis, N., Valente, M.T., 2016. Why we refactor? Confessions of GitHub contributors. http://aserg-ufmg.github.io/why-we-refactor.

Sivaraman, A., Abreu, R., Scott, A., Akomolede, T., Chandra, S., 2022. Mining idioms in the wild. In: Proceedings - International Conference on Software Engineering. pp. 187–196. http://dx.doi.org/10.1109/ICSE-SEIP55303.2022.9794062.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R., 2014. Dropout: A simple way to prevent neural networks from overfitting. J. Mach. Learn. Res. 15 (1), 1929–1958. http://dx.doi.org/10.1016/0370-2693(93)90272-J.

Sui, Y., Cheng, X., Zhang, G., Wang, H., 2020. Flow2Vec: Value-flow-based precise code embedding. In: Proceedings of the ACM on Programming Languages, 4(OOPSLA). http://dx.doi.org/10.1145/3428301.

Tsantalis, N., Chatzigeorgiou, A., 2011. Ranking refactoring suggestions based on historical volatility. In: Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR. pp. 25–34. http://dx.doi.org/10.1109/CSMR.2011.7.

Ullah, F., Jabbar, S., Al-Turjman, F., 2020. Programmers' de-anonymization using a hybrid approach of abstract syntax tree and deep learning. Technol. Forecast. Soc. Change 159 (June), 120186. http://dx.doi.org/10.1016/j.techfore.2020.120186.

Ullah, F., Naeem, M.R., Naeem, H., Cheng, X., Alazab, M., 2022. CroLSSim: Cross-language software similarity detector using hybrid approach of LSA-based AST-MDrep features and CNN-LSTM model. Int. J. Intell. Syst..

Vidal, S.A., Marcos, C., Díaz-Pace, J.A., 2016. An approach to prioritize code smells for refactoring. Autom. Softw. Eng. 23 (3), 501–532. http://dx.doi.org/10.1007/s10515-014-0175-x.

Wang, W., Li, G., Shen, S., Xia, X., Jin, Z., 2020. Modular tree network for source code representation learning. ACM Trans. Softw. Eng. Methodol. 29 (4), http://dx.doi.org/10.1145/3409331.

Weißgerber, P., Diehl, S., 2006. Identifying refactorings from source-code changes.

Williams, C., Spacco, J., 2008. SZZ revisited. 32, http://dx.doi.org/10.1145/1390817.1390826.

Xia, X., Lo, D., Wang, X., Yang, X., 2016. Collective personalized change classification with multiobjective search. IEEE Trans. Reliab. 65 (4), 1810–1829. http://dx.doi.org/10.1109/TR.2016.2588139.

Xue, J., Mao, X., Lu, Y., Yu, Y., Wang, S., 2019. History-driven fix for code quality issues. IEEE Access 7, 111637–111648. http://dx.doi.org/10.1109/ACCESS.2019.2934975.

Yahav, E., Levy, O., 2019. Code2Seq:Generatıng sequences from structured representat ıons of code. In: ICLR 2019 CODE2SEQ:, 1. pp. 1–19.

Yonai, H., Hayase, Y., Kitagawa, H., 2019. Mercem: Method name recommendation based on call graph embedding. In: Proceedings - Asia-Pacific Software Engineering Conference, APSEC, 2019-Decem. pp. 134–141. http://dx.doi.org/10.1109/APSEC48747.2019.00027.

Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., Liu, X., 2019a. A novel neural source code representation b ased on abstract syntax tree. pp. 783–794. http://dx.doi.org/10.1109/ICSE.2019.00086.

Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., Liu, X., 2019b. A novel neural source code representation based on abstract syntax tree. In: Proceedings - International Conference on Software Engineering, 2019-May. pp. 783–794. http://dx.doi.org/10.1109/ICSE.2019.00086.

Zhuang, Y., Suneja, S., Thost, V., Domeniconi, G., Morari, A., Laredo, J., 2019. Software vulnerability detection via deep learning over disaggregated code graph representation. ArXiv Preprint.

Buyuk O.O. was born in 1994 in Kadıköy, Istanbul, Turkey. In 2018 and 2021, he earned his bachelor's degree in computer engineering and biomedical engineering as a double major and master's degree in computer engineering from Fatih Sultan Mehmet Vakif University, respectively. He is currently Ph.D. student at Fatih Sultan Mehmet Vakif University and works at TÜBİTAK (The Scientific and Technological Research Council of Turkey). His research interests lie primarily in the fields of software development, deep learning, parallel programming, and clustering.

Oguzhan Oktay Buyuk has several awards, paper degrees, and honors. These are from GCIP (Global Cleantech Innovation Programme) by United Nations, e-Government from Turkey and Microsoft; IEEE PES in ICSG (Internationation Congress of Smart Grids); Academical Success Scholarship of Fatih Sultan Mehmet Vakif University, respectively.

Nizam A. was born in Fatih, İstanbul, Turkey in 1976. He received the B.S. degree in electronical and communication engineering from the Yıldiz Technical University, İstanbul, in 1997, and the and M.S. and Ph.D. degrees in electronic–biomedical engineering from İstanbul Technical University, İstanbul, in 2000 and 2009.

From 1997 to 2011, he worked at ISKI as a software engineer, project manager, and internal IT department manager. Since 2011, he has been an Assistant Professor with the Computer Engineering Department, Fatih Sultan Mehmet Vakif University, İstanbul, Turkey. He is the author of four books, one book chapter, and five articles. His research interests include software engineering, relational database concepts, and data science.

Dr. Nizam's awards and honors include the TÜBA (Turkey Academy of Science), the University Textbooks Award Program Best Original Book Award with Software Project Management Book.