Calibrating Deep Learning-based Code Smell Detection using Human Feedback

Himesh Nanadani, Mootez Saad, Tushar Sharma
Dalhousie University, Canada
{hnandani, mootez, tushar}@dal.ca

***Abstract*—Code smells are inherently subjective in nature. Software developers may have different opinions and perspectives on smelly code. While many attempts have been made to use deep learning-based models for code smell detection, they fail to consider each developer's subjective perspective while detecting smells. Ignoring this aspect defies the purpose of using deep learning-based smell detection methods because the models are not customized to the developer's context. This paper proposes a method that considers human feedback to account for such subjectivity. Towards this, we created a plugin for IntelliJ IDEA and developed a container-based web-server to offer services of our baseline deep learning model. The setup allowed developers to see code smells within the IDE and provide feedback. Using this setup, we conducted a controlled experiment with 14 participants divided into experimental and control groups. In the first round of our experiment, we show code smells predicted using the baseline deep learning model and collect feedback from the participants. In the second round, we fine-tune the model based on the experimental group's feedback and reevaluate its performance before and after adjustment. Our results show that using such calibration improves the performance of the smell detection model by 15.49% in F1 score on average across the participants of the experimental group. Our work carries implications for both researchers and practitioners. Practitioners can apply our approach to enhance the quality of their code in day-to-day development activities, aligning it with their own code smell definitions. Furthermore, software engineering researchers can leverage this study to adopt analogous approaches for addressing similar issues, including code review.**

***Index Terms*—Code smell detection, human feedback, deep learning.**

I. Introduction

Software development is a complex and error-prone process that demands attention to detail and continuous improvements for the production of high-quality code. As projects evolve, accommodating changes becomes increasingly challenging unless the development team puts extra efforts to maintain their code quality [1]. Code smells [2], [3] provide valuable insights into potential design flaws and maintainability problems. Detecting code smells early in the development process enables developers to address them before they become more onerous and expensive to rectify [4]

Traditional methods of code smell identification rely on metrics and heuristics-based analysis of source code [3], [5]. Such approaches produce a significant number of false positives since they are based on fixed rules and do not consider the context and subjectivity involved in smell detection [3]. Code smells can be subjective, with interpretations and perceptions varying among individual developers. This subjectivity introduces complexities in accurately detecting and addressing code smells. The research community has proposed many machine learning (ML), including deep learning (DL) -based approaches [6] [7] [8] to tackle the challenge posed by smells' subjectivity. However, applying a DL approach to detect code smells based on the dataset created by multiple participants misses subjectivity inherent in identifying smells at the individual developer or a team-level [9]. Hence, a generic DL-based approach to detect code smells will not be as effective because different developers may not agree about the presence of smells in the same code snippet. To overcome this challenge, our research focuses on *user-specific subjective analysis*, aiming to customize smell detection based on the user's perspective and context. By incorporating the user's preferences and understanding, we aim to develop a code smell detection system that aligns with subjective needs and provides more tailored results.

In this paper, we present a comprehensive framework that addresses the issue of subjectivity in code smell detection and enables personalized and accurate predictions. Our approach combines deep learning techniques, user feedback, and a containerized deployment architecture for a locally-run web-server to create a robust and adaptable system. We train a baseline DL model using the DACOS dataset [10]. The dataset emphasizes collecting annotations on potentially subjective code snippets and hence helps the model learn the latent features necessary to classify snippets into smelly or benign. We integrate our DL model into a Docker container behind a web-server to offer smell predictions and retrain the model easily as and when required. Our initial model predicts code smells that we show users in the IntelliJ IDEA environment with the help of our plugin *TagCoder*. *TagCoder* shows the smells to the users and collects their feedback. We train the deployed DL model using the collected user feedback. The fine-tuning allows the model to learn and adapt to individual user preferences and enhances the accuracy of smell detection. Our experiments to evaluate the proposed approach show that fine-tuning the DL model using collected user feedback outperforms the base model for all the participants. This performance improvement is achieved for each participant while considering their feedback and maintaining the customization of the model's behavior specifically trained for each participant.

We make the following contributions to the state of the art.

- We propose a robust framework that mitigates subjectivity in code smell detection by incorporating user feedback for our deep learning model. The proposed framework provides a systematic approach to address the challenges associated with subjective code smells, leading to more accurate and personalized analysis.

- We implement a containerized solution integrating a lightweight web-server for providing code smell inferences from our trained DL model. The implementation enables efficient deployment, ease of fine-tuning, and scalability. We make the framework implementation open-source allowing researchers to use and extend our framework.
- We implement a plugin *TagCoder* for IntelliJ IDEA. The plugin provides an interactive user interface for personalized code smell analysis and captures valuable user feedback, leading to continuous model improvement. *TagCoder* bridges the gap between developers and the code smell detection system, empowering developers to actively participate in the improvement of code quality and maintainability. The plugin has been made publicly available.

Replication package: Our framework, plugin, analysis scripts, and data can be found online [11].

II. Background and Related Work

Code smells can be classified based on granularity, scope, and artifacts [3]. Traditionally, code smells are broadly divided into three main categories based on the granularity and scope—*implementation* [12], architecture [13] and design [14] smells. Implementation smells usually impact a limited scope such as a method. *Complex method* and *Long method* are examples of implementation smells [12]. Design smells, such as *god class* and *multifaceted abstraction* [14], affect a broader scope, *i.e.,* at a class-level. The scope of architectural smells span multiple components. Some examples of architecture smells are *feature concentration* [15] and *scattered functionality* [16]. In this study, we keep our focus on implementation and design smells.

A. Traditional approaches for code smells detection

Metrics-based approaches involve quantifying specific code characteristics or metrics and using predefined thresholds or rules to identify potential code smells. These approaches rely on analyzing code quality metrics such as cyclomatic complexity, lack of cohesion in methods, and code duplication. Tools such as PMD [17] and SonarQube [18] utilize metrics-based approaches by examining code metrics and providing suggestions for refactoring. In such methods, the source code is processed to create a code model, metrics capturing code characteristics are calculated, and then these metrics are compared against predefined thresholds to detect code smells. For example, the *God class* smell can be detected using metrics such as Weighted Methods per Class (WMC), Access To Foreign Data (ATFD), and Tight Class Cohesion (TCC) [19] [20]. These metrics are compared against predefined thresholds and combined using logical operators.

In addition to metrics-based approaches, another traditional approach for code smell detection is rules-based detection. Rules-based smell detection methods define specific rules or heuristics to identify code smells. These methods take the source code model as input and, in some cases, additional software metrics. Code smells are detected when the defined rules or heuristics are satisfied. By applying these rules, potential issues can be identified and flagged as code smells. For instance, the *cyclic hierarchy* smell can be detected by implementing a rule that examines whether a class is referencing its subclasses [14]. When this condition is met, it indicates the presence of a *cyclic hierarchy* smell. Rules or heuristics are often combined with metrics to improve the effectiveness of smell detection.

B. Machine learning approaches for code smells detection

Machine Learning for code smell detection has gained a lot of momentum in the recent years. We elaborate on the approaches used to detect smells using traditional machine learning and deep learning techniques.

1) Smell detection using traditional machine learning: Khomh *et al.* [6] use Bayesian Networks to predict *blob*, *functional decomposition*, and *spaghetti code* in two open-source projects. Maiga *et al.* [21] proposed SVMDetect, an approach to detect anti-patterns, based on support vector machines. It predicts *functional decomposition, blob, swiss army knife* and *spaghetti code*. In a study conducted by Saeys *et al.* [22], hybrid feature selection techniques such as recursive feature selection with *random forest* and *support vector machine* were employed. The performance measures of single and ensemble feature selection were compared, and it was found that hybrid feature selection outperformed the other methods. Jiarpakdee *et al.* [23] examined 12 feature selection techniques on 14 open-source datasets and concluded that feature selection had an impact of up to 9% on prediction, and that wrapper methods were expensive to implement.

2) Smell detection using deep learning: Deep learning approaches, particularly those utilizing recurrent neural networks (RNNs) such as LSTMs [24], are effective in capturing long-term dependencies in sequential data. These methods have been applied to source code, either for improving semantic representations [25] or for solving downstream tasks.

Alternative approaches to mining source code have employed CNNs in order to learn features from various representations of code. Li *et al.* [26] have used single-dimension CNNs to learn semantic and structural features of programs by working at the AST level of granularity and combining the learned features with traditional hand-crafted features to predict software defects. Their method, however, incorporates hand-crafted features in the learning process and is not proven to yield transferable results. Similarly, a one-dimensional CNN based architecture has been used by Allamanis *et al.* [27] in order to detect patterns in source code and identify "interesting" locations where attention should be focused. Similarly, Ren *et al.* [28] use a CNN-based neural network to identify self-admitted technical debt. Sharma *et al.* [29] used CNN, RNNs, and Autoencoders (AEs) to detect code smells without explicitly specifying code features. They showed that DL

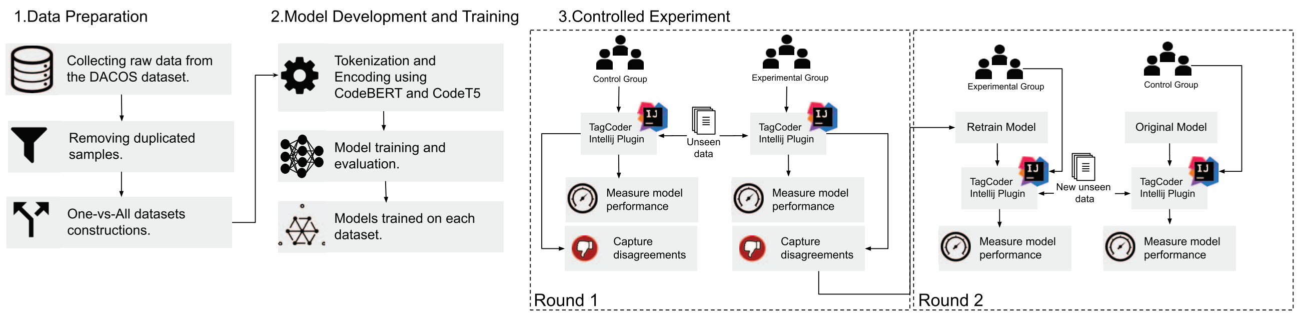
1.Data Preparation
Collecting raw data from the DACOS dataset.
Removing duplicated samples.
One-vs-All datasets constructions.
2.Model Development and Training
Tokenization and Encoding using CodeBERT and CodeT5
Model training and evaluation.
Models trained on each dataset.
3.Controlled Experiment
Control Group
Experimental Group
TagCoder Intellij Plugin
Unseen data
Measure model performance
Capture disagreements
Round 1
Retrain Model
Original Model
New unseen data
Round 2

Fig. 1: Overview of the approach

models are able to detect smells in direct and transfer learning context.

The main problem with ML-based methods for code smells detection is the high degree of disagreement on what constitutes a code smell among developers [30]. Hence, if a model performs well on a dataset annotated by a set of developers, it might perform poorly when evaluated by another set of developers. This deficiency makes these models unusable in real-life within an industrial software development environment.

C. Human feedback in machine learning studies

Feedback loops play a vital role in machine learning, enabling systems to continuously learn and adapt based on previous outputs [31]. In the context of software engineering, feedback loops have been widely explored for various purposes. Aguiar *et al.* [32] present a use case for feedback learning in live programming, demonstrating how real-time feedback can enhance the programming experience. Balzer's work [33] focuses on live coding and feedback learning, investigating how feedback loops can facilitate code development and improve programming efficiency. Brun *et al.* [34] explore the application of feedback loops in self-adaptive systems, where the system dynamically adjusts its behavior in response to changing environments and emerging requirements.

Despite the existing literature on feedback loops in software engineering, to the best of our knowledge, no previous work has specifically investigated the utilization of human feedback for DL-based code smell detection. In our research, we propose an approach that incorporates human feedback to enhance the performance of code smell detection models.

III. Overview

The *aim* of this research is to investigate the feasibility of customizing a DL model for code smell detection based on user feedback. To realize the aim of the study, we develop an initial DL model for smell prediction, gather user feedback on the model's predictions, and fine-tune the model based on the feedback received. By comparing the performance of the fine-tuned model with the original model, the study aims to determine if the DL model can be effectively customized to align with the specific preferences and requirements of individual users. This paper attempts to answer the following research questions:

RQ1 *Whether and to what extent does user feedback improve the accuracy of deep learning-based code smell detection?*

User feedback plays a crucial role in fine-tuning and refining DL models [31]. With this research question we aim to validate that user feedback can enhance the accuracy of DL-based code smell detection and to understand the extent to which this improvement can be observed.

RQ2 *Whether and to what extent does user feedback influence the accuracy of deep learning-based detection for individual code smells?*

Code smells exhibit distinct attributes that differentiate them from one another due to the variations in their characteristics, patterns, and severity. With this research question, we aim to examine whether the improvement in accuracy, if any, through user feedback is consistent across all considered smells or if it varies for each individual smell. Furthermore, we seek to quantify the extent of this variation to understand the degree of improvement achieved.

Figure 1 illustrates an overview of our approach. The approach has two major steps—training the DL model and conducting the controlled experiment. In the training phase of our study, we begin with the data preparation step, where we collect raw data samples from the DACOS dataset. To ensure data integrity and uniqueness, we carefully remove any duplicate samples from the collected data. With the clean dataset in hand, we proceed to train the DL models. To enhance the model's ability to leverage contextual information from the code samples, we employ two popular encoders: CodeBERT and CodeT5. The code samples are then passed through the tokenizer, which converts them into tokenized vectors. These tokenized vectors serve as inputs to the DL models for training. During the training process, we explore multiple DL models, comparing their performance on the given code smells.

For the controlled experiment, we conduct our study with the participation of users who are divided into two groups: the control group and the experiment group. In the initial phase, participants from both groups are presented with code samples, where we utilize the *TagCoder* plugin to capture their feedback. In the second phase, we introduce a feedback loop for the experiment group. We fine-tune the model using the user feedback captured from this group, tailoring the model's predictions according to their input. In contrast, the control group continues to interact with the same base model as in the first phase, without any fine-tuning based on user feedback.We then show predictions to both groups, and collect and compare their feedback.

IV. Deep Learning Models Training

We elaborate below the methods adopted to train and evaluate the base DL models. Later in this section, we present the obtained results for this training activity.

A. Methods

1) Code smells dataset: For the initial model training and evaluation, we leverage the dataset curated by Nandani *et al.* [10] for code smell detection. It is a manually labelled dataset consisting of $10,267$ annotations for $5,192$ code snippets. The dataset provide annotations for two implementation smells *viz. complex method*, *long parameter list* and one design smell *multifaceted abstraction*. The key advantage of using the DACOS dataset is that it collected annotations for non-trivial potentially subjective code snippets helping the machine learning-based classifiers to learn to segregate smelly and benign snippets with much more ease.

Allamanis [35] shows that duplicate samples can lead to inflated and misleading results during testing. To avoid the issues potentially caused by duplicate samples, we perform data de-duplication using a hash function to compute a unique hash value for each code instance and comparing the hash values to identify any duplicates. For the classification task, we create a dataset containing both smelly and benign samples for each code smell. We use a 70:30 split for training and testing. Table I illustrates statistics of the dataset. In the case of *multifaceted abstraction*, a sample refers to a Java class, whereas in case of *complex method* and *long parameter list*, a sample refers to a Java method.

TABLE I: Dataset statistics

Code smells	**#Training samples**	**#Test samples**	**Total samples**
Complex method	1,535	658	2,193
Long parameter list	1,137	487	1,624
Multifaceted abstraction	952	408	1,360

2) Generating initial code representations: To enable the DL model to leverage the contextual information present in the code samples, we incorporate an embedding step in our approach. By embedding the code samples, we facilitate the generation of meaningful context representations for further processing. In this study, we leverage the *transformers*[1] implementation of two Large Language Models pre-trained on code—*CodeBERT* [36] and *CodeT5* [37]. CodeBERT is a bimodal pre-trained language model tailored for Natural Language–Programming Language (NL–PL) tasks such as code search and documentation generation. It has been trained on an extensive dataset of six million GitHub projects, incorporating various programming languages, and employs a hybrid objective function to support its bimodal capabilities, utilizing both NL–PL data and unimodal data. On the other hand, CodeT5 is a unified pre-trained encoder-decoder Transformer model that capitalizes on the code semantics conveyed through developer-assigned identifiers [38]. CodeT5 is pre-trained with three distinct objective functions, namely masked span prediction, identifier tagging, and masked identifier prediction. These objectives serve as feedback signals to fine-tune the model parameters and enhance the code understanding capabilities. Both CodeBERT and CodeT5 have been extensively used in software engineering literature [39]–[41]. We use the last hidden state of each encoder. We then take the mean across all tokens, to generate a vector embedding for each code snippet of 768 dimensions. These embeddings serves as input for the classifiers described in the next sections.

[1]https://github.com/huggingface/transformers

3) Architecture of deep learning models: In this section, we describe the architecture of the DL models we used. The source code of the implementation is available online [42]. We implement three DL models: an Autoencoder with a dense Multi-Layer Perceptron (AE-MLP) classifier, an Autoencoder with a Long Short-Term Memory (AE-LSTM), and a *variational* Autoencoder (VEN) with a threshold-based strategy for classification.

4) Autoencoder with a dense classifier and LSTM: Figure 2 shows the architecture of the first two models. Each model is trained in two steps.

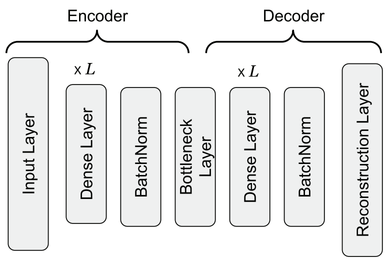

Fig. 2: Architecture of the employed Autoencoder model. It is composed of an encoder that compresses input into a latent representation that is then reconstructed using the decoder.

In the first step, we train an Autoencoder [43], a class of feed-forward neural networks designed to reconstruct the input data. Autoencoders possess the capability to compress the input data into a lower-dimensional representation, known as the *latent representation*, and subsequently reconstruct the output from this compressed representation. This process involves an encoder and a decoder as the key components. The encoder starts with an input layer followed by a series of dense layers. To improve training stability and efficiency, we add *Batch Normalization*, a normalization layer that standardizes inputs for each mini-batch. Subsequently, we replicate these layers in reverse order to construct the decoder, which is responsible for reconstructing the input data from the compressed representation.

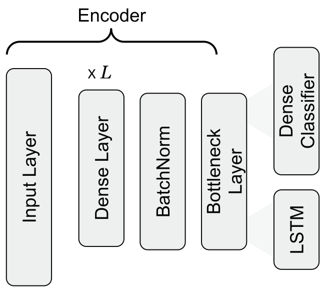

Fig. 3: Using Encoder of the Autoencoder for training the dense classifier and LSTM model

Table II provides an overview of the hyperparameters used for the DL models, including the Autoencoder. The hyperparameters include the number of encoder and decoder layers (Autoencoder), the loss functions used (MSE for the Autoencoder, CrossEntropy for MLP and LSTM), batch sizes (for all models), and the number of epochs.

TABLE II: Hyperparameters for the DL models

Hyperparameter	**Values**
Encoder Layers (Autoencoder)	1,2
Decoder Layers (Autoencoder)	1,2
Loss functions	MSE (AE), CrossEntropy (MLP, LSTM)
Batch Size (all)	32,64
Epochs (all)	5,10,15,20

During training, the autoencoder minimizes the reconstruction error between the input and the output. We use the Mean Squared Error (MSE) loss function. The MSE loss calculates the average squared difference between the input and reconstructed output as shown in the following equation.

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(x_i - \hat{x}_i)^2$$

By optimizing the network's parameters to minimize this loss, the autoencoder learns to encode and decode the input data effectively, capturing the most salient features in the process.

As Figure 3 shows, we extract the encoder and discard the decoder once the Autoencoder model is trained. The encoder is further utilized to compress new instances of input data into vectors output by the bottleneck layer, that is fed to either the dense classifier or the LSTM classification head.

5) Variational Autoencoders: A variational autoencoder (VAE) [44] is an extension of the conventional autoencoder that integrates probabilistic modeling. It serves as a deep generative model that employs Bayesian inference to estimate the latent representation. Similar to the autoencoder, the VAE consists of an encoder and a decoder. The encoder transforms the input x into a latent representation z, while the decoder reconstructs the original input data ($\hat{x}$) based on this latent vector. The model's joint distribution is defined as follows:

$$p_\theta(x,z) = p_\theta(x|z) \cdot p_\theta(z)$$

The encoder, denoted as $q_\phi(z|x)$, where ϕ represents its parameters, produces estimates of the mean and variance variables of a Gaussian distribution. Using these estimated parameters, the VAE generates a latent vector z by sampling from the distribution. The decoder, denoted as $p_\theta(x|z)$, then reconstructs the original input by mapping the latent vector z to the output space. The decoder's parameters are represented as θ. The VAE aims to find the maximum likelihood by optimizing the following expression:

$$\sum_{i=1}^{n} \log p_\Theta(x_i)$$

Where Θ represents the parameter of the encoder and decoder, and $\log p_\Theta(x_i)$ can be expressed as:

$$\log p_\Theta(x_i) = D_{KL}\left(q_\Phi(z|x_i) \| p_\Theta(z)\right) + L(\Theta;\Phi;x_i)$$

Where D_{KL} is the Kullback-Leibler divergence between the posterior and prior distributions $L(\Theta;\Phi;x_i)$ and is called the evidence variational lower bound (ELBO). We train a VAE for each smell, similar to the two previous classifiers. Specifically, we train the VAE on the positive training samples. To perform the classification, we set a threshold α: if the loss measured is greater than the threshold, then it is classified as negative. The reason we do this is since the VAE has been trained on one class, it would have learned the salient features of that particular class, minimizing the reconstruction error after epochs of training. Hence, a high loss entails that the VAE was exposed to an outlier, *i.e.,* a sample from a different class. The value of α is chosen after experimenting with multiple loss intervals with various steps, we report the value that yielded the highest predictive performance.

B. Results

Table III provides an overview of the classification results. using the encoders *i.e.,* CodeBERT and CodeT5, used for initial representation generation and used classifiers. With each encoder, we experiment with three combinations of Autoencoder (AE) and classifiers. The performance metrics of precision, recall, and F1-score are reported for each combination.

1) Using CodeBERT as an Encoder: For the *multifaceted abstraction* smell, the AE-MLP classifier achieved F1-score of 0.66. The AE-LSTM classifier demonstrated slightly better results with F1-score of 0.71. However, the VAE classifier outperformed both with F1-score of 0.85. This indicates that the VAE, leveraging the latent space representation, captured the distinguishing features of the *multifaceted abstraction* smell more effectively.

For the *complex method* smell, the VAE classifier achieved the best performance among the three models with a precision of 0.80, recall of 0.99, and F1-score of 0.89. This suggests that the VAE, by leveraging the probabilistic modeling and the threshold-based classification, effectively distinguished the *complex method* smell.

Finally, for *long parameter list* smell, the AE-MLP classifier achieved F1-score of 0.69. The AE-LSTM classifier exhibited

TABLE III: Classification results for each type of smell using CodeBERT and CodeT5 with different classifiers.

Encoder	Smell	AE-MLP			AE-LSTM			VAE		
		Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1
CodeBERT	Multifaceted Abstraction	0.56	0.80	0.66	0.70	0.73	0.71	0.79	0.91	**0.85**
	Complex Method	0.60	0.75	0.67	0.68	0.97	0.79	0.80	0.99	**0.89**
	Long Parameter List	0.62	0.78	0.69	0.66	0.97	0.79	0.81	0.90	**0.85**
CodeT5	Multifaceted Abstraction	0.60	0.72	0.66	0.45	0.99	0.62	0.77	0.89	**0.83**
	Complex Method	0.69	0.64	0.64	0.84	0.59	0.64	0.76	0.80	**0.78**
	Long Parameter List	0.60	0.72	0.65	0.80	0.57	0.67	0.83	0.79	**0.81**

F1-score of 0.79. However, the VAE classifier achieved the best results with F1-score of 0.85. The VAE's ability to capture the underlying probabilistic distribution of the *long parameter list* smell seemed to contribute to its superior performance.

Summary: The VAE consistently outperformed the AE-MLP and AE-LSTM classifiers across all three code smells. This can be attributed to the VAE's ability to model the latent space and capture the underlying probabilistic distribution. Leveraging the threshold-based classification, the VAE effectively distinguished positive and negative cases, resulting in higher precision and recall. In contrast, the AE-MLP and AE-LSTM classifiers demonstrated lower performance, potentially due to their limited capacity to capture complex patterns and dependencies in the data.

2) Using CodeT5 as an Encoder: For the code smell *multifaceted abstraction*, the CodeT5 model achieved an F1-score of 0.66 when using the AE-MLP model. When using the AE-LSTM model, F1-score was 0.62. For the VAE model, F1-score was 0.83.

For the code smell *complex method*, the CodeT5 model achieved an F1-score of 0.64 when using the AE-MLP model. When using the AE-LSTM model, the F1-score was 0.64. For the VAE model, the F1-score was 0.78.

For the code smell *long parameter list*, the CodeT5 model achieved an F1-score of 0.65 when using the AE-MLP model. When using the AE-LSTM model, F1-score was 0.67. F1-score was 0.81. Overall, CodeT5 exhibits varying performance across different code smells. While it demonstrates relatively strong precision for some smells, its recall and F1-scores vary.

Summary: For CodeT5 as well, VAE consistently outperforms AE-MLP and AE-LSTM across all the code smells. However, the performance of CodeT5 as an encoder is inferior compared to CodeBERT. CodeT5 shows lower recall scores for certain code smell types, indicating a higher rate of false negatives. This implies that CodeT5 has a more conservative approach to detecting code smells and may miss instances of code smells. Considering that the combination of AE-MLP and CodeBERT performs the best, we select this combination for our experiment.

V. The Controlled Experiment

A. Methods

In this section, we describe the tools and the protocol that we developed to capture human feedback, and the results we obtained.

*1) **Tools to capture human feedback**:* To conduct our experiment, we have developed two key software components—a web-server serving the model to perform inference and a plugin for popular IntelliJ IDEA Integrated Development Environment (IDE) for displaying classification results and capturing developers' feedback. In this section, we describe each software component in detail.

A web-server: The primary objective of the web-server to support the inference from our DL model that is decoupled from the user feedback collection system. We minimize the dependencies required to run the server by encapsulating the server as a Docker container.

For the implementation of the server, we utilize the Django framework, which is based on the Python programming language. In the beginning of the experiment, the server accepts method and class metrics generated by running Designite-Java [45] on the project. These metrics are stored in memory for subsequent processing. The server receives requests along with required data (such as metrics) from the client (in our case, our plugin for IntelliJ IDEA). The server offers the following four endpoints:

- *Metrics endpoint*: This endpoint accepts a POST request with two CSV files—one containing class metrics and the other containing method metrics. These files are generated by DesigniteJava and are recorded using our IntelliJ IDEA plugin.
- *Prediction endpoint*: A POST endpoint that accepts a source code file, along with a boolean value indicating whether it pertains to a class or a method. The file is passed to the DL model for inference.
- *Feedback endpoint*: Another POST endpoint that allows users to provide feedback on the model's predictions for a specific file. When a user is presented with a prediction, their feedback regarding the correctness (according to them) of the prediction is collected through this endpoint. The endpoint records the file and the user's feedback. Once a preset number of feedback instances have been collected, the model is fine-tuned.

- *Fine-tuning endpoint*: A GET request that enables users to explicitly trigger model fine-tuning whenever desired.

To expedite request processing and reduce server load, we leverage the metrics collected by DesigniteJava along with the rationale related to subjectivity used by Nandani *et al.* [10]; they identify whether a method or a class is definitely smelly based on metrics values higher than a threshold. Similarly, when the web-server receive a code snippet for prediction, we first look up the corresponding metrics. Based on preset thresholds as used in by Nandani *et al.* [10], we can quickly identify whether the sample is definitively smelly or definitely benign. For instance, if a method has eight parameters and the threshold for the "Parameter Count" metric is set between two and four, lower and higher threshold respectively, we can conclude that the method exhibits the *long parameter list* code smell. If the sample falls within the predefined threshold, we use the model to infer whether a smell is present or not and return the inference back to the plugin.

To simplify the deployment and setup process, we use a Docker script. This script automatically downloads all the necessary dependencies, including Python and the required DL libraries, ensuring a streamlined deployment of the server. We store the user feedback for the presented code samples on a configurable volume within the Docker container. This ensures that user feedback is not lost even if the container is shut down or restarted. Additionally, we have implemented a mechanism to fine-tune the model based on the collected feedback. Once the number of feedback instances reaches to a pre-defined, but configurable, threshold (currently set to 50), the web-server invokes fine-tuning the model incorporating the new information.

We setup and configure the web-server locally to avoid sending code snippets to a third-party server configured outside of an organization boundary. However, due to the flexibility of the containerized web-server, one can choose to install on their local machine, a server within their organization, or on public cloud-infrastructure.

TagCoder—An IntelliJ IDEA *plugin:* To enhance the usability and convenience of our code smell detection system, we developed *TagCoder*—a plugin for IntelliJ IDEA, a widely-used Integrated Development Environment (IDE) for software development, particularly in Java.

The objective of *TagCoder* is to provide users with a seamless experience in obtaining predicted code smells for their code and offering a straightforward mechanism for providing feedback on these predictions. When a user opens a project in the IDE, *TagCoder* automatically runs DesigniteJava in the background to analyze the project. The obtained results are then sent to our local web-server, which serves as the core of our code smell detection system.

After the initial analysis, whenever the user opens a file within the IDE, *TagCoder* automatically sends the corresponding class and method information to the web-server for code smell prediction. The server processes the received code snippets and returns the predictions back to the plugin.

The plugin then displays the predictions in the gutter on the left side pane of the IntelliJ IDEA editor, allowing users to conveniently view the code smells associated with specific classes and methods. This integration within the IDE's interface enables users to easily identify potential code smells without disrupting their workflow.

In addition, *TagCoder* supports recording users' feedback. Users can provide feedback on the identified smells directly from the plugin. This feedback is captured by the plugin and sent to the web-server for documentation, analysis, and model refinement. This iterative feedback loop helps improve the accuracy and reliability of the code smell predictions over time.

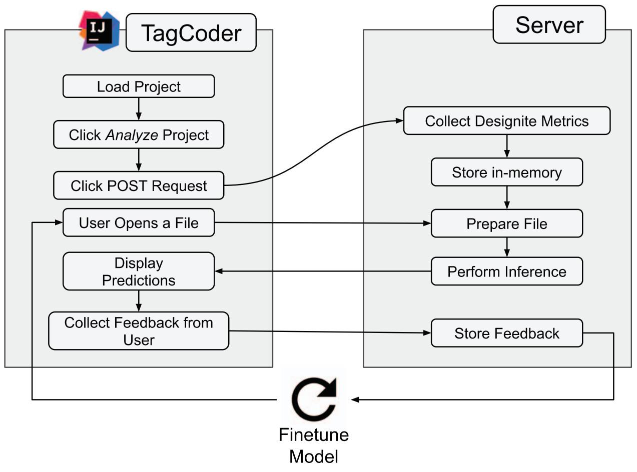

Fig. 4: Interaction between TagCoder and local web-server

Additionally, *TagCoder* offers an option to explicitly trigger model fine-tuning when users feel that a sufficient amount of feedback has been accumulated. This ensures that the model remains up-to-date and capable of capturing evolving code smells as the project progresses. By integrating *TagCoder* into IntelliJ IDEA, we aim to streamline the code smell detection process, providing developers with real-time insights into potential code quality issues and facilitating their active participation in improving the model's accuracy.

Experimental design and setup: The third step of Figure 1 illustrates the approach of conducing our controlled experiment. We elaborate the experimental design and setup in the rest of the section.

Participants: We recruited 14 participants that have a background in Computer Science and are enrolled in a graduate, post-graduate or doctoral degree program. They were solicited using a relevant internal mailing list. Participation was voluntary, but a small monetary reward was offered at the end of the experiments to compensate their time. All participants were informed about the purpose of the study and were asked to provide consent to record and publish the anonymous data. They were also informed that all personal information (such as name and email) gathered will be confidential and only the researchers involved in the study will have access to the personal data collected. The experiment took place at (redacted for blind review) spanning two days in May 2023.

Procedure: Before conducting the actual experiment, we performed a pilot study involving a participant to get their feedback regarding the *TagCoder* plugin. We do so to minimize any errors that could occur during the experimental procedure and get an estimate of the needed time to perform the annotation of at least 50 code snippets in each round. We then make necessary changes in the plugin as well as in the process based on the feedback we received.

During the experimentation days, each participant was randomly assigned to a group (*experimental* or *control*) in a way that both groups have the same number of participants. We provide the same computer for each participant with IntelliJ IDE and *TagCoder* plugin installed. The source code project that was imported into the IDE can be found in the replication package.

We then present the same source code to all the participants and ask them to perform the following tasks:

- Open the IDE and navigate to the "Tools" menu. In the menu, select the option to analyze code using *TagCoder*.
- Open the source code files one by one. The methods and classes would have the *TagCoder* icon in the gutter of the editor on the left.
- Assess the smells detected initially by the model by clicking on the *TagCoder* icon. The plugin shows the kind of the smell along with its description. They then provide their feedback (*i.e.,* agree or disagree with the detected smell) to the model from the same dialog box.

For the experimental group, after annotating at least 50 samples in Round 1, the model is fine-tuned. Upon fine-tuning the model, the web-server notifies the plugin, and the plugin shows a popup notification to inform the user about completing the fine-tuning process. However, a participant can also manually trigger the model's fine-tuning using the option present in the menu bar. The participants in the control group were presented with a modified version of the plugin. The modified plugin looks and behaves the same as the one presented to those in the experimental group except for a minor tweak-the model is not fine-tuned for the control group. Both the group members were unaware of their group and the difference in the plugin. For each session with a participant, we use a new copy of the original trained model and ensure that the fine-tuned models are saved for individuals and are not reused.

Data collection procedure: On average, completing both rounds took every participant approximately 48 minutes. The average number of annotations collected per user was 101 over both rounds. We collected $1,421$ annotations from this experiment, where the number of annotations performed by the control group ranged from 81 to 135. In contrast, the number for the experiment group ranged from 73 to 135. For each participant, we store the model's prediction and their response and calculate a hash for the code snippet to identify it uniquely. The classification performance metrics were calculated based on our received data by treating the participants' responses as the ground truth.

B. Results

In this section, we illustrate the results of the controlled experiment.

RQ1: Whether and to what extent does user feedback improve the accuracy of deep learning-based code smell detection? The violin plot in Figure 5 represents the distribution of F1-scores recorded for both groups after each round. The x-axis represents the rounds and the y-axis shows the F1-scores. The F1-scores varied between 0.57 and 0.90 (0.78 ± 0.1) for the experimental group in Round 1. In Round 1 of the experiment group, the violin plot exhibits a more widely distributed line in contrast to the control group. This variation can be attributed to human annotation behavior, where the F1-score's lower end, specifically a score of 0.57 for one of the participants, contributes to this dispersion. After the introduction of the feedback loop, the scores increased and ranged from 0.81 to 0.97 (0.88 ± 0.1) in Round 2. The results indicate that the feedback helped the model learn to classify smells better in the Round 2 for the experimental group. However, for the control group, the difference in the F1-scores in the both rounds is significantly lower than the control group.

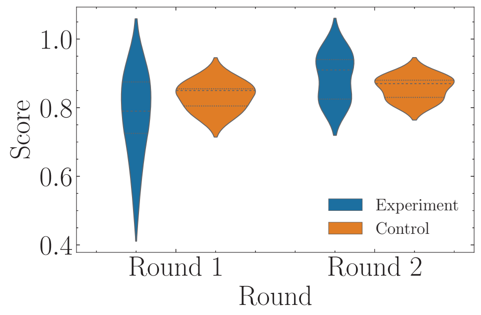

Fig. 5: Models' performance for each round.

We conduct statistical tests to determine the significance of the observed changes in the F1-score for both groups. Given the size of each group, we use a *Permutation test* for the parametric tests. We employ $n = 5,040$ permutations, representing all possible permutations, to ensure a robust analysis. The significance level is set at $\alpha = 0.05$ and we use *SciPy*'s [46] implementation for each test.

First, we examine the changes within the experimental group. The F1-scores obtained in Round 1 ($p = 0.367$) and Round 2 ($p = 0.197$) are found to follow a normal distribution based on the Shapiro-Wilk test [47]. Additionally, they satisfy the assumption of homoscedasticity, as determined by Levene's non-parametric test [48] ($p = 0.718$). Consequently, we perform a paired t-test to assess the significance of the F1-score changes within the experimental group. The null hypothesis (H_0) states that there is no significant increase in F1-scores between the two rounds for the experimental group.

The test yields a p-value of $0.015 < \alpha = 0.05$, with a t-value of -3.79. Therefore, we reject the null hypothesis, indicating a significant difference in F1-scores between the rounds. In addition, Hedge's $g = 1.13$ suggests a substantial difference between the experimental group's performance before and after introducing human feedback. The effect size indicates that the introduction of human feedback had a significant impact on the performance of the experimental group.

Furthermore, we explore the potential relationship between the number of annotations and the difference in F1-scores. We calculate Spearman's coefficient [49], resulting in $\rho = -0.03$ and $p = 0.963$. These findings indicate a negligible or near-nonexistent relationship between the two variables.

Similarly, the distributions of F1 measures for the control group after each phase were found to be normally distributed ($p = 0.805$ after Round 1 and $p = 0.466$ after Round 2) and exhibited homoscedasticity ($p = 0.6875$). However, the paired t-test yielded a p-value of $0.0625 > \alpha$ with a t-value of -2.497. With the t-test results, we cannot reject the null hypothesis of a significant F1-score change in the control group, *i.e.,* the change is not significant.

Summary: Our findings provide evidence that incorporating human feedback enhances the performance of DL models for code smell detection, as shown by the significant improvement of 15.49% on an average in F1-scores. The absence of a significant change in the control group further supports the conclusion that the observed improvements in the experimental group can be attributed to the incorporated feedback.

RQ-2: Whether and to what extent does user feedback influence the accuracy of DL-based detection for individual code smells?

Table IV summarizes the impact of the user feedback on the performance metrics for the *complex method*, *long parameter list* and *multifaceted abstraction* smells. The findings demonstrate the effectiveness of the feedback in enhancing the smell detection performance of the trained model.

TABLE IV: Influence of the feedback on classifiers' performance for the considered code smells individually

Smell	Round 1			Round 2		
	Precision	Recall	F1	Precision	Recall	F1
Multifaceted Abstraction	0.91	0.88	0.88	0.93 ↑	0.90 ↑	0.91 ↑
Complex Method	0.86	0.77	0.81	0.86 –	0.81 ↑	0.83 ↑
Long Parameter List	0.69	0.88	0.77	0.74 ↑	0.95 ↑	0.83 ↑

Before incorporating the feedback loop, the model achieved a precision of 0.91, a recall of 0.88, and an F1-score of 0.88 for the *multifaceted abstraction* code smell. Following the feedback loop, there was a slight improvement across all metrics, with the precision increasing to 0.93, recall to 0.90, and the F1-score to 0.91.

For the *complex method* code smell, the initial performance showed a precision of 0.86, a recall of 0.77, and an F1-score of 0.81. In the second round, there was a marginal enhancement in the recall and F1-score, while the precision remained unchanged at 0.86.

Similarly, the model's performance for the *long parameter list* smell demonstrated a precision of 0.69, a recall of 0.88, and an F1-score of 0.77 for the base model. Subsequently, introducing human feedback yielded significant improvements, with the precision increasing to 0.74, recall to 0.95, and the F1-score to 0.83. The results indicate that incorporating human feedback positively influenced the models' code smell detection capabilities. The inclusion of human feedback resulted in improved F1-scores, indicating enhanced precision and recall trade-off in the models' detection of code smells. Notably, the *long parameter list* code smell exhibited the most substantial improvement, followed by *multifaceted abstraction* and *complex method*. This suggests that the effectiveness of the feedback loop may vary depending on the specific code smell being detected.

The variation in performance after incorporating human feedback across different code smells can be attributed to several factors. The number of training samples for each smell influences the initial performance, with larger sample sizes potentially resulting in higher performance. The complexity and characteristics of each smell also play a role, with some smells being more straightforward to detect and classify accurately. For example, detecting *complex method* smell involves considerably difficult than *long parameter list* smell due to larger and more complex code snippet to process. Due to the complexity, it requires more number of training samples to learn to classify correctly.

Summary: Incorporating human feedback improves the performance of DL models across all smells. However, such improvement varies from one smell to another. The variation in performance can be attributed to different factors, such as the size and availability of data on which the initial model was trained and the nature of the code smell, where more nuanced smells can benefit more from feedback.

VI. Discussion

Deep learning models typically rely on a large dataset to learn and generalize patterns. However, during the fine-tuning process with extensive data, the subjectivity of individual users can be lost, resulting in a more generalized model that may not capture the unique perspectives and preferences of each user. One may wonder whether the individual models preserve the individual character due to learnt subjectivity and therefore, differ from other similar models.

The subjectivity of code smells is reflected in the threshold for a code snippet to become smelly; where it differs from one developer to another. Statistical coefficients like the Kappa-Cohen score [50], Gwett's AC1/AC2 [51], and Krippendorff's Alpha [52] are used to measure inter-annotator agreements. While all these metrics can be used to measure inter-annotator agreements, the Kappa-Cohen coefficient works best when used for comparison between two annotators [53]. Moreover, Gwett's AC1/AC2 and the Kappa-Cohen coefficient all possess a significant bias when there are a large number of non-random missing values [54]. For this reason, we selected the Krippendorff's Alpha coefficient to measure the inter-annotator agreement values. Krippendorff's Alpha works well with any number of annotators and can handle the missing data well [52]. Krippendorff's Alpha is computed as follows:

$$\text{Alpha} = 1 - \frac{D_o}{D_e}$$

where D_o is the observed disagreement and D_e is the expected disagreement.

We generate a matrix of all the samples annotated by two or more annotators before and after fine-tuning the model. We then construct a two-dimensional matrix and passed it to the Krippendorff Python library [55]. Table V presents the Krippendorff values before and after fine-tuning.

TABLE V: Krippendorff's Alpha coefficient values

Round	Smell	Alpha value
Round 1	Complex Method	0.44
	Multifaceted Abstraction	–
	Long Parameter List	0.35
	Overall	**0.46**
Round 2	Complex Method	0.51
	Multifaceted Abstraction	–
	Long Parameter List	0.36
	Overall	**0.48**

In Round 1, the *complex method* had an Alpha value of 0.44, while we did not have enough samples with common annotations for *multifaceted abstraction*. The *long parameter list* had an Alpha value of 0.35. The overall Alpha value for Round 1 was 0.46. In Round 2, the *complex method* had an Alpha value of 0.51, and the *long parameter list* had an Alpha value of 0.36. The overall Alpha value for Round 2 was 0.48. With these results, it is reasonable to state that the subjectivity is not *diluted* in the updated models. Despite the increase in the model's performance, the continued presence of relatively low Krippendorff's alpha values indicates that the subjective nature and variability among developers in their assessments of code smells persist. In addition, this complements our insight in Section V-B; the fact that certain smell models showed more substantial improvements (*long parameter list* vs *complex method*) with the consistent ranking of subjectivity, indicates that they have become more attuned to the subjective assessments of developers for those specific smells.

VII. Threats to Validity

To address potential *internal validity* threats, we employed random assignment of participants to the control and experimental groups. This helps mitigate selection bias by ensuring that any differences in the results between the groups are more likely due to the introduction of human feedback rather than pre-existing differences. Additionally, we controlled for the potential influence of maturation by limiting each experiment session to a maximum of 90 minutes. Moreover, to ensure the validity of the tools used to capture feedback, we conducted a pilot study to validate their effectiveness and reliability. This helped us to mitigate any potential biases or limitations associated with the data collection instruments.

Regarding *external validity*, we provided detailed information about the participant characteristics, such as being graduate students in Computer Science, and the source of recruitment through the university's mailing lists. This helps readers assess the generalizability of the findings within the target population. We also described the study setting, being conducted in a university environment, and provided contextual information to aid readers in evaluating the transferability of the findings to similar settings. In addition, the provision of a replication package, including the data and code used in the study, contributes to the external validity of the research.

To manage *conclusion validity* threats, we aimed for adequate sample size and performed statistical analysis using well-known statistical tests. By doing so, we aimed to minimize the risk of concluding a false effect. We controlled the significance level (alpha) to manage the risk of Type-I errors. Furthermore, by achieving sufficient statistical power, we aimed to mitigate Type-II errors. Finally, to address potential confounding variables, we employed randomization in the assignment of participants.

Conclusions

This study explored the effects of introducing human feedback on the performance of trained models in detecting code smells, considering the subjective nature of developers' perceptions. The findings revealed a significant improvement in the models' performance after incorporating human feedback. These results emphasize the importance of continuous feedback to align the model's predictions with developers' subjective judgments. The study also observed variations in the performance improvements across different code smells, suggesting that the impact of the feedback loop may be influenced by specific characteristics of code smells. In terms of implications, this research contributes to the understanding of how human feedback can enhance the accuracy of code analysis models. By leveraging the subjectivity of developers' perceptions, models can be refined to better align with their perspectives and improve overall software quality.

Future work in this direction includes applying the approach used in this study to other areas in software engineering with high subjectivity, such as code review and bug triaging. Additionally, the integration of active learning techniques could enhance the model's performance by strategically selecting the most informative samples for user feedback. We also aim to diversify the experiments by inviting software professionals in diverse domains as participants to not only generalize the findings but also to spot differences in perception arising from geography, organization, and other related factors.

References

[1] M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, 1980.

[2] M. Fowler, *Refactoring: Improving the Design of Existing Programs*, 1st ed. Addison-Wesley Professional, 1999.

[3] T. Sharma and D. Spinellis, "A survey on software smells," *Journal of Systems and Software*, vol. 138, pp. 158 – 173, 2018.

[4] I. M. Bertran, "Detecting architecturally-relevant code smells in evolving software systems," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11, 2011, p. 1090–1093.

[5] T. Mens and T. Tourwe, "A survey of software refactoring," *IEEE Trans. Softw. Eng.*, vol. 30, no. 2, pp. 126–139, Feb. 2004.

[6] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "Bdtex: A gqm-based bayesian approach for the detection of antipatterns," *Journal of Systems and Software*, vol. 84, no. 4, pp. 559–572, 2011, the Ninth International Conference on Quality Software.

[7] A. Barbez, F. Khomh, and Y.-G. Guéhéneuc, "A machine-learning based ensemble method for anti-patterns detection," *Journal of Systems and Software*, vol. 161, p. 110486, 2020.

[8] H. Liu, J. Jin, Z. Xu, Y. Zou, Y. Bu, and L. Zhang, "Deep learning based code smell detection," *IEEE transactions on Software Engineering*, vol. 47, no. 9, pp. 1811–1837, 2019.

[9] D. Oliveira, W. K. G. Assunção, A. Garcia, B. Fonseca, and M. Ribeiro, "Developers' perception matters: machine learning to detect developer-sensitive smells," *Empirical Software Engineering*, vol. 27, no. 7, Oct. 2022.

[10] H. Nandani, M. Saad, and T. Sharma, "Dacos—a manually annotated dataset of code smells," *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, 2023.

[11] ——, "Smart-dal/dlfeedback: v1.1.0," Aug. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.8259957

[12] M. Fowler, *Refactoring*. Addison-Wesley Professional, 2018.

[13] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Identifying architectural bad smells," in *2009 13th European Conference on Software Maintenance and Reengineering*. IEEE, 2009, pp. 255–258.

[14] G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for software design smells: managing technical debt*. Morgan Kaufmann, 2014.

[15] H. S. de Andrade, E. Almeida, and I. Crnkovic, "Architectural bad smells in software product lines: An exploratory study," in *Proceedings of the WICSA 2014 Companion Volume*, 2014, pp. 1–6.

[16] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Toward a catalogue of architectural bad smells," in *Architectures for Adaptive Software Systems: 5th International Conference on the Quality of Software Architectures, QoSA 2009, East Stroudsburg, PA, USA, June 24-26, 2009 Proceedings 5*. Springer, 2009, pp. 146–162.

[17] PMD, *PMD Source Code Analyzer*, PMD, 2021. [Online]. Available: https://pmd.github.io/

[18] SonarSource, *SonarQube*, SonarSource, 2021. [Online]. Available: https://www.sonarqube.org/

[19] R. Marinescu, "Measurement and quality in object-oriented design," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE, 2005, pp. 701–704.

[20] S. A. Vidal, C. Marcos, and J. A. Díaz-Pace, "An approach to prioritize code smells for refactoring," *Automated Software Engineering*, vol. 23, pp. 501–532, 2016.

[21] A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y.-G. Guéhéneuc, G. Antoniol, and E. Aïmeur, "Support vector machines for anti-pattern detection," in *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 2012, pp. 278–281.

[22] Y. Saeys, T. Abeel, and Y. Van de Peer, "Robust feature selection using ensemble feature selection techniques," in *Machine Learning and Knowledge Discovery in Databases*, W. Daelemans, B. Goethals, and K. Morik, Eds., Berlin, Heidelberg, 2008, pp. 313–325.

[23] J. Jiarpakdee, C. Tantithamthavorn, and C. Treude, "The impact of automated feature selection techniques on the interpretation of defect models," *Empirical Software Engineering*, vol. 25, no. 5, p. 3590–3638, 2020.

[24] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[25] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," 2018.

[26] H. Li, Z. Liu, H. Zhu, H. Wang, and Z. Yang, "Cp-miner: A tool for finding copy-paste and related bugs in operating system code," *IEEE Transactions on Software Engineering*, vol. 43, no. 4, pp. 335–355, 2017.

[27] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 378–389.

[28] R. Ren, C. Nistor, L. Schumacher, and B. Meyer, "Identifying self-admitted technical debt: A machine learning approach," *Empirical Software Engineering*, vol. 24, no. 5, pp. 3204–3242, 2019.

[29] T. Sharma, V. Efstathiou, P. Louridas, and D. Spinellis, "Code smell detection by deep direct-learning and transfer-learning," *Journal of Systems and Software*, vol. 176, p. 110936, 2021.

[30] T. Lewowski and L. Madeyski, "How far are we from reproducible research on code smell detection? a systematic literature review," *Information and Software Technology*, vol. 144, p. 106783, 2022.

[31] J. W. Vaughan, "Making better use of the crowd: How crowdsourcing can advance machine learning research," *Journal of Machine Learning Research*, vol. 18, no. 193, pp. 1–46, 2018. [Online]. Available: http://jmlr.org/papers/v18/17-234.html

[32] A. Aguiar, A. Restivo, F. F. Correia, H. S. Ferreira, and J. a. P. Dias, "Live software development: Tightening the feedback loops," in *Companion Proceedings of the 3rd International Conference on the Art, Science, and Engineering of Programming*, ser. Programming '19, 2019.

[33] R. Balzer, "A 15 year perspective on automatic programming," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 11, pp. 1257–1268, 1985.

[34] Y. Brun, G. Di Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw, *Engineering Self-Adaptive Systems through Feedback Loops*, Berlin, Heidelberg, 2009, pp. 48–70.

[35] 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*, ser. Onward! 2019, 2019, p. 143–153.

[36] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, Online, Nov. 2020, pp. 1536–1547.

[37] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, Online and Punta Cana, Dominican Republic, Nov. 2021, pp. 8696–8708.

[38] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *CoRR*, vol. abs/2109.00859, 2021. [Online]. Available: https://arxiv.org/abs/2109.00859

[39] S. Kwon, J.-I. Jang, S. Lee, D. Ryu, and J. Baik, "Codebert based software defect prediction for edge-cloud systems," in *Current Trends in Web Engineering*, G. Agapito, A. Bernasconi, C. Cappiello, H. A. Khattak, I. Ko, G. Loseto, M. Mrissa, L. Nanni, P. Pinoli, A. Ragone, M. Ruta, F. Scioscia, and A. Srivastava, Eds., Cham, 2023, pp. 11–21.

[40] C. S. Xia and L. Zhang, "Less training, more repairing please: Revisiting automated program repair via zero-shot learning," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022, 2022, p. 959–971.

[41] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Phung, "Vulrepair: A t5-based automated software vulnerability repair," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022, 2022, p. 935–947.

[42] H. Nandani, M. Saad, and T. Sharma, "himesh13/tagman_phase2: v1.1.0," Jan. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.7549420

[43] M. A. Kramer, "Nonlinear principal component analysis using autoassociative neural networks," *Aiche Journal*, vol. 37, pp. 233–243, 1991.

[44] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *arXiv preprint arXiv:1312.6114*, 2013.

[45] T. Sharma, "Designite - A Software Design Quality Assessment Tool," May 2016. [Online]. Available: https://doi.org/10.5281/zenodo.2566832

[46] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W.

Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[47] S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)," *Biometrika*, vol. 52, no. 3/4, pp. 591–611, 1965.

[48] B. Mandelbrot, "Contributions to Probability and Statistics: Essays in Honor of Harold Hotelling (Ingram Olkin, Sudhist G. Ghurye, Wassily Hoeffding, William G. Madow, and Henry B. Mann, eds.)," *SIAM Review*, vol. 3, no. 1, pp. 80–80, 1961.

[49] C. Spearman, "The proof and measurement of association between two things," *The American Journal of Psychology*, vol. 100, no. 3/4, pp. 441–471, 1987.

[50] H. C. Kraemer, "Kappa coefficient," *Wiley StatsRef: statistics reference online*, pp. 1–4, 2014.

[51] K. L. Gwet, "Computing inter-rater reliability and its variance in the presence of high agreement," *British Journal of Mathematical and Statistical Psychology*, vol. 61, no. 1, pp. 29–48, 2008.

[52] K. Krippendorff, "Computing krippendorff's alpha-reliability," 2011.

[53] A. Zapf, S. Castell, L. Morawietz, and A. Karch, "Measuring inter-rater reliability for nominal data – which coefficients and confidence intervals are appropriate?" *BMC Medical Research Methodology*, vol. 16, no. 1, Aug. 2016.

[54] W. Thompson and S. D. Walter, "A reappraisal of the kappa coefficient," *Journal of Clinical Epidemiology*, vol. 41, no. 10, pp. 949–958, 1988.

[55] S. Castro, "Fast krippendorff," 2023. [Online]. Available: https://pypi.org/project/krippendorff/