

Just-in-Time Code Duplicates Extraction

Eman Abdullah AlOmar^{a,1,*}, Anton Ivanov^{b,1}, Zarina Kurbatova^c, Yaroslav Golubev^c, Mohamed Wiem Mkaouer^d, Ali Ouni^e, Timofey Bryksin^f, Le Nguyen^d, Amit Kini^d, Aditya Thakur^d

^a*Stevens Institute of Technology, Hoboken, NJ, USA*

^b*HSE University, Moscow, Russia*

^c*JetBrains Research, Belgrade, Serbia*

^d*Rochester Institute of Technology, Rochester, NY, USA*

^e*ETS Montreal, University of Quebec, Montreal, QC, Canada*

^f*JetBrains Research, Limassol, Cyprus*

Abstract

Context: Refactoring is a critical task in software maintenance, and is usually performed to enforce better design and coding practices, while coping with design defects. The *Extract Method* refactoring is widely used for merging duplicate code fragments into a single new method. Several studies attempted to recommend *Extract Method* refactoring opportunities using different techniques, including program slicing, program dependency graph analysis, change history analysis, structural similarity, and feature extraction. However, irrespective of the method, most of the existing approaches interfere with the developer’s workflow: they require the developer to stop coding and analyze the suggested opportunities, and also consider all refactoring suggestions in the entire project without focusing on the development context.

Objective: To increase the adoption of the *Extract Method* refactoring, in this paper, we aim to investigate the effectiveness of machine learning and deep learning algorithms for its recommendation while maintaining the workflow of the developer.

Method: The proposed approach relies on mining prior applied *Extract Method* refactorings and extracting their features to train a deep learning classifier that detects them in the user’s code. We implemented our approach as a plugin for IntelliJ IDEA called ANTICOPYPASTER. To develop our approach, we trained and evaluated various popular models on a dataset of 18,942 code fragments from 13 Open Source Apache projects.

*Corresponding author

Email addresses: ealomar@stevens.edu (Eman Abdullah AlOmar), apivanov_1@edu.hse.ru (Anton Ivanov), zarina.kurbatova@jetbrains.com (Zarina Kurbatova), yaroslav.golubev@jetbrains.com (Yaroslav Golubev), mwmvse@rit.edu (Mohamed Wiem Mkaouer), ali.ouni@etsmt1.ca (Ali Ouni), timofey.bryksin@jetbrains.com (Timofey Bryksin), ln8378@rit.edu (Le Nguyen), ak3328@rit.edu (Amit Kini), at4415@rit.edu (Aditya Thakur)

¹Authors contributed equally

Results: The results show that the best model is the Convolutional Neural Network (CNN), which recommends appropriate *Extract Method* refactorings with an F-measure of 0.82. We also conducted a qualitative study with 72 developers to evaluate the usefulness of the developed plugin.

Conclusion: The results show that developers tend to appreciate the idea of the approach and are satisfied with various aspects of the plugin’s operation.

Keywords: refactoring, machine learning, software quality

1. Introduction

Duplicating a code fragment is the act of copying and pasting it with or without minor modifications into another section of the code base. Despite being an intuitive practice of code reuse, duplicate code brings its own challenges to software maintenance and evolution [1, 2]. Recent studies have shown that duplicate code has become a problem that affects both developers and researchers. Developers can suffer from fixing a bug in a duplicate code fragment, which might then need to be applied to all of its siblings [3]. This can complicate and slow down the maintenance and lead to bug propagation. On the other hand, researchers may face problems when building machine learning models, combined with natural language processing techniques, which are designed to learn from code to support various software development practices. The threat of duplicate code can easily introduce data leakage when appearing in both training and testing data [4, 5]. Therefore, removing duplicate code via refactoring has become a natural response to arising challenges [6].

Refactoring is the practice of improving software quality without altering its behavior [7]. Developers intuitively refactor their code for multiple purposes: improving program comprehension, removing duplicate code, reducing complexity, dealing with technical debt, and removing code smells [8, 9, 10, 11]. Refactoring duplicate code consists in taking a code fragment and moving it to create a new method, while replacing all instances of that fragment with a call to this newly created method. This refactoring is known as *Extract Method*. Various studies have recommended refactorings using different motivation and base, such as improving code structure [12, 13], feature extraction [14, 15, 16], reducing code duplication [17, 18, 19, 20], and removing the *Long Method* code smell [21, 22, 23, 24, 25, 26].

Despite their promising results, the adoption of these studies is challenged by their need to exhaustively search the entire code base to recommend proper *Extract Method* refactorings. In other words, they take the entire source code of the project as input and analyze it to find any *Extract Method* refactoring opportunities. However, developers may not have the privilege or the knowledge to perform a program-wide refactoring, which makes them reluctant to adopt these recommendations. Moreover, this process requires developers to find a separate time to review various refactoring suggestions that are unrelated to their current work in the code, which reduces the chances of these recommendations to be

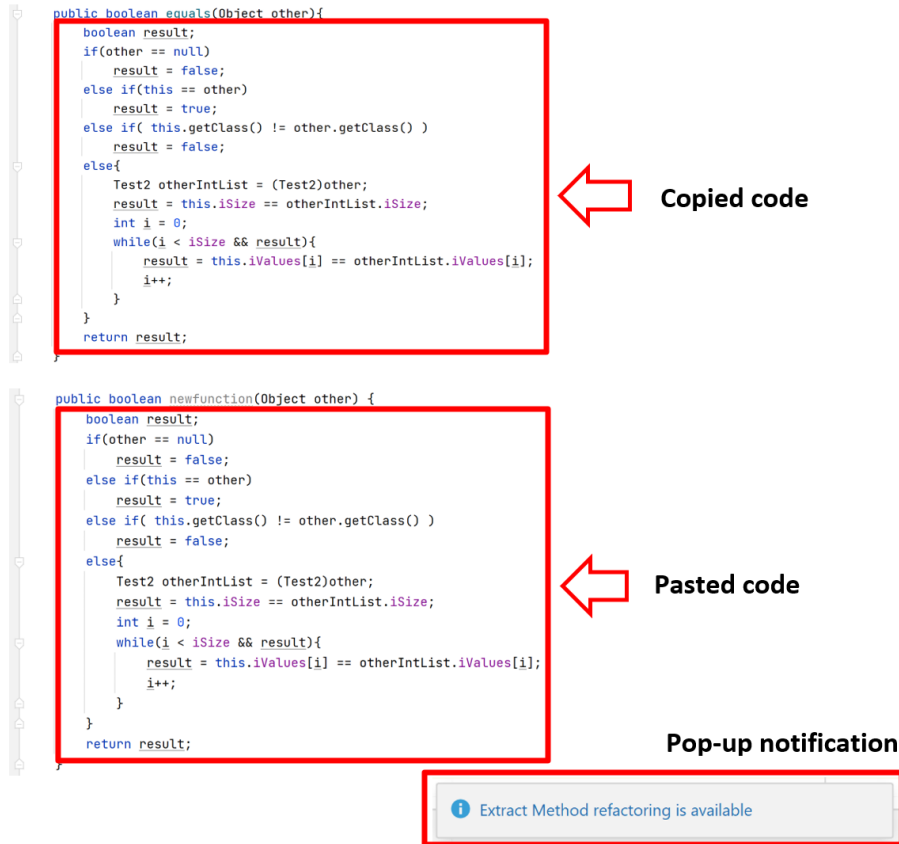
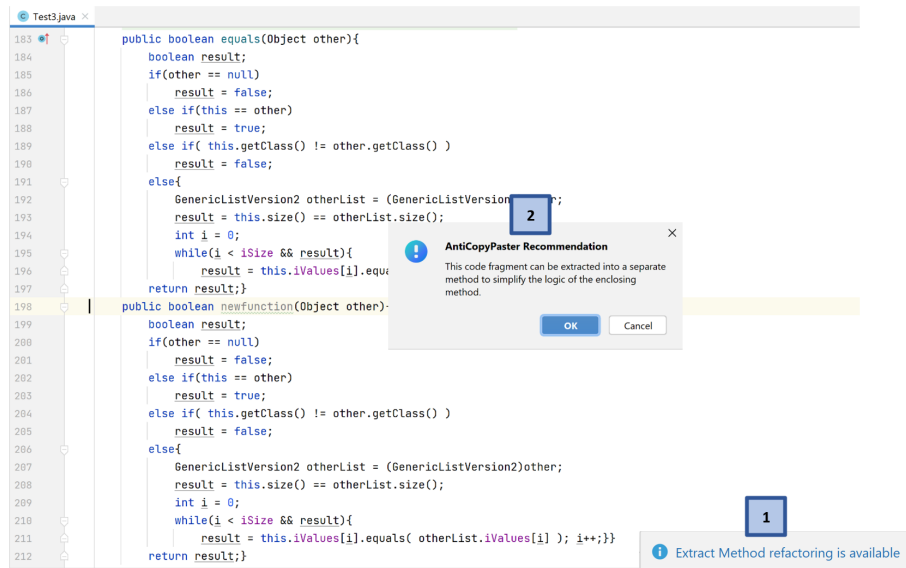


Figure 1: *Extract Method* refactoring opportunity (code fragments extracted from [28]).

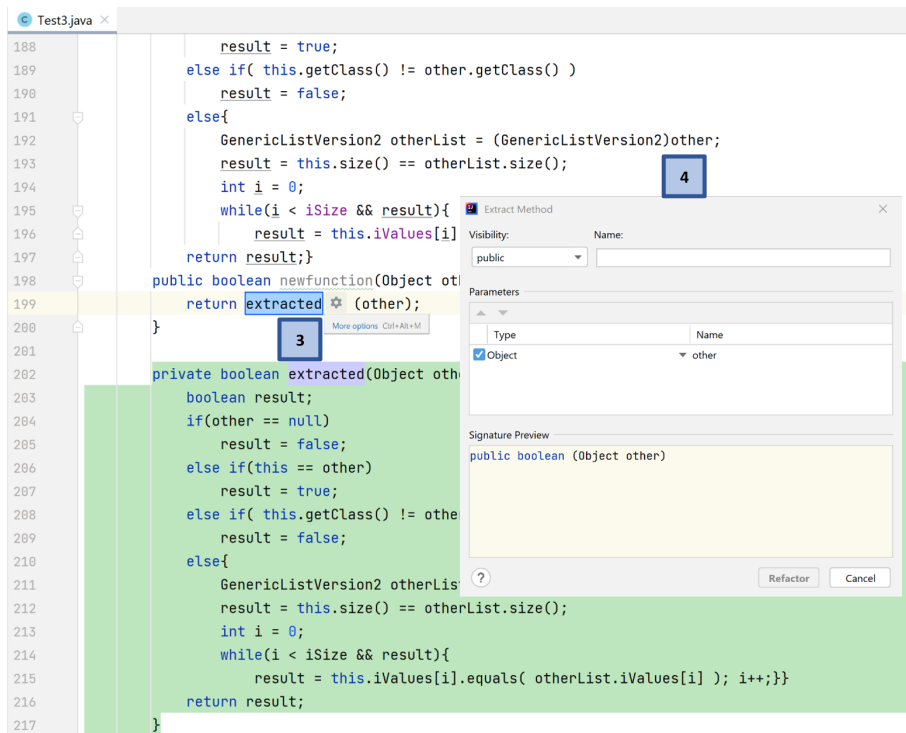
accepted in practice. Even though the *Extract Method* refactoring is a built-in feature in modern programming environments, its adoption by developers is still limited even in this setting [27].

To address the above-mentioned challenges, the goal of this paper is to aid developers with *just-in-time* refactoring of duplicate code. Unlike the existing approaches that follow a posterior approach of removing accumulated duplicate code, we aim at increasing the awareness of developers *while writing* their code, *i.e.*, removing duplicate code as soon as it is introduced in their code base. To do so, we design an automated approach implemented as an IntelliJ IDEA² plugin called ANTICOPYPASTER that monitors the introduction of potential duplicate code and pro-actively recommends its refactoring using the IDE's *Extract Method* feature.

²IntelliJ IDEA: <https://www.jetbrains.com/idea/>



(a) Identification of Duplicate Code instances.



(b) Correction of Duplicate Code through the application of *Extract Method* refactoring.

Figure 2: ANTI-COPYPASTER in action, showing the identified Duplicated Code and the recommended *Extract Method* refactoring.

As shown in Figures 1 and 2, when a duplicate piece of code is pasted and is not edited for some time, a pop-up notification appears at the bottom of the screen, alerting the developer of a potential *Extract Method* refactoring. The developer can choose whether to click on the notification or ignore it, until it disappears after a few seconds. If the notification is clicked, the *Extract Method* feature is called with the duplicate code as input, and a refactoring preview window is opened. The developer can either apply the refactoring and suggest a name to the newly created method, or cancel the process.

The main advantage of this solution, in comparison with previous works, is the ability to recommend *just-in-time* refactorings, which increases the chances of their acceptance, since it recommends changes to a code that is (1) just edited by the developer, and (2) within the current context of development. However, not all duplicate code fragments need to be extracted, and the main challenge is being able to recommend refactoring only when the refactoring is *worth it*, in order not to bother the developer by suggesting to extract random pieces of code or trivial statements.

In essence, we tackle the problem of whether the given duplicate code fragment should be extracted as a binary classification problem. First, the pasted code fragment is parsed using the IDE’s Program Structure Interface (PSI)³ to generate its corresponding syntactic and semantic model. This model is used to calculate a set of 78 comprehensive structural and semantic metrics, previously used in various studies recommending the *Extract Method* refactoring [16, 23, 29, 30, 31, 32, 33, 34, 35, 36]. Based on these metrics, a binary classification is performed to decide whether or not to suggest the refactoring. To handle the high dimensionality of the data, we design our binary classifier using a Convolutional Neural Network (CNN).

We evaluate our approach in terms of two dimensions: *correctness* and *usefulness*. The first dimension evaluates the CNN’s learning ability to correctly detect whether a given code fragment should be refactored. We trained and tested our model on a dataset of 18,942 code fragments mined from 13 mature Apache projects. Our experimental results show that our CNN model achieves high performance in the binary classification, with an F-measure of 0.82, which outperforms all other evaluated machine learning algorithms, such as Random Forest, Support Vector Machine, Naive Bayes, and Logistic Regression, while also being lightweight and convenient to use.

As for the second dimension, we designed a survey that invites developers to use ANTICOPYPASTER and reflect on its usability and usefulness. In total, 72 developers participated in the survey. The results show that the vast majority of participants found our ANTICOPYPASTER tool useful and were satisfied with its operation. Furthermore, the survey has also shown that the majority of participants are not necessarily familiar with the *Extract Method* built-in IDE feature, and therefore, we foresee their usage of the plugin as an opportunity to raise the awareness of this refactoring type.

³PSI: <https://plugins.jetbrains.com/docs/intellij/psi-files.html>

This paper extends our recently accepted tool paper [37] by providing more details about the used model and its design, discussing the used metrics and the performance of various models, and describing the full comprehensive analysis of user’s feedback, including their general usage of refactorings and the open coding of their answers. To summarize, this paper makes the following key contributions:

- We design and implement a novel approach called ANTI`COPY`PASTER that pro-actively identifies and recommends the *Extract Method* refactoring as soon as a duplicate code fragment is pasted into the file.
- We train, deploy, and evaluate a CNN model that has shown good performance with respect to other models, achieving an F-measure of 0.82.
- We provide ANTI`COPY`PASTER as an open source tool publicly available for the community [38]. We also provide a comprehensive replication package containing the dataset, survey results, and scripts, as well as documentation on how to embed another model into the plugin, allowing researchers and practitioners to customize the tool and match their own preferences [39].

The remainder of this paper is organized as follows. Section 2 elaborates on the main concepts of the model. Section 3 reviews the existing studies related to the *Extract Method* refactoring opportunities. Section 4 outlines our approach, including data collection, metrics selection, and model training. The tool implementation is discussed in Section 5. Section 6 describes the conducted evaluation and its research questions, as well as our findings. Section 7 captures threats to the validity of our work, and we conclude the paper in Section 8.

2. Background

In this section, we elaborate on the main concepts of CNNs as discussed by Krizhevsky *et al.* [40].

Convolutional neural networks (CNNs) are a type of neural network architecture that uses a series of filters to extract significant features for the purpose of classification. These filters are typically convolutional layers, mapping inputs from the previous layer to the next layer by applying trainable weights to a predefined window of data and then outputting the weighted sum of that window. Another filter that can be applied is a pooling layer. Pooling allows the accumulation of features from maps generated during the convolution. The idea of pooling is to reduce the spatial size of the representation and decrease the number of parameters and computation in the network. Typically, this is done via a max pooling layer, which again looks at a window of data from the previous input layer, and then takes the maximum value in that window to output to the next layer. Pooling layers are a computationally efficient way to distill significant features from the previous layer, since there are no weights to train. In our work, we also used traditional fully connected feedforward layers as well,

which take input values, apply a trainable weight to it and feed it through an activation function before outputting.

Since neural networks may quickly overfit when trained on fewer but similar instances, there are various strategies to mitigate such overfitting. Dropout is a regularization strategy that deactivates a defined percentage of neurons in a layer at random every training epoch. Deactivating these neurons ensures that not every neuron is exposed to all the data every epoch, so it cannot overfit to superfluous patterns in the data. Also, it reduces any feature detection co-adaptations as the deactivated neurons cannot influence the retained ones.

Batch normalization is a transformation applied to the current batch of input data to the network. As the name implies, it normalizes the current input batch to take out unnecessary bias inherent in the feature data, such as some features having a different order of magnitude than others. By normalizing the input data, the CNN can train more effectively and will converge with higher accuracy.

The choice of activation function is critical to the learning rate of neural networks. It also specifies the model's type of prediction. In this paper, Rectified Linear Unit (ReLU) was selected as the activation function. The ReLU function is a piecewise function that is zero for negative values and linear for positive values. This is a classic function used in neural networks because it masks out any negatively weighted features and then gives linear importance to the positively weighted ones.

During the model's training, the loss function computes the error between the ground truth data labels and the predicted labels. This error between the actual versus the predicted labels (*i.e.*, loss) is used to update the weights of the model to maximize accuracy (minimize loss). Loss minimization is carried out by first propagating forward through the network, and applying all of the current filters and weights to the input data to calculate predictions. Then, the loss is calculated from the current predictions, before performing a backward propagation to find the gradient of each weight with respect to the loss (*i.e.*, finding the contribution of each weight to the loss). Using the gradients from each weight, the stochastic gradient descent is carried out on the loss space, which results in minimizing it.

3. Related Work

Various studies that relate to software refactoring have been of importance to both practitioners and researchers. A considerable effort has been spent by the research community on identifying and suggesting *Extract Method* refactorings. The focus of these studies ranges from using program slicing techniques [30, 31, 41] and graph representations of code [12, 23, 32, 35] to relying on scoring functions to find the most appropriate refactoring candidates [21, 29, 33, 34] and using machine learning techniques [14, 15, 16, 36]. We summarize the key studies in Table 1.

Maruyama [30] developed a semi-automated approach for suggesting refactorings, which decomposed the control flow graph using block-based program

Table 1: Related work in identifying the *Extract Method* refactoring opportunities.

Study	Year	Approach	Technique	Tool	Design Defect	Plugin?
Murayama [30]	2001	Rule-based	Code slicing	Not mentioned	Not mentioned	No
Murphy-Hill & Black [9]	2008	Rule-based	Assertion	Not mentioned	Not mentioned	No
Tsantalis & Chatzigeorgiou [41, 31]	2009 & 2011	Rule-based	Code slicing	JDeodorant	Long Method	Yes
Yang <i>et al.</i> [21]	2009	Score-based	Fragment identification	AutoMed	Long Method	No
Kanemitsu <i>et al.</i> [12]	2011	Score-based	Program dependency graph	ReAF	Not mentioned	No
Sharma [32]	2012	Graph-based	Data & structure dependency	N/A	Not mentioned	No
Silva <i>et al.</i> [33, 34]	2014 & 2015	Graph-based	Structural similarity	JExtract	Not mentioned	Yes
Charalampidou <i>et al.</i> [42]	2016	Score-based	Functional relevance	SEMI	Long Method	No
Haas & Hummel [29]	2016	Score-based	Control & data flow graph	ComQAT	Long Method	No
Xu <i>et al.</i> [14]	2017	ML-based	Feature extraction	GEMS	Not mentioned	No
Yue <i>et al.</i> [15]	2018	ML-based	Feature extraction	CREC	Code Clone	No
Yoshida <i>et al.</i> [17]	2019	Rule-based	Code modification analysis	Not mentioned	Code Clone	Yes
Aniche <i>et al.</i> [16]	2020	ML-based	Feature extraction	N/A	Not mentioned	No
Van der Leij <i>et al.</i> [36]	2021	ML-based	Feature extraction	N/A	Not mentioned	No
Shahidi <i>et al.</i> [35]	2022	Graph-based	Dependency graph analysis	N/A	Long Method	No
Tiwari & Joshi [23]	2022	Graph-based	Segmentation	N/A	Long Method	Yes
This work		DL-based	Feature extraction	ANTICOPYPASTER	Duplicate Code	Yes

slicing. This approach was later adapted and implemented by Tsantalis and Chatzigeorgiou [41] in the JDEODORANT tool that identified *Extract Method* refactoring opportunities using code slicing along with a set of rules to ensure behavior preservation after slice extraction. In a follow-up work, Tsantalis and Chatzigeorgiou [31] proposed a set of additional behavior preservation rules that exclude refactoring opportunities related to slices, the extraction of which could possibly cause a change in the program behavior. In another study, Murphy-Hill and Black [9] presented three features to improve the adoption and usage of the *Extract Method* refactoring, namely, selection assist, box view, and refactoring annotation. Their formative study shows that user satisfaction was significantly increased with these features. Sharma [32] proposed *Extract Method* candidates based on the data and the structure dependency graph. Their suggestions were obtained by eliminating the longest dependency edge in the graph.

Kanemitsu *et al.* [12] presented a visualization method for identifying *Extract Method* refactorings and introduced an implementation of the proposed method called REAF. Another approach was proposed by Yang *et al.* [21], the authors identified fragments to be extracted from long methods. Their approach is implemented as a prototype called AUTOMED. The evaluation results suggested that the approach may reduce the refactoring cost by 40%.

Silva *et al.* [33] used a similarity-based approach to recommend automated *Extract Method* refactoring opportunities that hide structural dependencies that are rarely used by the remaining statements in the original method. Their evaluation on a sample of 81 *Extract Method* opportunities achieved the precision and recall rates close to 50% when detecting refactoring instances. In another study, Silva *et al.* [34] extended their work by designing an Eclipse plugin called JEXTRACT that automatically identified, ranked, and applied refactorings upon request. Inspired by the study of Silva *et al.* [34], Haas and Hummel [29] developed a scoring function aimed to decrease the complexity of code by considering the code's length and nesting depth. The evaluation against 10 experienced developers showed that they accepted 86% of the suggested refactorings. Another study by Charalampidou *et al.* [42] shows the application of functional relevance to detect the *Long Method* code smell. The authors developed a tool called SEMI to automate the application of this approach for Java classes.

Xu *et al.* [14] proposed a plugin called GEMS that used both structural and functional features of code fragments from the real-world *Extract Method* refactorings to train a model to suggest refactorings. GEMS utilized the same extraction algorithm as JEXTRACT [34] to create code fragments for the extraction. Yue *et al.* [15] presented a tool called CREC that combined static analysis and the analysis of the history of code to suggest code clones that should be extracted into a separate method. Another experiment was conducted using a pro-active clone recommendation system. Yoshida *et al.* [17] designed an Eclipse plugin that tracks user code modification and constructed a system for supporting clone refactoring. When the system detects an *Extract Method* refactoring being performed, it automatically searches for clones of the extracted fragment and suggests to extract them as well. However, this still requires a developer to think about whether a code clone is worth refactoring.

Aniche *et al.* [16] used a machine learning approach that involves predicting refactorings using code, process, and ownership metrics. The resulting models predict 20 different refactorings at the levels of a class, method, and variable with an accuracy often higher than 90%. Another experiment that predicts refactorings was conducted using quality metrics. Van der Leij *et al.* [36] explored the recommendation of the *Extract Method* refactoring at ING. They observed that machine learning models can recommend *Extract Method* refactorings with high accuracy, and the user study reveals that ING experts tend to agree with most of the recommendations of the model.

More recently, Shahidi *et al.* [35] automatically identified and refactored the *Long Method* code smells in Java code using advanced graph analysis techniques. Their proposed approach was evaluated on five different Java projects. The findings reveal the applicability of the proposed method in establishing the single responsibility principle with a 21% improvement. In another study, Tiwari and Joshi [23] introduced *Segmentation* as a graph-based technique to identify *Extract Method* refactoring with the aim of achieving higher performance with fewer refactoring suggestions. The authors compared their approach against two state-of-the-art approaches, *i.e.*, JEXTRACT and SEMI, and showed an improvement over both of them.

Overall, recommending *Extract Method* refactoring opportunities has been extensively studied [16, 17, 31, 36]. Although some of the proposed techniques utilized various code metrics as a new way for recommending *Extract Method* refactoring, to the best of our knowledge, no prior studies have proposed a just-in-time automated *Extract Method* refactoring tool for the purpose of specifically eliminating code duplication while using code metrics as features to predict whether a piece of code should be extracted. The *just-in-time* automatic aspect is crucial since it allows developers to remain in the context of the suggestion and thus make a timely decision. This alleviates the burden of reviewing a long list of refactoring opportunities, located in code fragments that are irrelevant to development context.

To gain a more in-depth understanding of the issue, increase the awareness of duplicate code, and increase the adoption of *Extract Method* refactorings, in this paper, we develop a tool called ANTICOPYPASTER that is fast, conformable to use, and integrated into the developer’s IDE editor. Our study complements the existing efforts that are carried out to recommend *Extract Method* refactorings in general [9, 16, 32, 33, 36] or specifically in order to eliminate certain code smells [17, 23, 29, 31, 35, 42].

4. Approach

In a nutshell, the goal of our work is to automatically provide *just-in-time* recommendations of *Extract Method* refactoring opportunities as soon as duplicate code is introduced in the opened file in the IDE. Our approach takes code metrics as input and makes a binary decision on whether the code fragment has to be extracted. The present work can be divided into four phases as shown in Figure 3. It consists of: (1) data collection, (2) refactoring detection, (3)

Table 2: The overview of the data.

Item	Count
Number of projects	13
Software quality metrics	78
Extracted code fragments (Positive Examples)	9,471
Non-Extracted code fragments (Negative Examples)	1,000,000
Selected non-Extracted code fragments (Negative Examples)	9,471
Final dataset	18,942

code metrics selection, and (4) tool design and evaluation. The dataset, tool, and scripts utilized in this study are available in the replication package [39] for extension and replication purposes.

4.1. Data Collection

Our first step consists of selecting 13 mature projects from the Apache Software Foundation,⁴ which are popular open-source Java projects hosted on GitHub [43, 44]. These curated projects were selected with respect to both project size and activity, while verifying that they were Java-based, the only language supported by RefactoringMiner [45, 46]. An overview of the extracted data is provided in Table 2.

4.2. Refactoring Detection

To extract the entire refactoring history of each project, we used RefactoringMiner v2.0,⁵ a widely-used refactoring detection tool introduced by Tsantalis *et al.* [45, 46]. We decided to use RefactoringMiner as it has shown good results in detecting refactorings compared to other available tools (a precision of 99.8% and a recall of 95.8%) and is suitable for a study that requires a high degree of automation since it can be used through its external API.

We identify methods that underwent an *Extract Method* refactoring (*i.e.*, positive examples) using RefactoringMiner. In total, the tool mined 9,471 cases of *Extract Method* refactorings. Specifically, we discovered *Extract Method* refactorings, then traversed the history to the previous commit and took the code fragment that had been extracted. This allowed us to detect fragments that are *worth* of being extracted, since they were extracted in mature projects. These refactorings are not necessarily only applied in the context of duplicate code, and thus our model learns from various contexts (*e.g.*, splitting long methods). Our model is not intended to identify duplicate code, since this is handled by another algorithm in the tool, but to evaluate whether the given duplicate code is worth refactoring.

⁴Apache projects on GitHub: <https://github.com/apache>

⁵RefactoringMiner: <https://github.com/tsantalis/RefactoringMiner>

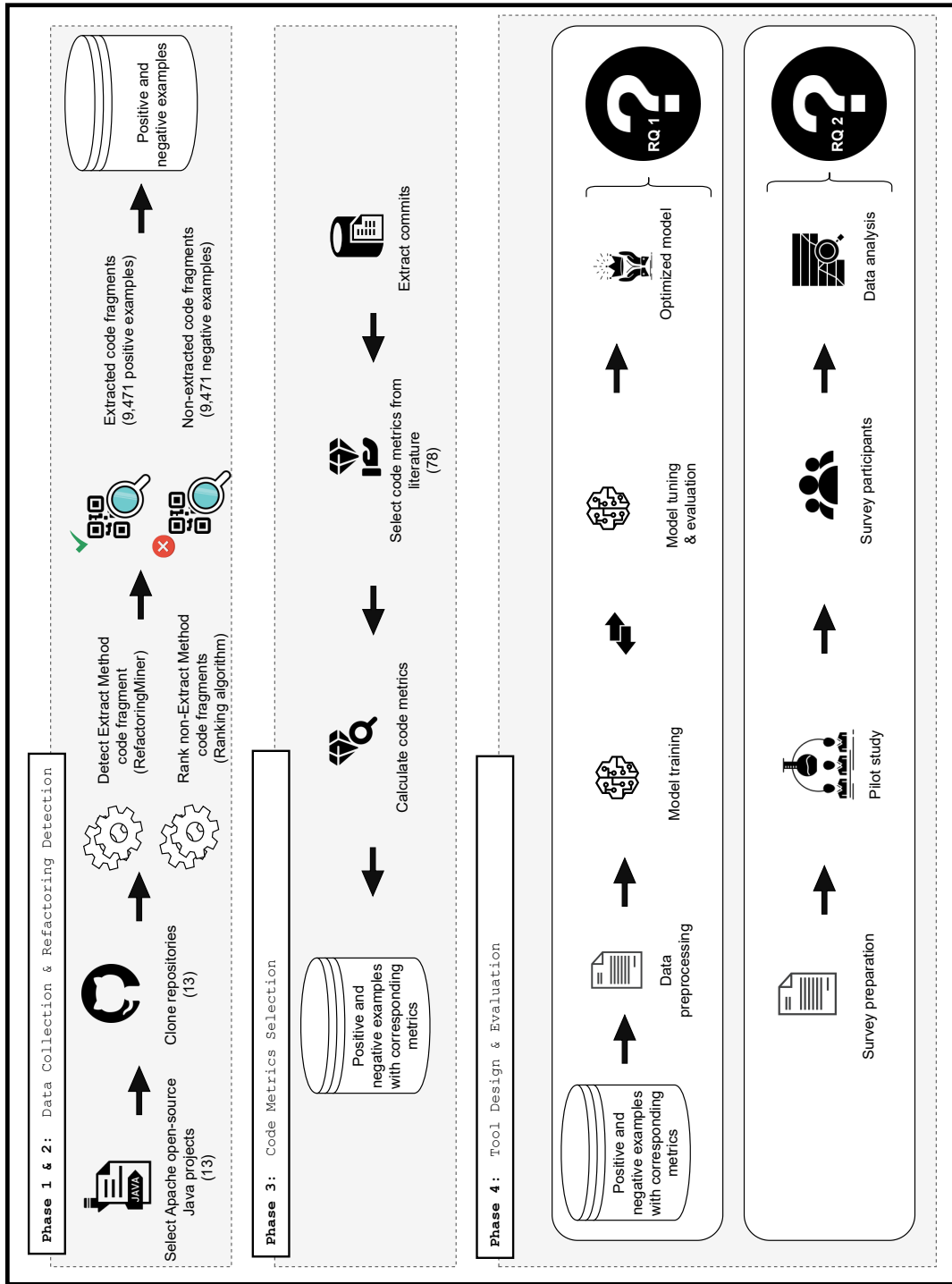


Figure 3: The overall pipeline of our work.

As mentioned in Section 3, various approaches rank candidate code fragments for method extraction. These techniques can be also used to discover the opposite: code fragments that are *less likely* to be extracted, *i.e.*, negative samples for our model. In our work, we use the ranking formula inspired by the work of Haas and Hummel [29], since the authors corroborated its validity by providing a human evaluation of the results. To collect the negative samples, we start with selecting all sequences of statements that are eligible to be extracted. Then, they are ranked according to a special scoring formula proposed by Haas and Hummel [29] that optimizes independent code metrics. From their formula, we used three terms: *statement length*, *nesting depth*, and *nesting area*. After ranking the fragments, according to the original work, the fragments that are more likely to be extracted will be located at the top of the list. In order to gather the ones that are less likely to be extracted, we skip the first 5% of fragments and select the bottom 95% of the list. We carried out this process for all 13 projects, then, to match the number of positive examples, we sampled 9,471 negative examples to constitute the final dataset.

4.3. Code Metrics Selection

After collecting positive and negative examples, we characterize them through various metrics. The goal of selecting metrics is to identify patterns in their values to allow distinguishing between the two classes of fragments. To do so, we gathered all the metrics that have been extensively used in previous studies [15, 16, 29, 47, 48, 49] and then removed all the redundant metrics to avoid generating features with similar values. In machine learning, duplicated values is a well-known problem that can have adverse effects on models and cause training algorithms to overfit or inflate classification metrics [5]. In total, we selected 78 metrics that can be divided into three main categories:

1. **Metrics that relate to the code fragment:** *e.g.*, length of the code fragment in symbols, `if` keyword count, etc.
2. **Metrics that relate to the enclosing method:** *e.g.*, length of the enclosing method in lines, etc.
3. **Coupling metrics:** *e.g.*, number of references to fields from the enclosing class in the code fragment, etc.

The list of metrics is available in our replication package [39].

4.4. Model Training

4.4.1. Dataset

To prepare the dataset, we label code fragments that underwent an *Extract Method* refactoring with “1”, and code fragments that are less likely to be extracted with “0”. Our feature vectors consist of the collected code metrics values, calculated for the positive and negative examples. In total, we annotated 9,471 code fragments as positive and 9,471 code fragments as negative.

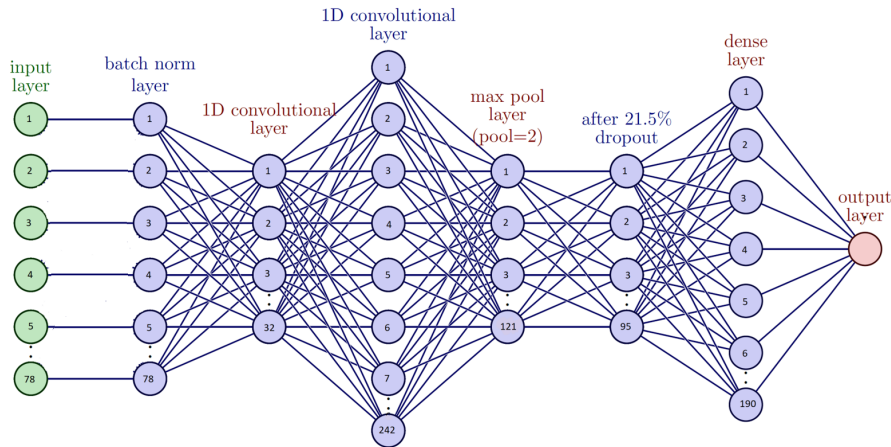


Figure 4: The architecture of the proposed CNN model.

4.4.2. CNN Binary Classification

We define the detection of an *Extract Method* opportunity as a binary classification problem. Our intended model takes a set of metrics as input, and uses them as features to learn patterns in their values that distinguish between duplicate code fragments that are more likely and less likely to be extracted. Since the input corresponds to 78 metrics, we chose to rely on CNNs for building our model. CNNs have been proven to achieve higher performance when classifying with a high number of features [50, 51].

Our CNN model consists of multiple layers of fully connected nodes, structured into a convolutional, deconvolutional, dense layers, and a dropout stage to prevent overfitting. The visualization of this architecture can be found in Figure 4. The input to the CNN is a vector of 78 metrics values that are batch-normalized to stabilize their distributions, through introducing additional network layers to control their mean and variance. The batch normalised inputs are fed to a convolution that reduces the feature space from 78 to 32. This convolution allows the model to adjust the weighting of the features, so the more significant ones are signal-boosted while less significant ones get suppressed (without being entirely dismissed). To do so, it takes a subset of the input vector, and applies a given weight to each element before summing them up and evaluating them using an activation function. We use the Rectified Linear Unit (ReLU) as the activation function for the convolutional layers. As an example of visualization, in Figure 5, a convolutional layer with a 3×1 filter multiplies the filter values by the learned weights and sums them up to distill dimensionality down to the most important features. During the training, the gradients were computed using a back-propagation algorithm.

Then, the convoluted data is fed to the deconvolutional layer. This is essentially the opposite of a convolutional layer where we take one value and multiply

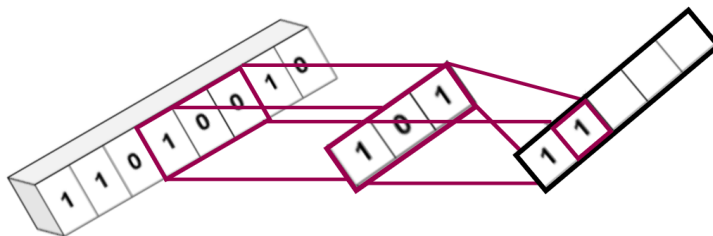


Figure 5: 1D convolution with a kernel size 3 and stride 3.

it by the layer’s weights to increase the dimensionality of our feature space. In this layer, the most important features are boosted with higher weights once the model is trained. Thus, the training of the model adjusts the weights to boost important features and suppresses the noise by reducing the feature space to what is most significant, then immediately expanding it again to make those significant features more prominent in the network. To further refine the weighting of our set of features, we use a max-pooling layer.

The max pool layer contains a filter of size 2 and takes the largest number in the said filter. This cuts our feature space in half and only considers the most prominent values. It is also computationally efficient since there are no weights associated with this max function. Since the training data is randomized, there is a possibility of the same class instances being consecutively fed during the training, which potentially biases the model into overpredicting that class. To avoid such overtraining bias, we added a dropout section to hide a subset of the nodes every epoch.

The final layer in the CNN is a dense one, in which each node receives the input from all nodes of the previous layer. Essentially, it maps the input to the corresponding output once the weights are adjusted properly (learned). The dense layer will output to a single node, making a probabilistic decision about whether a code fragment has to be refactored, given its original inputs (metric values). The CNN has been trained to minimize the binary cross-entropy loss function that calculates the distance between the model’s predicted label and the expected one.

4.4.3. Model Tuning

The purpose of this stage in the model construction process is to obtain the optimal set of classifier parameters that provide the minimized loss value; in other words, the objective of this task is to tune the hyperparameters. For our model, we optimized the batch size, number of nodes for the deconvolutional 1D layer, the dropout percentage, and the size of the dense layer. A randomized search was used to optimize these parameters [52]. For each set of parameters, 3 epochs were run to calculate the loss value. For batch size, we generated random values between 10 to 256. For sizes of layers, we generated random numbers between 16 and 256. For the dropout percentage, we tried random percentages

Table 3: Optimal parameter values for CNN.

Parameter	Value
n_epochs	3
batch size	20
convolutional layer	32
deconvolutional 1D layer	242
dropout	21.5
dense layer	190

between 0 and 50. Lastly, for the dense layer size, we tried random sizes between 5 and 256. The size of the convolutional layer was kept constant at 32. The parameters which had the minimal loss value after 3 epochs were considered as the best set of parameters (see Table 3). As a result, the hyperparameter tuning has set the optimal batch size to 20, the size of the deconvolutional 1D layer to 242, the dropout percentage to 21.5, and the size of the dense layer to 190.

5. Tool implementation

In this section, we describe the specific implementation of our proposed approach and the trained model. `ANTICOPYPASTER` is a plugin for IntelliJ IDEA, one of the most popular IDEs for Java. The plugin is composed of four main components.

Duplicate Detector. To detect duplicates, we use bag-of-words token-based clone detection [53]. This code similarity-based approach takes a given code fragment as input, then parses all methods inside the same file, so that each method is represented as tokens. The next step is to compute the similarity between the code fragment and methods via their abstracted token representation. This approach can detect an exact match, *i.e.*, when the code fragment is a substring of the method body. The bag-of-tokens similarity also takes into account minor changes in the pasted fragment, such as reordering the sequence of code, or renaming an identifier.

Since it is possible that a code fragment will be significantly edited soon after it is pasted, in order to avoid the immediate flagging of the pasted code as duplicate, and potentially interfering with the developer’s flow, we implement a *delay* and place the pasted code fragment in a queue. Then, two sanity checks are executed: we check whether the pasted fragment is Java code and whether it constitutes a correct syntactic statement. To do that, the plugin tries to build a PSI tree of the fragment. A PSI (Program Structure Interface) tree is a concrete syntax tree that is used in the IntelliJ Platform to represent the code [54]. If a PSI tree can be built and represents a valid statement, and if the duplicates still remain after the delay, the code fragment is passed to the *Code Analyzer*.

Code Analyzer. This component takes the duplicate fragment as input and uses its PSI representation to calculate the 78 metrics that we discussed

in Section 4. The code fragment, with its corresponding vector of metrics, constitute the input to the *Method Extractor*.

Method Extractor. This component takes as input the vector of metrics, and feeds it to the pre-trained model in order to make the binary decision of whether this code fragment is similar to the ones that have been previously refactored in the training dataset. If the binary classifier confirms the refactoring, then *Refactoring Launcher* is called.

Refactoring Launcher. This component starts with checking if the pasted code fragment could be extracted into a separate method without any compilation errors. If all checks pass, a notification is then enabled to appear in the bottom right corner of the editor, informing the developer that an *Extract Method* refactoring is recommended (see Figure 1). If the user responds to the tip, *Refactoring Launcher* passes the duplicate fragment as an input to the IDE’s built-in *Extract Method* API, and initiates the preview window. The user previews the code change and has the choice to either confirm the refactoring, while renaming the newly extracted method, or cancel the entire process.

6. Evaluation and Discussion

This section describes our empirical study aimed to evaluate the proposed approach, as well as its main results. We formulated two research questions:

- RQ1.** To what extent is the CNN model able to correctly detect the Extract Method refactoring compared to other models?
- RQ2.** Do users find ANTICOPYPASTER and the recommended *Extract Method* refactorings useful?

6.1. RQ1: Correctness

6.1.1. Approach

To address RQ1, we explore the ability of our CNN to accurately detect *Extract Method* refactoring opportunities. Furthermore, we compare the performance of our CNN model with four machine learning classifiers: Random Forest (RF), Support Vector Machine (SVM), Naive Bayes (NB), and Logistic Regression (LR). We selected these ML classifiers since their performance was competitive in similar binary classification problems [16, 36, 55, 56, 57, 58]. To evaluate the performance of the algorithms, we use out-of-sample bootstrap validation since this validation technique yields the best balance between the bias and variance in comparison to single-repetition holdout validation [59].

6.1.2. Results

The comparison between the classification algorithms is reported in Table 4. Based on our findings, the F-measure of CNN is 82%, higher than its competitors RF, SVM, NB, and LR, achieving 81%, 76%, 56%, and 71%, respectively. We conjecture that a proper conveyance of the semantics behind the source code

Table 4: The performance of different classifiers. Highest values are highlighted in **bold**.

Classifier	Precision	Recall	F-measure	PR-AUC
Random Forest	0.83	0.78	0.81	0.86
Support Vector Machine	0.78	0.74	0.76	0.86
Naive Bayes	0.72	0.46	0.56	0.72
Logistic Regression	0.73	0.70	0.71	0.79
Convolutional Neural Network	0.82	0.82	0.82	0.86

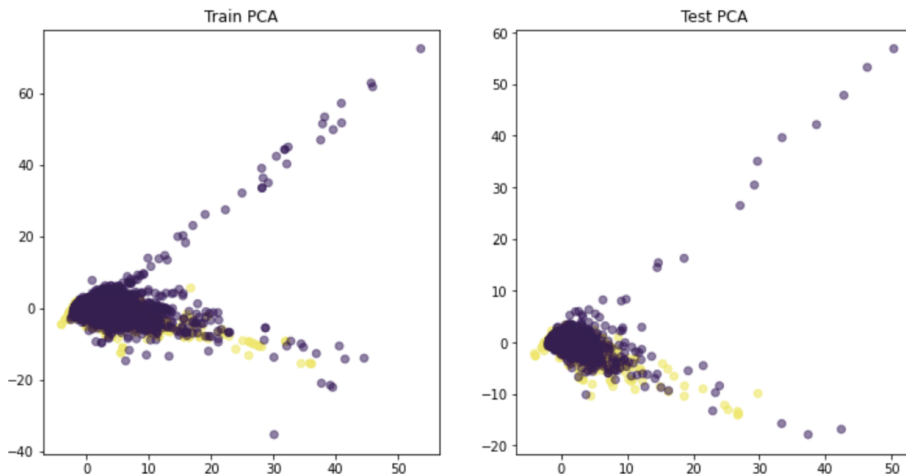


Figure 6: PCA plots between the positive (yellow) and negative (purple) classes for both training and testing data.

would have required complex feature engineering using neural network classification strategy rather than traditional machine algorithms. This observation has been also supported by previous studies that utilized deep learning to source code analysis [60, 61].

To further assess the efficiency of the identification of *Extract Method* refactoring, we used Isolation Forest (iForest) [62] to perform a one-class classification (anomaly detection) to evaluate the model’s ability to characterize the positive examples compared to noise. iForest detects the abnormal data in sample data by looking at how many branches are needed to classify a data point and judges whether the sample is isolated based on this branching.

To evaluate the effectiveness of iForest in *Extract Method* identification, we compare iForest with other models considered in Table 4. As can be seen, traditional classification methods obtain better performance and the Isolation Forest does not do better than random guessing (Accuracy = 0.49). Further, to better visualize the overlap of data points from the positive and negative examples, we performed the clustering on the data. Figure 6 illustrates the PCA plots for both training and testing data. We can see there is high overlap

Table 5: Statistical comparison between different classification algorithms (McNemar’s test and Odds Ratio). “*” captures the smallest OR among 10 statistical tests.

Comparison	p -value	OR
CNN vs RF	< 0.005	1.45*
CNN vs SVM	< 0.005	2.19
CNN vs NB	< 0.005	3.59
CNN vs LR	< 0.005	2.55
RF vs LR	< 0.005	1.95*
RF vs SVM	< 0.005	1.54*
RF vs NB	< 0.005	2.96*
SVM vs NB	< 0.005	2.59
SVM vs LR	< 0.005	1.77
LR vs NB	< 0.005	2.11

between the positive (yellow) and negative (purple) classes. This indicates that the one-class classification (anomaly detection) is insufficient to classify this data because our noise overlaps too much with the positive class. Note that we used PCA for dimensionality reduction so we can get a better visualization of the clustering, but K-means clustering was carried out in the entire 78-dimensional space as well and found the same overlap (it could not distinguish between the groups better than random guessing).

Since there is no model that outperforms all the others in both precision and recall, the choice of the model can become the decision of the practitioner who is adopting the tool. For this problem, the lack of precision indicates potential recommendations of code that is not necessarily worth refactoring (*e.g.*, less complex), which would result in creating one extra method. At the same time, the lack of recall indicates missing opportunities of recommending code that is worth refactoring. We opted to deploy CNN because it not only provides the highest recall, but also delivers the best trade-off in terms of F-measure.

In order to statistically compare the performance of the classification algorithms, we use the McNemar test [63] and the Odds Ratio (OR) effect size, where an OR greater than 1 indicates that the first technique outperforms the second one. We compared the performance of each pair of the classifiers by running statistical tests 10 times. Since multiple comparisons are performed, we adjusted the p -values using the Bonferroni correction [64]. In this context, we define the following null hypothesis H_0 for each test: *there is no statistically significant difference between the performance of two algorithms* and our alternate hypothesis H_1 is: *there is a statistically significant difference between the performance of the two algorithms*. Table 5 describes the p -values and OR obtained for each test. We tested the hypotheses at a 5% significance level and used an adjusted alpha value (*i.e.*, 0.005) for the comparisons. As shown in the table, the McNemar test results show that null hypothesis is rejected as there are statistically significant differences (p -value < 0.05/10) in the performance

of the algorithms, with the CNN having 2.19, 3.59, and 2.55 more chances to correctly recommend *Extract Method* refactoring opportunities than SVM, NB, and LR, respectively, and at least 1.45 more chances than RF. The table also reveals that the performance of the other two classifiers are statistically significant with OR greater than 1.

Summary: *CNN outperforms traditional machine learning algorithms, having at least 1.45 more chances to recommend proper Extract Method refactoring opportunities.*

6.2. RQ2: Usefulness

6.2.1. Approach

To assess the usefulness of ANTICOPYPASTER, we performed an external validation by involving 109 participants from the Rochester Institute of Technology, Stevens Institute of Technology, and ETS Montreal. All participants volunteered to participate in the experiment. Of the set of invited participants, 72 developers accepted and participated in the survey (yielding a response rate of 66.1%, which is considered high for software engineering research [65]), and 50 out of 72 participants confirmed that they executed the plugin. Table 6 summarizes the developers’ experience. 88.9% of the participants have more than 1 year of coding experience. Also, 50% of the participants have between 1 to more than 10 years of development in either industry or open source.

As suggested by Kitchenham and Pfleeger [66], we constructed the survey to use a 5-point ordered response scale (‘Likert scale’) questions, 7 open-ended questions, and 8 multiple choice questions with an optional ‘Other’ category, allowing the respondents to share thoughts not mentioned in the list. The survey consisted of 21 questions. The first part of the survey includes questions about the demographics of participants. Next, we asked about the usefulness, usability, and functionality of the proposed idea and the plugin. We provide participants with a video demonstrating how to use the plugin, along with a link to download it.

Furthermore, we asked the participants to potentially share with us the log of the plugin events. This log tracks the usage of the tool. Such information

Table 6: Professional development experience of the participants in years.

Years of Professional Experience	Professional Experience (%)	Programming Experience (%)
< 1	36 (50%)	8 (11.11%)
1-3	26 (36.12%)	24 (33.33%)
4-6	5 (6.94%)	25 (34.73%)
7-10	0 (0%)	9 (12.5%)
10+	5 (6.94%)	6 (8.33%)

provides us with more detailed information about the plugin’s usage over time. The log does not track any personal information related to the user or the source code. The log features the following events:

- **copyCount.** Action of copying a code fragment.
- **pasteCount.** Action of pasting a code fragment.
- **notificationCount.** Appearance of a notification about a potential refactoring opportunity.
- **extractMethodAppliedCount.** Acceptance of the recommendation by clicking on the notification and applying the refactoring.
- **extractMethodCanceledCount.** Cancellation, when the notification was clicked, but the process was canceled.

To share the log with us, the participants need to upload an XML file, which is auto-generated by the plugin, at the end of the survey.

We analyzed the responses to open-ended questions to create a comprehensive high-level list of themes by adopting a thematic analysis approach based on guidelines provided by Cruzes *et al.* [67]. Thematic analysis is one of the most used methods in Software Engineering literature [8, 68]. This is a technique for identifying and recording patterns (or “themes”) within a collection of descriptive labels, which we call “codes”. For each response, we proceeded with the analysis using the following steps: *i*) Initial reading of the survey responses; *ii*) Generating initial codes (*i.e.*, labels) for each response; *iii*) Translating codes into themes, sub-themes, and higher-order themes; *iv*) Reviewing the themes to find opportunities for merging; *v*) Defining and naming the final themes, and creating a model of higher-order themes and their underlying evidence. The above-mentioned steps were performed independently by two authors. One author performed the labeling of responses to open-ended questions independently from the other author, who was responsible for reviewing the currently drafted themes. Then, the authors met and refined the themes. It is important to note that the approach is not a single-step process. As the codes were analyzed, some of the first cycle codes were subsumed by other codes, relabeled, or dropped altogether. As the two authors progressed in the translation to themes, there was some refinement and reclassification of data into different or new codes.

6.2.2. Results

We started the survey by asking participants how often they use the *Extract Method* refactoring feature in the IDE. Figure 7 shows the breakdown of the answers. It can be seen that *Extract Method* refactoring is not very frequently used in the IDE as 55.7% of developers indicated that they never used the *Extract Method* refactoring feature, while 34.3% said they used it several times a year. Only 4.3% of the users said that they extracted methods approximately once per month, and just 5.7% perform *Extract Method* several times a month. It appears that not many developers have used the IDE to apply the *Extract*

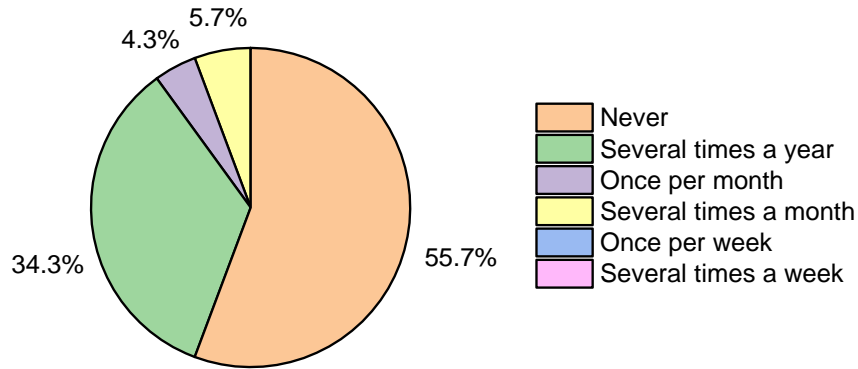


Figure 7: Answers to the question “How often do you use the *Extract Method* refactoring feature in the IDE?”

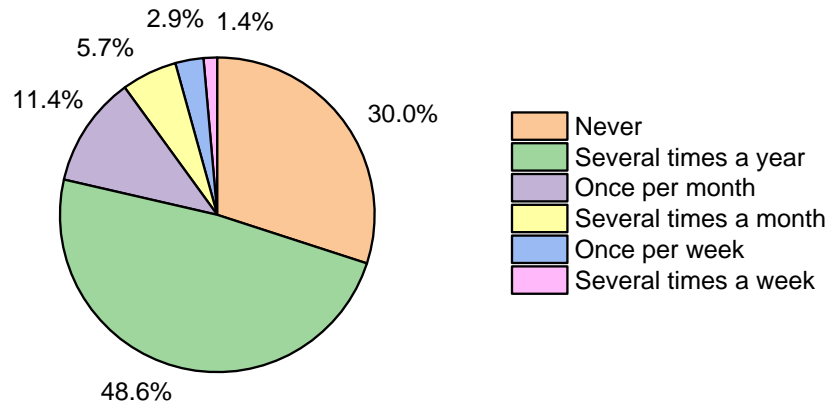


Figure 8: Answers to the question “How often have you refactored duplicate code / code clones?”

Method refactoring. This observation corroborates the finding of a recent survey on refactoring [27] that shows significantly fewer developers using the IDE feature for the *Extract* refactorings, especially when compared to the *Rename* feature. The fact that *Extract* is not as intuitive as *Rename* represents a significant challenge to the ongoing research and studies that develop recommendation algorithms to extract code at any level (method, class, package, etc.). This is one of the main motivations behind the design of our solution. We believe that our tool can further support developers to practice this type of refactoring, or at least raise their awareness of its existence regardless.

Table 7: The results of tracking features in collected logs.

Event	Count
notificationCount	63
extractMethodAppliedCount	59
extractMethodCanceledCount	0
copyCount	350
pasteCount	379

Concerning the frequency of refactoring specifically duplicated code, Figure 8 depicts that almost half of the developers answered that they refactor duplicated code several times a year. 30% of the respondents said that they never refactor duplicated code. 11.4% of the respondents said that they refactor duplicated code once per month. We conjecture that despite the existence of automated code clone detectors (*e.g.*, CCFinder [69]), these tools might lack integration, as developers acknowledge the existence of code clones but they do not have a preference on how to refactor them using automated tools. Besides, several studies have also shown that not all duplicates are harmful to the code [70].

Table 7 reveals the sum of metric numbers, extracted from the log files shared by the participants. Our first observation is that only 17% of the pasted code was evaluated as worth refactoring by the model. This can be due to the CNN being *picky* by nature, which explains that 93% of its recommendations were actually accepted by the participants. Since the number of cancelled refactorings is zero, we conjecture that the 7% of cases where the model recommends a refactoring that was not applied, can be due to developers simply ignoring the notification shown in their screen. This is another advantage of the pop-up notification — it has a minimal disturbance on the developers when they are busy.

In Table 8, we report the main thoughts, comments, and suggestions about the overall impression of the usefulness, usability, and functionality of the proposed idea, in accordance with the conducted labeling. Table 8 also presents samples of the participants’ comments to illustrate their impressions.

Usefulness. Generally, the respondents found the tool to be useful in regard to four main aspects: effort, quality, automation, and awareness. The majority of the participants commented that the proposed idea saves time and effort for developers who would not have to examine and refactor duplicates manually. Other participants communicated that reducing redundant code assists in increasing its readability and efficiency while reducing its complexity, which helps improve overall code quality. A moderate subset of developers revealed that the tool’s ability to identify duplicates within a file and reduce them to a single method allows users to only correct errors in a specific location pro-actively and automatically. Further, some developers commented that the tool aids developers in identifying their redundancy when updating a source file that they are not familiar with. Another noteworthy point mentioned was that the tool helps less experienced and novice coders in writing well-structured code.

Table 8: Developer’s insight about the usefulness, usability, and functionality of the tool.

Category	Sub-category	Example (Excerpts from a related survey response)
	Effort	<p>“I feel it should be useful as this saves time and effort put but the developer who is developing a software.”</p> <p>“A plugin that automatically extracts methods from duplicated code sounds very useful for the sake of reducing complexity and improving readability of code. In my own coding experience, I’ve definitely had times where I recognized that I was duplicating code, but the means by which the method should be extracted wasn’t immediately obvious.”</p>
	Quality	<p>“For small code projects, I don’t think it would be as useful, but for large code projects it probably would be. A suggestion, to make the tool even more useful, would be to allow for static analysis of large code bases (where the tool would be most useful) that have already been coded, then recommending extract method refactorings. Companies with large messy code bases would love to have a tool like that to clean up all their code duplicates without having to meticulously go through it manually.”</p>
Usefulness	Automation	<p>“As a developer, when working with large files and thousands of code lines we tend to repeat some of the functions already present in the code which results to duplicity and less efficient. This tool is very helpful and makes the developer aware of their approach and not to make the mistakes of repeating the code again and again.”</p>
	Awareness	<p>“I guess adding a notification was great touch to the project, but I would rather have an option to enable or disable the notification. Plus i would also like it to highlight the duplications in the code itself rather then a notification.”</p>
Usability	Notification	<p>“I would remove the delay and just put a UI notification in bottom that refactoring is available. If the user tries to edit the code, then the notification disappears.”</p>
	Delay	<p>“I would remove the dialog as it can interrupt a users workflow. I would set it as an alert in the UI such that it doesn’t interrupt a user but rather notifies them and gives the ability to make the fix or easily continue developing with the duplicate code.”</p>
	Dialog box	<p>“A change in the operation of the plugin would be showcase a preview of the code itself instead of just the signature preview. Seeing the code change as you name the new function would be a nice change.”</p>
	Preview	<p>“Perhaps give a default name to the function, instead of asking user to enter manually.”</p>
	Recommendation	<p>“Instead of making a pop-up appear whenever duplicated code is detected, I would prefer to have an option to scan for instances of duplicated code and have it automatically refactored after that.”</p>
Functionality	Refactoring	

Usability. Based on the feedback provided by the respondents to the survey, the key areas in usability related to the notification, the delay, the dialog box, and the preview. The notification and the delay aspects of the tool were the primary areas identified for change or improvement. The main suggestions about the pop-up notification were driven by developers' personal preferences of how notifications should be shown. For example, some respondents viewed the current pop-up notification (Figure 1) as something that could be distracting, and suggested other options. These options included highlighting the duplicate code, icon flashing at the bottom taskbar, or just using a warning message. Some responses stated that the users of the tool should have more control to set the delay time to wait before triggering the suggestion. Other responses also stated that triggering the tool after a save operation would be less disruptive. While many of these ideas can be added to tailor the tool to developers' preferences, we find the suggestion of highlighting duplicate code to be the most practical and we plan to implement it in the future.

Functionality. From the participants' feedback, we have also extracted suggestions to improve the tool's functional features. One proposed feature was to recommend the proper method name for the extracted code, based on its functionality. Moreover, since the participants found the tool to be useful, they suggested to support additional refactorings to remove duplicates at the class level (*i.e.*, *Extract Class*). Participants also suggested scanning all project files for duplicates (instead of just the current one, as it is now implemented) and then interactively suggest their refactoring one by one. We found this suggestion to be particularly interesting, since it does not match the rationale of our solution. Implementing this suggestion would require training with positive samples that were exclusively executed to remove duplicates. It would also require taking into account the imbalanced nature of the codebase by creating a significantly higher number of negative samples.

50 out of 72 participants confirmed that they executed the plugin. We asked them about their satisfaction with various aspects of the tool. Figure 9 presents an overview of their answers. With respect to the tool setup, most of the respondents (43 participants) reported that they are satisfied with the tool. Regarding the tool documentation, the majority of the respondents agreed that the documentation is useful; only 4 participants were unsatisfied. For the ease of use aspect, a larger group (41 participants) was satisfied. Several participants found that the tool is not easy to use, so we will work on improving its usability. Concerning the execution time, 39 participants were happy with it. For the amount of pop-up notifications, the majority of respondents (26 participants) agreed that the amount of pop-up notifications is acceptable, although there were a few participants that remained neutral. There were also a few participants who were not happy with the amount of pop-up notifications, and we are planning on improving this aspect of the tool in the future.

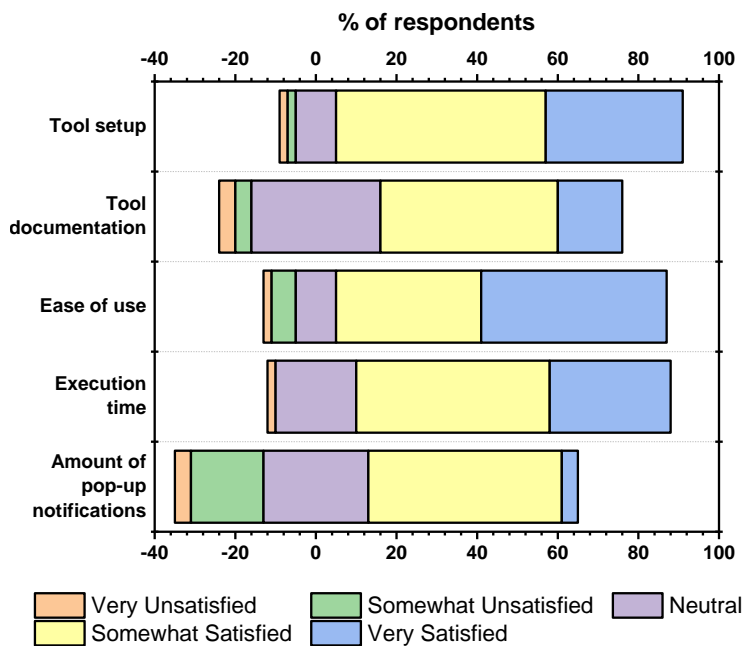


Figure 9: Participants' satisfaction with various aspects of the ANTI-COPYPASTER tool.

Summary: Overall, the participants were satisfied with the plugin and rated its various aspects highly. The aspects of the tool that can be improved are the UI of highlighting and more settings for users' preferences.

7. Threats to Validity

In this section, we identify potential threats to the validity of our approach and our experiments as discussed in the work of Ampatzoglou *et al.* [71].

Internal Validity. Our analysis is mainly threatened by the accuracy of the refactoring mining tool because the tool may miss the detection of some refactorings. However, previous studies [45, 46] report that RefactoringMiner has high precision and recall scores (99.8% and 95.8%, respectively) compared to other state-of-the-art refactoring detection tools, which gives us confidence in using this tool. Also, since the developers have the option to just "skip" the notification, there were no cases when developers started the process and then rejected it. It is also possible that developers undid the refactoring after conducting it.

Construct Validity. Collecting positive examples for our model requires not only finding the refactoring itself but navigating to the previous commit to see the context, from where the method was extracted. In some cases, the detection of the previous commit might not be straightforward because there are

several branches in the repository. We found several such cases when manually checking the collected data. As for the collection of negative examples, it was done using the ranking algorithm inspired by the one of Haas and Hummel [29], so our work inherits any limitation associated with that algorithm. Concerning the completeness and correctness of our interpretation of the open-ended responses within the survey, we did not extensively discuss all responses because some of them are open to various interpretations, and we need further follow-up surveys or interviews to clarify them.

External Validity. Our analysis was performed on 13 mature open-source Java projects belonging to the Apache ecosystem that are varied in size, contributors, number of commits and refactorings. However, we cannot claim the generality of our observations to projects written in other programming languages or belonging to other ecosystems. Further investigation of even more projects is needed to mitigate this threat. Regarding the study participants, the majority of our participants involved students, with some of them having industrial experience. To avoid bias in the experiment, we make providing feedback anonymous and not mandatory, to increase the magnitude of tool usage experience. Although feedback was optional, 95% of students have completed it after removing arbitrary submissions. As future work, we plan to perform another round of external validation with professional software engineers in industry to hear their perception.

8. Conclusion

Recommending *Extract Method* refactoring opportunities is of paramount importance to the research community and industry. Although a plethora of studies have utilized a variety of approaches to identify *Extract Method* refactoring, recommending this refactoring type without interfering with developers' workflow remains largely unexplored. In this study, we proposed ANTICOPYPASTER as an IntelliJ IDEA plugin, and experimented with machine learning models in order to increase the adoption and usage of the *Extract Method* refactoring while maintaining the workflow of a developer. Our results reveal that machine learning models are able to recommend *Extract Method* refactoring opportunities as soon as code duplicates are introduced in the IDE, and the participants were satisfied with the ANTICOPYPASTER tool.

In particular, the proposed CNN demonstrated an F-measure of 0.82 and outperformed other machine learning models. In the survey, we discovered that a majority of developers carry out *Extract Method* refactorings very rarely or never at all, so the proposed pro-active pipeline for their recommendation can also fulfill the educational needs. Overall, the participants rated various aspects of the plugin highly, while also providing valuable ideas for future development. In particular, we would like to implement the highlighting of refactorable code duplicates in the editor and give the users more control over various aspects of the plugin for customization.

References

- [1] C. K. Roy, J. R. Cordy, R. Koschke, Comparison and evaluation of code clone detection techniques and tools: A qualitative approach, *Science of computer programming* 74 (7) (2009) 470–495.
- [2] B. Hu, Y. Wu, X. Peng, J. Sun, N. Zhan, J. Wu, Assessing code clone harmfulness: Indicators, factors, and counter measures, in: *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2021, pp. 225–236.
- [3] P. Thongtanunam, W. Shang, A. E. Hassan, Will this clone be short-lived? Towards a better understanding of the characteristics of short-lived clones, *Empirical Software Engineering* 24 (2) (2019) 937–972.
- [4] C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, J. Vitek, Déjàvu: a map of code duplicates on GitHub, *Proceedings of the ACM on Programming Languages* 1 (OOPSLA) (2017) 1–28.
- [5] M. Allamanis, The adverse effects of code duplication in machine learning models of code, in: *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2019, pp. 143–153.
- [6] R. Fanta, V. Rajlich, Removing clones from the code, *Journal of Software Maintenance: Research and Practice* 11 (4) (1999) 223–243.
- [7] M. Fowler, *Refactoring: Improving the design of existing code*, Addison-Wesley Professional, 2018.
- [8] D. Silva, N. Tsantalis, M. T. Valente, Why we refactor? Confessions of GitHub contributors, in: *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016*, ACM, 2016, pp. 858–870.
- [9] E. Murphy-Hill, A. P. Black, Breaking the barriers to successful refactoring: Observations and tools for Extract Method, in: *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 421–430.
- [10] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, K. Deb, Multi-criteria code refactoring using search-based software engineering: An industrial case study, *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25 (3) (2016) 1–53.
- [11] W. Mkaouer, M. Kessentini, A. Shaout, P. Koligheu, S. Bechikh, K. Deb, A. Ouni, Many-objective software remodularization using NSGA-III, *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24 (3) (2015) 1–45.
- [12] T. Kanemitsu, Y. Higo, S. Kusumoto, A visualization method of program dependency graph for identifying Extract Method opportunity, in: *Proceedings of the 4th Workshop on Refactoring Tools*, 2011, pp. 8–14.
- [13] G. Bavota, A. De Lucia, A. Marcus, R. Oliveto, Automating Extract Class refactoring: An improved method and its evaluation, *Empirical Software Engineering* 19 (6) (2014) 1617–1664.
- [14] S. Xu, A. Sivaraman, S.-C. Khoo, J. Xu, GEMS: An Extract Method refactoring recommender, in: *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, 2017, pp. 24–34.
- [15] R. Yue, Z. Gao, N. Meng, Y. Xiong, X. Wang, J. D. Morgenthaler, Automatic clone recommendation for refactoring based on the present and the past, in: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2018, pp. 115–126.

- [16] M. Aniche, E. Maziero, R. Durelli, V. H. Durelli, The effectiveness of supervised machine learning algorithms in predicting software refactoring, *IEEE Transactions on Software Engineering* 48 (4) (2020) 1432–1450.
- [17] N. Yoshida, S. Numata, E. Choiz, K. Inoue, Proactive clone recommendation system for Extract Method refactoring, in: 2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR), IEEE, 2019, pp. 67–70.
- [18] J. P. S. Alcocer, A. S. Antezana, G. Santos, A. Bergel, Improving the success rate of applying the Extract Method refactoring, *Science of Computer Programming* 195 (2020) 102475.
- [19] K. Hotta, Y. Sano, Y. Higo, S. Kusumoto, Is duplicate code more frequently modified than non-duplicate code in software evolution? An empirical study on open source software, in: Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE), 2010, pp. 73–82.
- [20] Y. Higo, S. Kusumoto, K. Inoue, A metric-based approach to identifying refactoring opportunities for merging code clones in a Java software system, *Journal of Software Maintenance and Evolution: Research and Practice* 20 (6) (2008) 435–461.
- [21] L. Yang, H. Liu, Z. Niu, Identifying fragments to be extracted from long methods, in: 2009 16th Asia-Pacific Software Engineering Conference, IEEE, 2009, pp. 43–49.
- [22] R. Morales, Z. Soh, F. Khomh, G. Antoniol, F. Chicano, On the use of developers’ context for automatic refactoring of software anti-patterns, *Journal of systems and software* 128 (2017) 236–251.
- [23] O. Tiwari, R. Joshi, Identifying Extract Method Rrefactorings, in: 15th Innovations in Software Engineering Conference, 2022, pp. 1–11.
- [24] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, G. Antoniol, An exploratory study of the impact of antipatterns on class change- and fault-proneness, *Empirical Software Engineering* 17 (3) (2012) 243–275.
- [25] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, Do they really smell bad? A study on developers’ perception of bad code smells, in: 2014 IEEE International Conference on Software Maintenance and Evolution, IEEE, 2014, pp. 101–110.
- [26] F. Palomba, G. Bavota, M. D. Penta, F. Fasano, R. Oliveto, A. D. Lucia, On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation, *Empirical Software Engineering* 23 (3) (2018) 1188–1221.
- [27] Y. Golubev, Z. Kurbatova, E. A. AlOmar, T. Bryksin, M. W. Mkaouer, One thousand and one stories: A large-scale survey of software refactoring, in: 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2021, pp. 1303–1313.
- [28] Example of Java code, <https://www.cs.utexas.edu/~scottm/cs307/javacode/codeSamples/IntListVer2.java>, (Accessed on 01/02/2023).
- [29] R. Haas, B. Hummel, Deriving Extract Method refactoring suggestions for long methods, in: International Conference on Software Quality, Springer, 2016, pp. 144–155.
- [30] K. Maruyama, Automated method-extraction refactoring by using block-based slicing, in: Proceedings of the 2001 symposium on Software reusability: putting software reuse in context, 2001, pp. 31–40.
- [31] N. Tsantalis, A. Chatzigeorgiou, Identification of Extract Method refactoring opportunities for the decomposition of methods, *Journal of Systems and Software* 84 (10) (2011) 1757–1782.

- [32] T. Sharma, Identifying extract-method refactoring candidates automatically, in: Proceedings of the Fifth Workshop on Refactoring Tools, 2012, pp. 50–53.
- [33] D. Silva, R. Terra, M. T. Valente, Recommending automated Extract Method refactorings, in: Proceedings of the 22nd International Conference on Program Comprehension, 2014, pp. 146–156.
- [34] D. Silva, R. Terra, M. T. Valente, JExtract: An eclipse plug-in for recommending automated Extract Method refactorings, arXiv preprint arXiv:1506.06086.
- [35] M. Shahidi, M. Ashtiani, M. Zakeri-Nasrabadi, An automated Extract Method refactoring approach to correct the long method code smell, *Journal of Systems and Software* 187 (2022) 111221.
- [36] D. van der Leij, J. Binda, R. van Dalen, P. Vallen, Y. Luo, M. Aniche, Data-driven Extract Method recommendations: A study at ING, in: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2021, pp. 1337–1347.
- [37] E. A. Alomar, A. Ivanov, Z. Kurbatova, Y. Golubev, M. W. Mkaouer, A. Ouni, T. Bryksin, L. Nguyen, A. Kini, A. Thakur, AntiCopyPaster: extracting code duplicates as soon as they are introduced in the IDE, in: 37th IEEE/ACM International Conference on Automated Software Engineering, 2022, pp. 1–4.
- [38] E. A. Alomar, A. Ivanov, Z. Kurbatova, Y. Golubev, M. W. Mkaouer, A. Ouni, T. Bryksin, L. Nguyen, A. Kini, A. Thakur, AntiCopyPaster on GitHub (2023). URL <https://github.com/JetBrains-Research/anti-copy-paster>
- [39] E. A. Alomar, A. Ivanov, Z. Kurbatova, Y. Golubev, M. W. Mkaouer, A. Ouni, T. Bryksin, L. Nguyen, A. Kini, A. Thakur, Replication Package (2023). URL <https://zenodo.org/record/7428835>
- [40] A. Krizhevsky, I. Sutskever, G. E. Hinton, ImageNet classification with deep convolutional neural networks, *Communications of the ACM* 60 (6) (2017) 84–90.
- [41] N. Tsantalis, A. Chatzigeorgiou, Identification of Extract Method refactoring opportunities, in: 2009 13th European Conference on Software Maintenance and Reengineering, IEEE, 2009, pp. 119–128.
- [42] S. Charalampidou, A. Ampatzoglou, A. Chatzigeorgiou, A. Gkortzis, P. Avgeriou, Identifying Extract Method refactoring opportunities based on functional relevance, *IEEE Transactions on Software Engineering* 43 (10) (2016) 954–974.
- [43] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, S. Panichella, How the Apache community upgrades dependencies: An evolutionary study, *Empirical Software Engineering* 20 (5) (2015) 1275–1317.
- [44] M. Di Penta, G. Bavota, F. Zampetti, On the relationship between refactoring actions and bugs: A differentiated replication, in: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020, pp. 556–567.
- [45] N. Tsantalis, M. Mansouri, L. Eshkevari, D. Mazinianian, D. Dig, Accurate and efficient refactoring detection in commit history, in: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), IEEE, 2018, pp. 483–494.
- [46] N. Tsantalis, A. Ketkar, D. Dig, RefactoringMiner 2.0, *IEEE Transactions on Software Engineering* 48 (3) (2020) 930–950.

- [47] M. Caulo, G. Scanniello, A taxonomy of metrics for software fault prediction, in: 2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), IEEE, 2020, pp. 429–436.
- [48] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, M. Ó Cinnéide, Recommendation system for software refactoring using innovization and interactive dynamic optimization, in: Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, 2014, pp. 331–336.
- [49] M. D’Ambros, M. Lanza, R. Robbes, Evaluating defect prediction approaches: A benchmark and an extensive comparison, *Empirical Software Engineering* 17 (4) (2012) 531–577.
- [50] J. Gu, Z. Wang, J. Kuen, L. Ma, A. Shahroudy, B. Shuai, T. Liu, X. Wang, G. Wang, J. Cai, et al., Recent advances in convolutional neural networks, *Pattern Recognition* 77 (2018) 354–377.
- [51] C. Liu, C. Gao, X. Xia, D. Lo, J. Grundy, X. Yang, On the reproducibility and replicability of deep learning in software engineering, *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31 (1) (2021) 1–46.
- [52] J. Bergstra, Y. Bengio, Random search for hyper-parameter optimization, *Journal of machine learning research* 13 (2).
- [53] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, C. V. Lopes, SourcererCC: Scaling code clone detection to big-code, in: Proceedings of the 38th International Conference on Software Engineering, 2016, pp. 1157–1168.
- [54] Z. Kurbatova, Y. Golubev, V. Kovalenko, T. Bryksin, The intellij platform: a framework for building plugins and mining software data, in: 2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW), IEEE, 2021, pp. 14–17.
- [55] E. A. AlOmar, J. Liu, K. Addo, M. W. Mkaouer, C. Newman, A. Ouni, Z. Yu, On the documentation of refactoring types, *Automated Software Engineering* 29 (1) (2022) 1–40.
- [56] E. A. AlOmar, A. Peruma, M. W. Mkaouer, C. Newman, A. Ouni, M. Kessentini, How we refactor and how we document it? On the use of supervised machine learning algorithms to classify refactoring documentation, *Expert Systems with Applications* 167 (2021) 114176.
- [57] E. A. AlOmar, M. W. Mkaouer, A. Ouni, Toward the automatic classification of self-affirmed refactoring, *Journal of Systems and Software* 171 (2021) 110821.
- [58] S. Levin, A. Yehudai, Boosting automatic commit classification into maintenance activities by utilizing source code changes, in: 13th International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE, 2017, pp. 97–106.
- [59] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, K. Matsumoto, An empirical comparison of model validation techniques for defect prediction models, *IEEE Transactions on Software Engineering* 43 (1) (2016) 1–18.
- [60] F. Zampetti, A. Serebrenik, M. Di Penta, Automatically learning patterns for self-admitted technical debt removal, in: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2020, pp. 355–366.
- [61] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, D. Shybyanyk, An empirical study on learning bug-fixing patches in the wild via neural machine translation, *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28 (4) (2019) 1–29.

- [62] F. T. Liu, K. M. Ting, Z.-H. Zhou, Isolation forest, in: 2008 Eighth IEEE international conference on data mining, IEEE, 2008, pp. 413–422.
- [63] T. G. Dietterich, Approximate statistical tests for comparing supervised classification learning algorithms, *Neural computation* 10 (7) (1998) 1895–1923.
- [64] P. Dalgaard, Analysis of variance and the Kruskal-Wallis test, *Introductory Statistics with R* (2002) 111–127.
- [65] E. Smith, R. Loftin, E. Murphy-Hill, C. Bird, T. Zimmermann, Improving developer participation rates in surveys, in: 2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE), IEEE, 2013, pp. 89–92.
- [66] B. A. Kitchenham, S. L. Pfleeger, Personal opinion surveys, in: *Guide to advanced empirical software engineering*, Springer, 2008, pp. 63–92.
- [67] D. S. Cruzes, T. Dyba, Recommended steps for thematic synthesis in software engineering, in: 2011 international symposium on empirical software engineering and measurement, IEEE, 2011, pp. 275–284.
- [68] E. A. AlOmar, M. Chouchen, M. W. Mkaouer, A. Ouni, Code review practices for refactoring changes: an empirical study on OpenStack, in: *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 689–701.
- [69] T. Kamiya, S. Kusumoto, K. Inoue, CCFinder: A multilinguistic token-based code clone detection system for large scale source code, *IEEE transactions on software engineering* 28 (7) (2002) 654–670.
- [70] K. Mens, S. Nijssen, H.-S. Pham, The good, the bad, and the ugly: Mining for patterns in student source code, in: *Proceedings of the 3rd International Workshop on Education through Advanced Software Engineering and Artificial Intelligence*, 2021, pp. 1–8.
- [71] A. Ampatzoglou, S. Bibi, P. Avgeriou, M. Verbeek, A. Chatzigeorgiou, Identifying, categorizing and mitigating threats to validity in software engineering secondary studies, *Information and Software Technology* 106 (2019) 201–230.