Are Sparse Autoencoders Useful for Java Function Bug Detection?

Rui Melo
*FEUP & INESC-ID**
Porto, Portugal
up202408602@up.pt

Claudia Mamede†
FEUP & CMU
Carnegie Mellon University
Pittsburgh, USA
cmamede@andrew.cmu.edu

Andre Catarino†
FEUP
Porto, Portugal
up202408593@up.pt

Rui Abreu
FEUP & INESC-ID
Porto, Portugal
rma@fe.up.pt

Henrique Lopes Cardoso
FEUP & LIACC
Porto, Portugal
hlc@fe.up.pt

***Abstract*—Software vulnerabilities such as buffer overflows and SQL injections are a major source of security breaches. Traditional methods for vulnerability detection remain essential but are limited by high false positive rates, scalability issues, and reliance on manual effort. These constraints have driven interest in AI-based approaches to automated vulnerability detection and secure code generation. While Large Language Models (LLMs) have opened new avenues for classification tasks, their complexity and opacity pose challenges for interpretability and deployment. Sparse Autoencoder offer a promising solution to this problem. We explore whether SAEs can serve as a lightweight, interpretable alternative for bug detection in Java functions. We evaluate the effectiveness of SAEs when applied to representations from GPT-2 Small and Gemma 2B, examining their capacity to highlight buggy behaviour without fine-tuning the underlying LLMs. We found that SAE-derived features enable bug detection with an F1 score of up to 89%, consistently outperforming fine-tuned transformer encoder baselines. Our work provides the first empirical evidence that SAEs can be used to detect software bugs directly from the internal representations of pretrained LLMs, without any fine-tuning or task-specific supervision.**

***Index Terms*—Sparse Autoencoders, Large Language Models, Bugs, Patches, Interpretability**

I. Introduction

Software vulnerabilities such as buffer overflows and SQL injections are a major source of security breaches [1], [2]. Such vulnerabilities frequently stem from developers lacking security expertise [3]. Therefore, ensuring the security of software systems is both a practical necessity and an ongoing challenge in computer science.

Traditional methods for vulnerability detection, e.g., static analysis, code reviews, and formal verification, remain essential but are limited by high false positive rates [4], scalability issues, and reliance on manual effort. These constraints have driven interest in AI-based approaches to automated vulnerability detection and secure code generation.

While Large Language Models (LLMs) have opened new avenues for classification tasks, their complexity and opacity pose challenges for interpretability and deployment [5], raising serious concerns about the safe deployment of LLMs in real-world programming environments [6].

*Work done as an External Collaborator
†Equal Contribution

Sparse Autoencoders (SAEs) offer a promising solution to this problem. In particular, recent advances using SAEs have shown that neuron activations in transformers can be represented as sparse, linear combinations of meaningful features [7]–[11]. By representing activations in this sparse form, we move beyond aggregate or surface-level metrics and gain direct insight into which features contribute to buggy code identification. Thus, mechanistic interpretability provides a promising path forward by seeking to decompose neural networks into human-interpretable components.

In this paper, we bridge the gap between interpretability research and software security by investigating how LLMs internally represent buggy code patterns. We explore whether SAEs can serve as a lightweight, interpretable alternative for bug detection in Java functions, and ask: **Are Sparse Autoencoders useful for Java function bug detection?**

We evaluate the effectiveness of SAEs when applied to representations from GPT-2 Small and Gemma 2B, examining their capacity to highlight buggy behaviour without fine-tuning the underlying LLMs. Our goal is to assess whether mechanistic interpretability tools, specifically SAEs, can surface actionable features for software vulnerability detection in LLM-generated code. To this end, we conduct the first in-depth empirical study that leverages SAEs in the context of bug detection. We perform a comprehensive layer-wise analysis across both models. Sparse features extracted from neuron activations are then used to train lightweight classifiers, such as random forests.

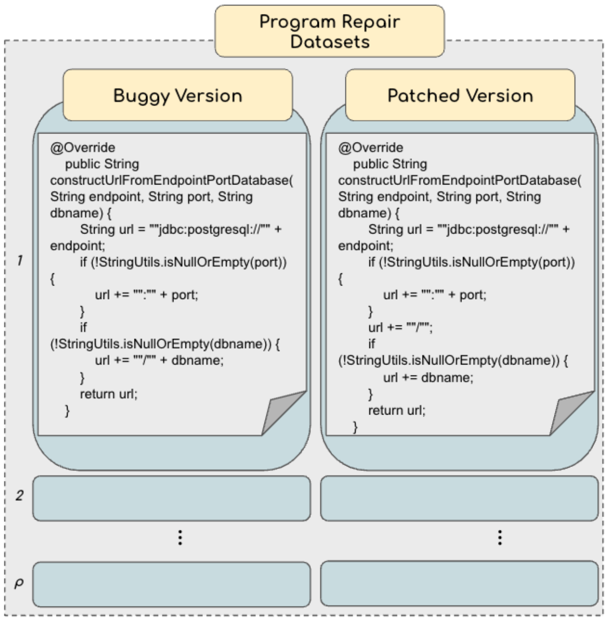
This paper makes the following contributions:

1) **Bridging interpretability and software security:** We conduct the first in-depth empirical study applying SAEs to internal activations of LLMs for the task of software bug detection, focusing on Java functions.
2) **Layer-wise analysis of model representations:** We systematically analyse sparse features extracted from two pretrained LLMs, GPT-2 Small and Gemma 2B, across all transformer layers.
3) **Lightweight classification using interpretable features:** We demonstrate that simple classifiers (e.g., random forests) trained on SAE-extracted features can detect buggy code with up to 89% F1 score, outperforming fine-

tuned transformer encoder baselines.

4) **A scalable, interpretable framework for bug detection:** Our approach highlights a practical path for integrating interpretability tools into security-critical workflows and shows how feature-based representations from SAEs can surface actionable signals for vulnerability detection, without any additional fine-tuning.

Our work provides the first empirical evidence that SAEs can be used to detect software bugs directly from the internal representations of pretrained LLMs, without any fine-tuning or task-specific supervision. We show that simple classifiers trained on these interpretable features can outperform fine-tuned baselines on Java vulnerability detection. These results highlight a practical and scalable approach for using interpretability tools in real-world security tasks and offer a new direction for research at the intersection of AI safety and software engineering.



```
@Override
   public String
constructUrlFromEndpointPortDatabase(
String endpoint, String port, String
dbname) {
       String url = ""jdbc:postgresql://"" +
endpoint;
       if (!StringUtils.isNullOrEmpty(port))
{
           url += "":"" + port;
       }
       if
(!StringUtils.isNullOrEmpty(dbname)) {
           url += ""/"" + dbname;
       }
       return url;
   }
```

```
@Override
   public String
constructUrlFromEndpointPortDatabase(
String endpoint, String port, String
dbname) {
       String url = ""jdbc:postgresql://"" +
endpoint;
       if (!StringUtils.isNullOrEmpty(port))
{
           url += "":"" + port;
       }
       url += ""/"";
       if
(!StringUtils.isNullOrEmpty(dbname)) {
           url += dbname;
       }
       return url;
   }
```

Fig. 1: Overview of data used.

II. Problem Description

Let each code sample $p_i \in \mathcal{P}$ be represented as a pair of code snippets:

$$p_i = (p_i^{\text{buggy}},\ p_i^{\text{patched}}),$$

where p_i^{buggy} denotes the buggy version and p_i^{patched} its corrected counterpart. The dataset $\mathcal{P}$, visualised in Figure 1, consists of such paired examples.

To analyse internal structure, we extract activation vectors $\mathbf{x} \in \mathbb{R}^d$ from a frozen LLM at a specific hidden layer. Each activation $\mathbf{x}$ is then encoded using a SAE, yielding a sparse representation $\mathbf{c} \in \mathbb{R}^k$, where $k \gg d$.

Our central hypothesis is that a subset of sparse SAE features captures semantically meaningful distinctions between buggy and patched code. Specifically, we posit that across many examples in $\mathcal{P}$, some SAE features consistently differ in their activation between buggy and corrected code.

Given a dataset $\mathcal{D}$ of individual code snippets to be classified, our objective is to learn a binary function $f : \mathbb{R}^k \to \{0, 1\}$, where $f(\mathbf{c}) = 1$ denotes buggy code and $f(\mathbf{c}) = 0$ denotes non-buggy code. Here, $\mathbf{c} = \text{SAE}(\mathbf{x})$, with $\mathbf{x} \in \mathbb{R}^d$ being the activation vector derived from a code snippet.

We restrict f to operate on a small subset of the most informative SAE features, selected using a function TopK. Formally, we aim to learn:

$$\text{Bug}(d) \approx f\left(\text{TopK}(\text{SAE}(\mathbf{x}_d))\right),$$

where $\mathbf{x}_d$ denotes the LLM activation vector extracted from code snippet d, and f is a lightweight classifier such as logistic regression or a random forest.

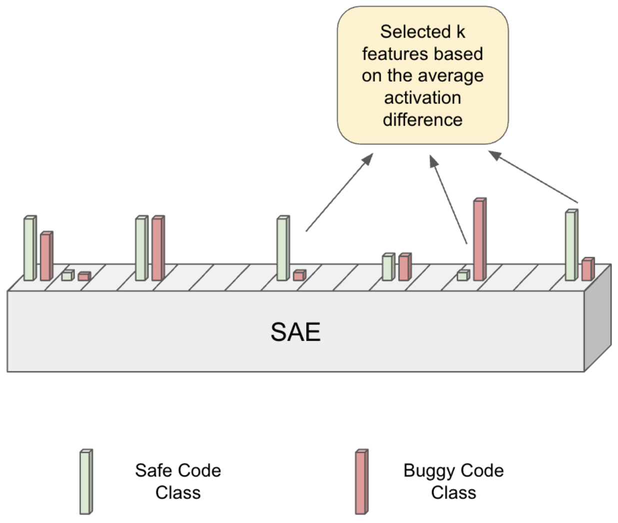

Selected k features based on the average activation difference
SAE
Safe Code Class
Buggy Code Class

Fig. 2: An illustration of the SAE feature selection based on the average difference between feature activation for buggy and safe code.

A. Feature Activation Case Studies

We further illustrate our methodology via token-level activation patterns.

Listing 1 presents a buggy and patched implementation of PostgreSQL URL construction. The buggy code improperly appends a trailing slash even when the database name is absent, risking malformed URLs.

The token-wise activation of Feature `6111` (Figure 3) shows elevated activations around bug-prone operations, such as URL concatenations. This indicates GPT-2 Small's capacity to localise critical code regions associated with security risks.

A second example, shown in Listing 2, involves string literal construction. The buggy code conditionally selects quotation marks, potentially introducing inconsistencies.

Similarly, for Gemma 2B we can see the activation patterns of Feature `1987` (Figure 4) highlight critical tokens involved in conditional logic, again showcasing the model's capacity to surface bug-relevant semantics.

Listing 1: Buggy and patched implementations of URL construction.

```
// Buggy Code:
@Override
public String constructUrlFromEndpointPortDatabase(
    String endpoint,
String port, String dbname) {
    String url = "jdbc:postgresql://" + endpoint;
    if (!StringUtils.isNullOrEmpty(port)) {
        url += ":" + port;
    }
    url += "/";
    if (!StringUtils.isNullOrEmpty(dbname)) {
        url += dbname;
    }
    return url;
}

// Patched Code:
@Override
public String constructUrlFromEndpointPortDatabase(
    String endpoint,
String port, String dbname) {
    String url = "jdbc:postgresql://" + endpoint;
    if (!StringUtils.isNullOrEmpty(port)) {
        url += ":" + port;
    }
    if (!StringUtils.isNullOrEmpty(dbname)) {
        url += "/" + dbname;
    }
    return url;
}
```

Together, these examples demonstrate that sparse autoencoder features capture meaningful semantic distinctions between buggy and secure code, offering a promising pathway toward interpretable bug detection.

Ultimately, our research question is whether this approach, relying only on latent representations from a static LLM and sparse dictionary features, can effectively distinguish between buggy and non-buggy Java functions without model fine-tuning.

III. Methodology

Our approach centers on retrieving the activated features from various SAEs, each corresponding to individual layers of the LLM. We then select the most meaningful of these features and use them as input for simple classifiers.

Notably, our method does not require fine-tuning the LLM. Instead, we aim to leverage the representations produced by a generic encoder and repurpose them for our specific task.

To evaluate our approach, we compare it against State of the Art (SOTA) classifiers and more naive baseline methods. An overview of the full methodology is illustrated in Figure 5.

A. Data

Following recent related work [12]–[15], we select three Java Datasets for our evaluation, scoping it to single-function bugs: `Defects4J` [16], `HumanEval` [17], and `Github-Bug Java` [18]. `Defects4J` comprises 835 real-world bugs sourced from 17 open-source Java projects, from which we filtered 488 single-function bugs. `Github-Bug Java`, on the other hand, is a benchmark of recent bugs collected from the 2023 commit history of 55 open-source repositories, encompassing a total of 90 single-function bugs. `HumanEval` contains 162 single-function bugs artificially inserted in HumanEval [6].

All datasets contain buggy snippets and patched versions, allowing us to have a balanced dataset immediately. We focus on binary classification, with the target being either 0 or 1. To train and evaluate our models, we randomly split each dataset into 80% training and 20% testing sets. The selected datasets will be used to systematically investigate model behaviour in detecting software bugs. These datasets provide a diverse set of

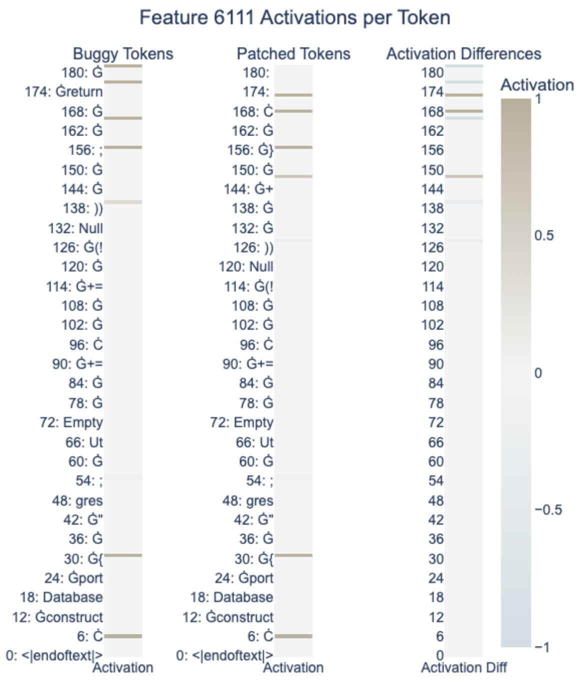


Fig. 3: Token-wise activation of Feature `6111` for the code in Figure 1. Elevated activations correlate with bug-relevant tokens.

Listing 2: Buggy and patched implementations of string literal construction.

```
// Buggy Code:
@Override
public String asStringLiteral()
{
    return this.value.indexOf('\"') == -1 ? '"' +
        this.value + '"' : '\'' + this.value + '\'';
}

// Patched Code:
@Override
public String asStringLiteral()
{
    return '"' + this.value + '"';
}
```

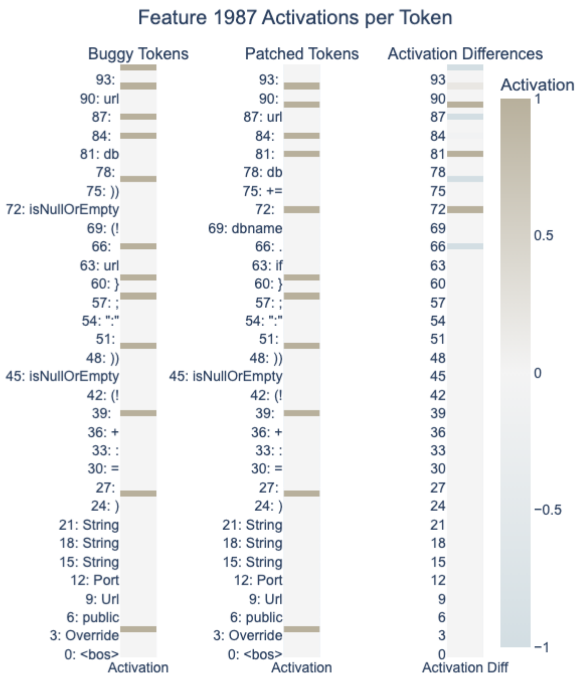


Fig. 4: Token-wise activation of Gemma 2B's Layer 0 Feature `1987` for the code in Code 2.

real-world coding errors, enabling a robust model performance evaluation. The simple data structure can be visualised in Figure 1.

B. Pairwise Approach

We analyse how a model processes different types of inputs, focusing on the differences in its handling of *buggy code* versus *patched code*. We employ Random Forest to evaluate the classification performance on the SAEs' learned feature representations.

C. Classical Data Mining

We then apply TF-IDF vectorisation to the token sequences for our classical baselines, producing feature representations. To ensure uniform input sizes across models, we pad each vector to a fixed dimensionality of 5000 features. We perform an extensive grid search to identify optimal hyperparameters for each classical algorithm.

a) K-Nearest Neighbours (KNN): For KNN, we explore a range of neighbourhood sizes to balance bias and variance. Since KNN performance is sensitive to the choice of distance metric and weighting scheme, we fixed the metric to Euclidean distance and used distance-based weighting. The hyperparameters tuned were:

- `n_neighbors`: {1, 3, 5, 7, 10, 20, 50}
- `weights`: {`uniform`, `distance`}
- `metric`: {`euclidean`}

b) Random Forest: Random Forest was evaluated using varying numbers of decision trees and different configurations for tree depth and branching criteria. We aimed to capture a balance between underfitting and overfitting by tuning the following hyperparameters:

- `n_estimators`: {100, 300, 1000}
- `max_features`: {`sqrt`, `log2`}
- `max_depth`: {None, 10, 50, 100}
- `min_samples_split`: {2, 5, 10}
- `min_samples_leaf`: {1, 2, 4}

D. Sparse Autoencoders for Activation Analysis

Rather than merely identifying where the model processes code information, our approach aims to uncover how its processing differs between our two categories. We sample the internal activations on the residual stream.

By directly analysing model activations and decision patterns for different inputs, this method helps identify key features that influence code classification as buggy or its patched counterpart. These insights enhance the interpretability of the model's internal representations and decision-making processes.

To identify the most informative latent features for classification, we compute the average absolute activation difference between the buggy and patched versions across the dataset P with regard to our training set.

Given an SAE encoder function SAE$(\cdot)$ that maps an input activation vector to a sparse latent vector $c \in \mathbb{R}^k$, we define the feature-wise difference as:

$$\Delta_j = \frac{1}{p} \sum_{i=1}^{p} \left| \text{SAE}(p_{\text{buggy}})_j - \text{SAE}(p_{\text{patched}})_j \right|, \qquad (1)$$

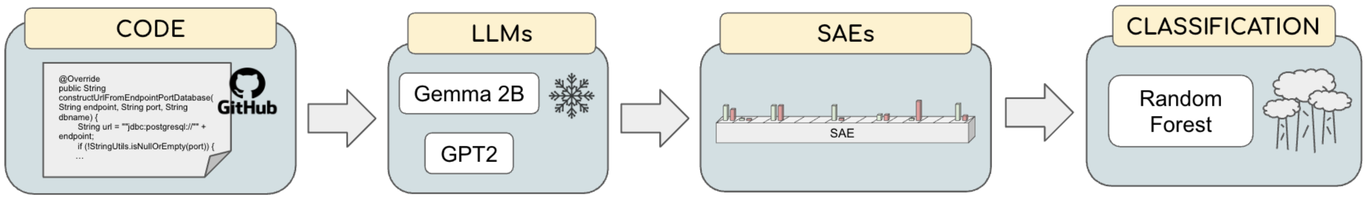
where Δ_j denotes the average activation difference of feature j across all N training code pairs. The operator SAE$(x)_j$ denotes the activation of feature j for input x.

We define the set of the most discriminative features as:

$$\text{TopK}(\Delta,\ k) = \{j_1, j_2, \ldots, j_k\} \qquad (2)$$
$$\text{such that} \quad \Delta_{j_1}, \ldots, \Delta_{j_k} \text{ are the top-}k \text{ values of } \Delta.$$

This selection ensures that we use only the most activation-sensitive features, those that consistently vary between buggy and patched versions, for our downstream classification. These features are then used as input to lightweight Random Forest classifiers. We summarise the full feature selection and classifier training pipeline in Algorithm 1.

In our experiments, we integrate pre-trained SAEs sourced from prior work for each model under study. For GPT-2 Small, we use the SAEs developed by Joseph Bloom [19], where each SAE replaces one of the 12 attention layers in the architecture. To facilitate feature-level analysis, we run inference over the entire training set and record the activations of each latent in the corresponding SAE. For Gemma 2B, we utilise the comprehensive suite of JumpReLU SAEs released



```
@Override
public String
constructUrlFromEndpointPortDatabase(
String endpoint, String port, String
dbname) {
        String url = ""jdbc:postgresql://"" +
endpoint;
        if (!StringUtils.isNullOrEmpty(port)) {
        ...
```

Fig. 5: Overview of the proposed methodology.

Algorithm 1 SAE-Based Bug Classification

Input: Paired dataset $D = \{(x^{(i)}_{\text{buggy}}, x^{(i)}_{\text{patched}})\}_{i=1}^{N}$, SAE encoder SAE$(\cdot)$, number of top features k

Output: Trained classifier $\hat{f}$

```
1:  begin
2:     Activation Difference Computation:
3:        Initialize $\Delta \in \mathbb{R}^k$ to zeros;
4:  for each $(x^{(i)}_{\text{buggy}}, x^{(i)}_{\text{patched}}) \in D$ do
5:           $c^{(i)}_{\text{buggy}} \leftarrow \text{SAE}(x^{(i)}_{\text{buggy}})$;
6:           $c^{(i)}_{\text{patched}} \leftarrow \text{SAE}(x^{(i)}_{\text{patched}})$;
7:           $\Delta \leftarrow \Delta + \left| c^{(i)}_{\text{buggy}} - c^{(i)}_{\text{patched}} \right|$;
8:  end for
9:        $\Delta \leftarrow \Delta / N$;
10:       Select top-$k$ most discriminative features: $\mathcal{F}_{\text{top}} = \text{TopK}(\Delta, k)$;
11:    Training Data Construction (using only $\mathcal{F}_{\text{top}}$):
12:       $\mathcal{D}_{\text{train}} = \emptyset$;
13: for each $(x^{(i)}_{\text{buggy}}, x^{(i)}_{\text{patched}}) \in D$ do
14:          $v_1 = \text{SAE}(x^{(i)}_{\text{buggy}})[\mathcal{F}_{\text{top}}]$    ▷ retain only TopK features
15:          $v_2 = \text{SAE}(x^{(i)}_{\text{patched}})[\mathcal{F}_{\text{top}}]$;
16:          $\mathcal{D}_{\text{train}} = \mathcal{D}_{\text{train}} \cup \{(v_1, 1), (v_2, 0)\}$;
17: end for
18:    Model Training:
19:       Train classifier $\hat{f}$ on $\mathcal{D}_{\text{train}}$;
20:    return $\hat{f}$;
21: end
```

by DeepMind as part of the Gemma Scope project [20], covering all layers of the Gemma 2B model.

We observe that a substantial fraction of SAE latents remain largely inactive across inputs. Specifically, each GPT-2 Small SAE contains 24,576 features, while SAEs in Gemma 2B contain 16,384 features. Due to the intrinsic sparsity encouraged by the SAE training objective, most latents are rarely activated: a phenomenon illustrated in Figure 6.

Predictive Utility Evaluation. After characterizing feature behaviour at the token level, we evaluate their predictive strength across the dataset.

We train Random Forest classifiers using the selected top-k features, assessing how well sparse latent activations can support downstream bug classification tasks. Performance is measured across varying values of of top-k, highlighting the trade-offs between feature sparsity and classification accuracy.

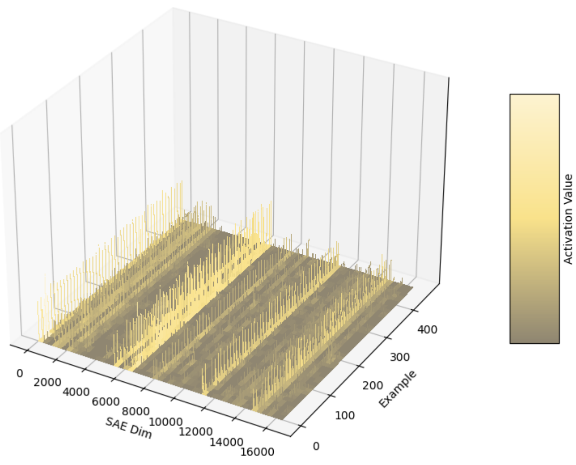


Fig. 6: Gemma 2B Layer 17 SAE activations throughout the buggy and safe versions of `HumanEval`

E. Fine-Tuning Neural Baselines

To benchmark the performance of our sparse feature-based approach against strong neural baselines, we fine-tune three transformer-based models for binary classification: GraphCodeBERT [21], ModernBERT-base [22], and ModernBERT-large. Each model is trained to distinguish between buggy and patched code snippets using our curated dataset of Java functions.

a) GraphCodeBERT: GraphCodeBERT is pre-trained on code and code-related natural language using a multi-modal objective. We fine-tune it for 20 epochs with a batch size of 16, a learning rate of 2e^{-5}, and weight decay of 0.01. We use the AdamW optimizer and apply evaluation at each epoch, retaining the best-performing checkpoint based on validation accuracy.

b) ModernBERT-base and ModernBERT-large: ModernBERT [22] is a recently proposed transformer architecture, with improvements in pre-training dynamics and token handling. We include both the base and large variants. ModernBERT-base contains approximately 110M parameters, while ModernBERT-large has around 345M parameters. We adopt the same training configuration as with GraphCodeBERT.

These models represent high-capacity, end-to-end fine-tuned baselines that directly contrast with our interpretable pipeline based on SAE activations. By including both a code-specialised transformer (GraphCodeBERT) and modern general-purpose models (ModernBERT), we provide a comprehensive view of model performance across paradigms.

IV. Results

Figure 7 provides a comprehensive visualisation of the F1 score (a) and accuracy (b) for each model.

Across all datasets, models leveraging representations from Gemma 2B achieved higher F1 Scores and accuracy metrics than the baselines. Notably, in the `Github-Bug Java` dataset, RF Gemma 2B attained an F1 Score of 0.89, outperforming all other configurations by a significant margin.

Interestingly, GPT-2 Small exhibited strong results with simple Random Forest models, suggesting that fine-grained information can be captured by simpler LLM, providing highly predictive information.

For the `Defects4J` dataset, although all methods showed lower performance relative to `Github-Bug Java`, GPT-2 Small models still provided a noticeable advantage over baselines. GPT-2 Small RF achieved an F1 Score of 0.76, significantly surpassing GraphCodeBERT and ModernBERT variants.

On the `HumanEval` dataset, results were more nuanced. Here, Gemma 2B RF led with a score of 0.67, suggesting that Gemma 2B's learned representations generalise better to unseen examples. It is essential to highlight that GPT-2 Small and Gemma 2B maintained strong performance despite the increased complexity and variability of code snippets.

A. Feature Importance Analysis

To provide further insight into the learned representations, we examine the feature importance derived from Random Forest classifiers applied at layer 0 for each model. Figure 9 presents the cumulative feature importance plots for GPT-2 Small and Gemma 2B.

As shown in Figures 9a and 9b, Gemma 2B achieves high cumulative importance using fewer features, indicating that its early-layer representations are more compact and discriminative. In contrast, the feature importance for GPT-2 Smallis distributed more gradually, suggesting broader but less concentrated utilisation of the feature space.

These findings, while not part of the primary evaluation, support the broader hypothesis that Gemma 2B encodes more salient and expressive code semantics, particularly in its early layers. Classical machine learning methods such as Random Forests can leverage this structured information effectively, even with minimal preprocessing.

B. Transferability

To further assess the robustness of our learned representations, we conduct a transferability study in a formal setting.

Let D_{source} and D_{target} denote two code datasets, each consisting of code snippet pairs as defined in Section II. We consider a model f_{source} trained and tuned on D_{source}. The F1 Score achieved by f_{source} when evaluated on dataset D is denoted by

$$\text{F1}(f_{\text{source}}, D). \tag{3}$$

We define the relative performance shift Δ when transferring f_{source} to D_{target} as:

$$\Delta(D_{\text{source}} \rightarrow D_{\text{target}}) = \frac{\text{F1}(f_{\text{source}}, D_{\text{target}}) - \text{F1}(f_{\text{source}}, D_{\text{source}})}{\text{F1}(f_{\text{source}}, D_{\text{source}})}. \tag{4}$$

Here, Δ captures the proportional change in F1 Score:

- A small absolute value $|\Delta|$ indicates strong transferability and robust feature extraction.
- A large negative Δ suggests overfitting to domain-specific patterns or limited cross-domain applicability.

Specifically, we evaluate cases such as $\Delta($`Github-Bug Java` $\rightarrow$ `Defects4J`$)$ and $\Delta($`Defects4J` $\rightarrow$ `Github-Bug Java`$)$, measuring how models generalise between different code corpora.

The results, illustrated in Figure 10, provide insight into the degree to which GPT-2 Small and Gemma 2B encode transferable, domain-independent representations. Our GPT-2 Small approach seems to have more degradation when transferring onto HumanEval. Throughout the other permutations of transferability settings between datasets, our approaches seem to be competitive, compared to transformer-based approaches, even surpassing (e.g. $\Delta($`HumanEval` $\rightarrow$ `Github-Bug Java`$)$.

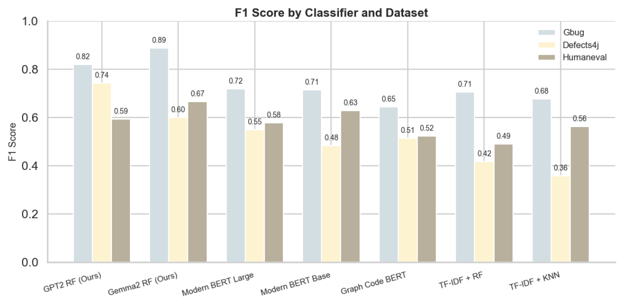
V. Discussion

Our study highlights several insights regarding the use of SAEs for bug detection in code without fine-tuning LLMs.

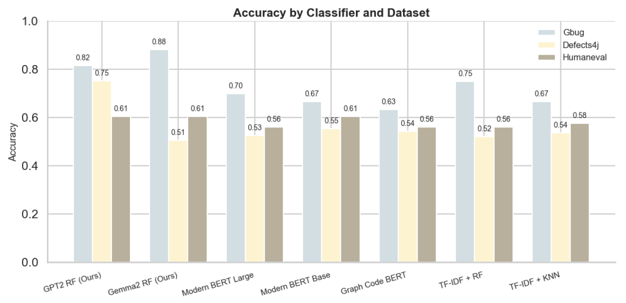
First, we observe that feature representations extracted from pre-trained LLMs, particularly through sparse encoding, can yield competitive or even superior bug detection performance compared to fully fine-tuned baselines. In particular, features extracted from Gemma 2B consistently outperform those from GPT-2 Small, suggesting that newer, larger models encode more discriminative and robust representations even without task-specific adaptation. This supports the emerging perspective that modern LLMs serve as strong feature extractors across diverse downstream tasks.

Second, our findings show that relatively simple models, such as random forests, are capable of effectively leveraging sparse LLM-derived features for classification. This result contrasts with the conventional reliance on deep fine-tuning and hints at an alternative paradigm for adapting foundation models: extracting and manipulating sparse internal representations instead of modifying model weights.

Third, the transferability analysis reveals that models built upon SAE features generalise reasonably well across different code datasets, though some degradation is expected. Notably, Gemma 2B's representations showed more consistent cross-dataset performance, suggesting that its latent space captures higher-level semantic properties rather than dataset-specific



(a) F1 Scores



(b) Accuracy

Fig. 7: Model Evaluation Results: F1 scores, accuracy for different models.

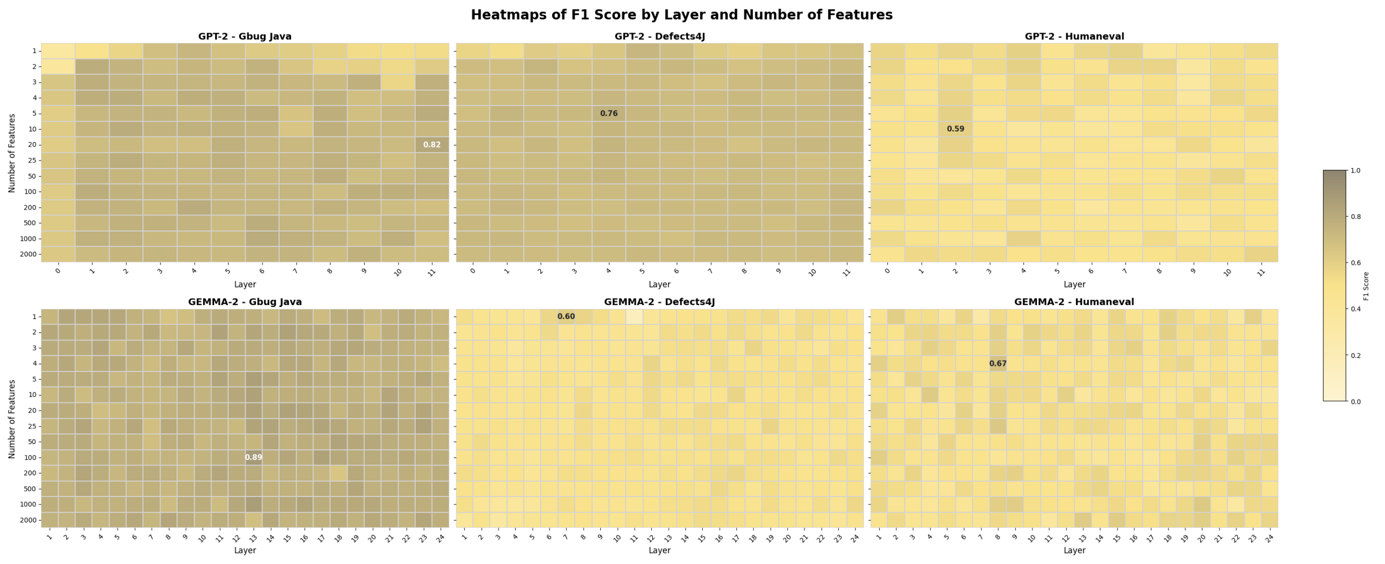


Fig. 8: Layer-wise classification performance heatmap using SAE-derived features. This heatmap visualises the F1 scores achieved by Random Forest classifiers trained on `Top-K` SAE features extracted from each layer of GPT-2 Small and Gemma 2B. Each cell corresponds to a specific model-layer combination, with colour intensity indicating predictive strength (darker is better).

artefacts. However, a non-trivial drop in transfer performance between certain dataset pairs highlights the challenge of domain shifts even within the realm of Java code.

While promising, these results warrant caution. As discussed in Section VII, recent findings indicate that SAE latents can appear interpretable even in untrained models. This raises the possibility that some of the extracted features, while predictive, may not correspond to meaningful computational circuits but instead reflect superficial patterns such as token statistics or architectural biases. Further work is needed to systematically disentangle these factors.

Additionally, our reliance on simple classification tasks (binary buggy vs. safe) might overestimate the true semantic alignment between SAE features and buggy concepts. Extending this approach to finer-grained buggy categorisation (e.g., distinguishing buffer overflows from injection flaws) would provide stronger evidence of semantic coherence.

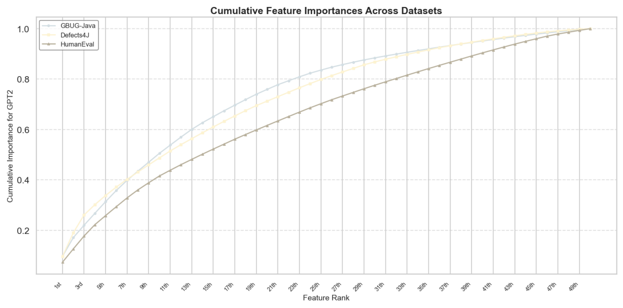
Finally, while our approach eliminates the need for fine-tuning, it still assumes access to SAE models trained on the residual stream activations. Although recent public efforts have made such models increasingly available, the cost and complexity of training high-quality SAEs remain non-trivial, especially for very large LLMs.

Overall, our results suggest a promising but nuanced role for sparse feature extraction in AI-driven software security. Rather than relying solely on black-box fine-tuned models, future systems may blend mechanistic interpretability with lightweight classical learning to build more trustworthy, auditable, and efficient AI tools for code analysis.

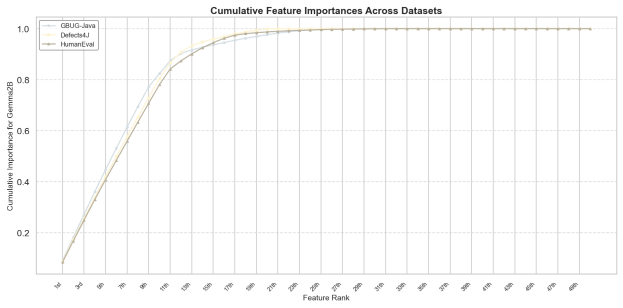
VI. Conclusion

We applied SAE to bug detection in Java, using models trained on LLM residual stream activations [19], [20]. This enabled us to extract meaningful latents that distinguish buggy from safe code without fine-tuning the underlying models.

Our results demonstrate that sparse representations derived from pre-trained LLMs, when paired with simple classical



(a) Cumulative Importance — GPT-2 Small



(b) Cumulative Importance — Gemma 2B

Fig. 9: Cumulative feature importance from Random Forest models applied to layer 0 representations.

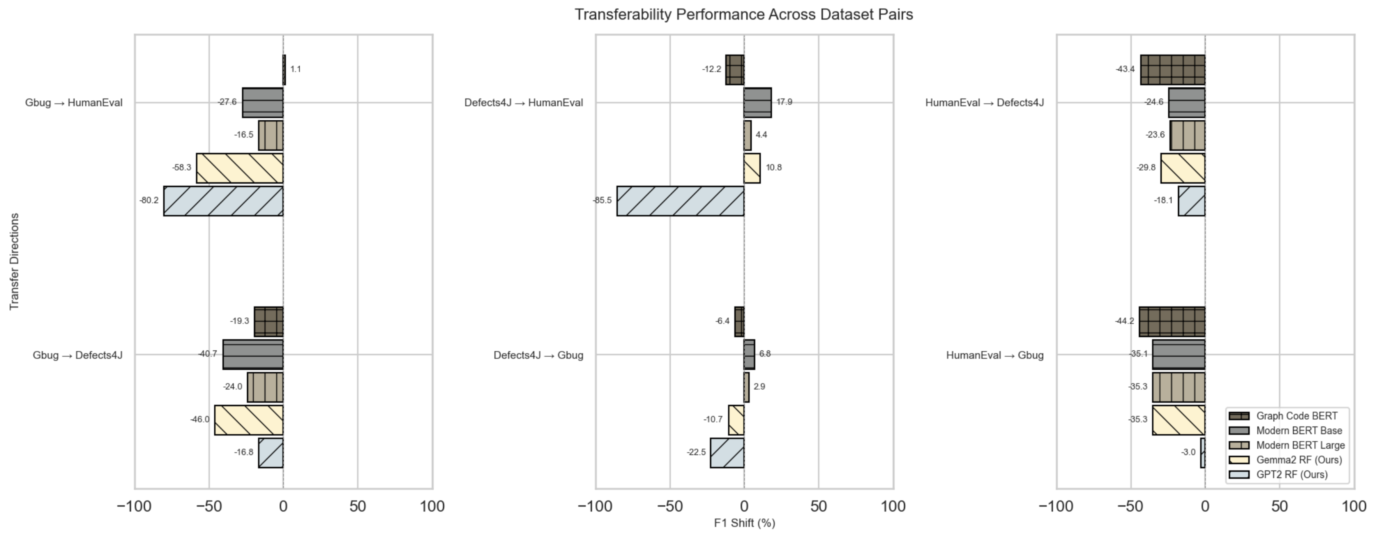


Fig. 10: Relative F1 Score shift Δ when transferring models across datasets.

classifiers like Random Forest, can outperform fine-tuned transformer-based baselines such as GraphCodeBERT and ModernBERT on multiple datasets. Furthermore, our qualitative case studies show that specific SAE features align with semantically meaningful buggy patterns, suggesting a promising direction for interpretable security analysis.

We also highlight the encouraging transferability of sparse features across datasets, supporting the notion that large pretrained models encode robust code representations that can be repurposed efficiently. However, challenges remain, particularly in verifying the true semantic depth of extracted features and generalising beyond binary classification tasks.

Overall, this study advances the case for lightweight and interpretable alternatives to full model fine-tuning in AI-driven software security. Future work should extend these findings across multiple programming languages, enhance feature-level interpretability, and explore hybrid approaches that integrate sparse neural features. Together, these directions aim to enable more trustworthy, scalable, and accountable bug and vulnerability detection systems. Our results lay a foundation for developing AI-driven security tools that prioritise interpretability and reliability alongside predictive performance.

VII. Threats to Validity

a) Interpretability of Sparse Autoencoders: While SAEs have recently gained popularity as a method for extracting interpretable representations from transformer models, recent evidence suggests caution when interpreting their outputs. Heap et. al [23] demonstrate that SAEs trained on the activations of randomly initialised transformers, models that have never been trained on meaningful data, can yield latent representations with auto-interpretability scores comparable to those from fully trained models. This finding challenges the assumption that interpretable SAE latents necessarily reflect meaningful or learned computational structure.

In our work, we use SAEs for downstream code classification. Yet, since similar SAE features can arise in untrained models, our results may reflect superficial input patterns or architectural biases, rather than model-computed semantics.

b) Cost of SAE Pretraining: While our method eliminates the need for fine-tuning the base LLM, it relies on access to SAEs trained on the model's residual stream activations. Training high-quality SAEs is itself a non-trivial task, often requiring substantial computational resources, especially for large models like Gemma 2B or newer architectures. Publicly available pre-trained SAEs [24] mitigate this cost for commonly studied models. However, for novel, proprietary, or rapidly evolving models, the requirement to train new SAEs remains a significant overhead. Future research could investigate lightweight or on-the-fly sparse feature extraction methods to reduce the computational burden further.

VIII. Related Work

At its core, mechanistic interpretability seeks to model neural networks as structured circuits: composable programs that manipulate interpretable features representing meaningful input properties [25], [26]. These features might correspond to linguistic constructs in Language Models (LMs) or security-relevant patterns in code-generation systems. Complementary to this perspective, propositional interpretability frames Artificial Intelligence (AI) cognition through the lens of propositional attitudes, belief-like states that govern knowledge representation and decision-making [27]. Together, these approaches enable a systematic analysis of how models internally represent and manipulate domain-specific concepts.

The integration of Explainable AI (XAI) presents a promising avenue for identifying specific model behaviours that lead to bugs or vulnerabilities, thereby enabling targeted interventions. For instance, by analysing the internal representations of LLMs during code generation, researchers can discern patterns associated with insecure coding practices and develop mechanisms to mitigate such risks. This enhanced interpretability not only fosters trust in the model but also facilitates the development of more secure AI systems.

One of the most important approaches in LLM-interpretability is the use of SAEs [24]. This Autoencoder variant is designed to learn efficient representations by enforcing sparsity constraints on the hidden layer activations. Unlike conventional Autoencoders, which mainly aim at reconstructing input data, SAEs incorporate an additional restriction that promotes the activation of only a small subset of neurons at any given moment [28]–[32].

Let $M \in \mathbb{R}^{d_{\text{hid}} \times d_{\text{in}}}$ denote the feature dictionary matrix. Given an input vector $x \in \mathbb{R}^{d_{\text{in}}}$, the SAE computes the hidden-layer activations $c \in \mathbb{R}^{d_{\text{hid}}}$ with $c = \text{ReLU}(Mx + b)$, where $b \in \mathbb{R}^{d_{\text{hid}}}$ is a bias vector. The reconstruction $\hat{x} \in \mathbb{R}^{d_{\text{in}}}$ is then obtained via $\hat{x} = M^{\top} c$. Here, c serves as a sparse latent representation of the input, with the non-zero coefficients indicating which dictionary atoms[1] are active in reconstructing x. The autoencoder is trained to minimise the objective: $\mathcal{L}(x) = \|x - \hat{x}\|_2^2 + \alpha \|c\|_1$, where α controls the sparsity of the reconstruction. The $\mathcal{L}_1$ loss term on c encourages our reconstruction to be a sparse linear combination of the dictionary features.

Contemporary research has expanded along several axes: improving SAE reconstruction fidelity [28]–[32], enhancing training efficiency [35], and developing evaluation benchmarks [36]. Interestingly, and as noted by Kantamneni et al. [37], there is a notable scarcity of studies presenting negative evidence regarding the effectiveness of SAEs on downstream tasks. To our knowledge, only a limited number of investigations have reported such findings. For instance, Chaudhary and Geiger [38] observed that SAE-derived latent representations perform worse than individual neurons when it comes to disentangling geographic features. Similarly, Farrell et al. [39] demonstrated that constraining related SAE latents is less effective than conventional methods for the task of unlearning sensitive bioweapon-related knowledge. Given the paucity of such counterexamples, it remains uncertain whether SAEs are on the verge of proving distinctly advantageous, or if their utility has been overstated.

[1]In the context of SAEs and dictionary learning, a *dictionary atom* refers to a learned basis vector used to reconstruct the input [33], [34].

IX. Future Work

Several directions remain open for future research. First, a deeper analysis of attention patterns could enhance our understanding of how different layers contribute to bug detection. We may uncover more precise circuits responsible for identifying insecure code structures by isolating attention heads and studying their activation behaviours.

Second, expanding our study to larger and more contemporary LLMs, such as Deepseek, would provide insights into whether our findings generalise across architectures. Investigating whether the same feature representations emerge in different models would strengthen the case for mechanistic interpretability in security applications.

Finally, our work suggests the potential for a hybrid approach combining mechanistic interpretability with traditional code analysis tools. By fusing neural representations with symbolic reasoning methods, we may develop more comprehensive frameworks for AI-driven bug detection.

We believe that advancing these directions will contribute to building safer and more transparent AI-assisted programming environments, ultimately reducing security risks in automated software development.

Acknowledgements

This research used computing resources partially provided by the Google Cloud Research Credits Program. This work was partially funded by projects AISym4Med (101095387) supported by Horizon Europe Cluster 1: Health, ConnectedHealth (n.º 46858), supported by Competitiveness and Internationalisation Operational Programme (POCI) and Lisbon Regional Operational Programme (LISBOA 2020), under the PORTUGAL 2020 Partnership Agreement, through the European Regional Development Fund (ERDF) and Agenda "Center for Responsible AI", nr. C645008882-00000055, investment project nr. 62, financed by the Recovery and Resilience Plan (PRR) and by European Union - NextGeneration EU, and also by FCT plurianual funding for 2020-2023 of LIACC (UIDB/00027/2020 UIDP/00027/2020).

References

[1] D. A. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, "A first step towards automated detection of buffer overrun vulnerabilities." in *NDSS*, vol. 20, no. 0, 2000, p. 0.

[2] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE transactions on dependable and secure computing*, vol. 1, no. 1, pp. 11–33, 2004.

[3] I. A. Elia, N. Antunes, N. Laranjeiro, and M. Vieira, "An analysis of openstack vulnerabilities," in *2017 13th European Dependable Computing Conference (EDCC)*. IEEE, 2017, pp. 129–134.

[4] A. Bessey, K. Block, B. Chelf, A. Chou, S. Hallem, C. Henri-Gros, S. McPeak, and D. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," in *Communications of the ACM*, vol. 53, no. 2. ACM, 2010, pp. 66–75.

[5] R. Bommasani, D. A. Hudson, E. Adeli, R. Altman, S. Arora, S. von Arx, M. S. Bernstein, J. Bohg, A. Bosselut, E. Brunskill *et al.*, "On the opportunities and risks of foundation models," *arXiv preprint arXiv:2108.07258*, 2021.

[6] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[7] C. Olah, A. Mordvintsev, and L. Schubert, "Feature visualization," *Distill*, vol. 2, no. 11, p. e7, 2017.

[8] S. Arora, Y. Li, Y. Liang, T. Ma, and A. Risteski, "Linear algebraic structure of word senses, with applications to polysemy," *Transactions of the Association for Computational Linguistics*, vol. 6, pp. 483–495, 2018.

[9] Z. Yun, Y. Chen, B. A. Olshausen, and Y. LeCun, "Transformer visualization via dictionary learning: contextualized embedding as a linear superposition of transformer factors," *arXiv preprint arXiv:2103.15949*, 2021.

[10] D. Chanin, J. Wilken-Smith, T. Dulka, H. Bhatnagar, and J. Bloom, "A is for absorption: Studying feature splitting and absorption in sparse autoencoders," *arXiv preprint*, 2024.

[11] T. Bricken, A. Templeton, J. Batson, B. Chen, and A. Jermyn, "Towards monosemanticity: Decomposing language models with dictionary learning," *Transformer Circuits Thread*, 2023. [Online]. Available: https://transformer-circuits.pub/2023/monosemantic-features/index.html

[12] A. Silva, S. Fang, and M. Monperrus, "Repairllama: Efficient representations and fine-tuned adapters for program repair," *arXiv preprint arXiv:2312.15698*, 2023.

[13] C. S. Xia and L. Zhang, "Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt," *arXiv preprint arXiv:2304.00385*, 2023.

[14] C. S. Xia, Y. Wei, and L. Zhang, "Practical program repair in the era of large pre-trained language models," *arXiv preprint arXiv:2210.14179*, 2022.

[15] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1161–1173.

[16] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: a database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 437–440. [Online]. Available: https://doi.org/10.1145/2610384.2628055

[17] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of code language models on automated program repair," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1430–1442.

[18] A. Silva, N. Saavedra, and M. Monperrus, "Gitbug-java: A reproducible benchmark of recent java bugs," in *Proceedings of the 21st International Conference on Mining Software Repositories*, ser. MSR '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 118–122. [Online]. Available: https://doi.org/10.1145/3643991.3644884

[19] J. I. Bloom, "Open source sparse autoencoders for all residual stream layers of gpt2-small," December 2023, aI Alignment Forum. [Online]. Available: https://www.alignmentforum.org/posts/f9EgfLSurAiqRJySD/open-source-sparse-autoencoders-for-all-residual-stream

[20] T. Lieberum, S. Rajamanoharan, A. Conmy, L. Smith, N. Sonnerat, V. Varma, J. Kramár, A. Dragan, R. Shah, and N. Nanda, "Gemma scope: Open sparse autoencoders everywhere all at once on gemma 2," *arXiv preprint arXiv:2408.05147*, 2024.

[21] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pre-training code representations with data flow," 2021. [Online]. Available: https://arxiv.org/abs/2009.08366

[22] B. Warner, A. Chaffin, B. Clavié, O. Weller, O. Hallström, S. Taghadouini, A. Gallagher, R. Biswas, F. Ladhak, T. Aarsen, N. Cooper, G. Adams, J. Howard, and I. Poli, "Smarter, better, faster, longer: A modern bidirectional encoder for fast, memory efficient, and long context finetuning and inference," 2024. [Online]. Available: https://arxiv.org/abs/2412.13663

[23] T. Heap, T. Lawson, L. Farnik, and L. Aitchison, "Sparse autoencoders can interpret randomly initialized transformers," 2025, arXiv preprint arXiv:2501.17727. [Online]. Available: https://arxiv.org/abs/2501.17727

[24] H. Cunningham, A. Ewart, L. Riggs, R. Huben, and L. Sharkey, "Sparse autoencoders find highly interpretable features in language models," *arXiv preprint arXiv:2309.08600*, 2023.

[25] R. Millière and C. Buckner, "A philosophical introduction to language models-part ii: The way forward," *arXiv preprint arXiv:2405.03207*, 2024.

[26] L. Sharkey, B. Chughtai, J. Batson, J. Lindsey, J. Wu, L. Bushnaq, N. Goldowsky-Dill, S. Heimersheim, A. Ortega, J. Bloom *et al.*, "Open problems in mechanistic interpretability," *arXiv preprint arXiv:2501.16496*, 2025.

[27] D. J. Chalmers, "Propositional interpretability in artificial intelligence," *arXiv preprint arXiv:2501.15740*, 2025.

[28] S. Rajamanoharan, T. Lieberum, N. Sonnerat, A. Conmy, V. Varma, J. Kramár, and N. Nanda, "Jumping ahead: Improving reconstruction fidelity with jumprelu sparse autoencoders," *arXiv preprint arXiv:2407.14435*, 2024.

[29] B. Bussmann, P. Leask, and N. Nanda, "Batchtopk sparse autoencoders," *arXiv preprint arXiv:2412.06410*, 2024.

[30] L. Gao, T. d. la Tour, H. Tillman, G. Goh, and R. Troll, "Scaling and evaluating sparse autoencoders," *arXiv preprint*, 2024.

[31] T. Conerly, A. Templeton, T. Bricken, J. Marcus, and T. Henighan, "Circuits updates - april 2024 — transformer circuits," 2024, [Accessed 15-02-2025]. [Online]. Available: https://transformer-circuits.pub/2024/april-update/index.html#training-saes

[32] T. Bricken, A. Templeton, J. Batson, B. Chen, A. Jermyn, T. Conerly, N. Turner, C. Anil, C. Denison, A. Askell, R. Lasenby, Y. Wu, S. Kravec, N. Schiefer, T. Maxwell, N. Joseph, Z. Hatfield-Dodds, A. Tamkin, K. Nguyen, B. McLean, J. E. Burke, T. Hume, S. Carter, T. Henighan, and C. Olah, "Towards monosemanticity: Decomposing language models with dictionary learning," 2023. [Online]. Available: https://transformercircuits.pub/2023/monosemantic-features/index.html

[33] J. Mairal, F. Bach, J. Ponce, and G. Sapiro, "Online learning for matrix factorization and sparse coding," 2010. [Online]. Available: https://arxiv.org/abs/0908.0050

[34] T. Bricken, A. Templeton, J. Batson, B. Chen, A. Jermyn, T. Conerly, N. Turner, C. Anil, C. Denison, A. Askell *et al.*, "Towards monosemanticity: Decomposing language models with dictionary learning," *Transformer Circuits Thread*, vol. 2, 2023.

[35] A. Mudide, J. Engels, E. J. Michaud, M. Tegmark, and C. S. de Witt, "Efficient dictionary learning with switch sparse autoencoders," *arXiv preprint*, 2024.

[36] J. Huang, Z. Wu, C. Potts, M. Geva, and A. Geiger, "Ravel: Evaluating interpretability methods on disentangling language model representations," *arXiv preprint*, 2024.

[37] S. Kantamneni, J. Engels, S. Rajamanoharan, M. Tegmark, and N. Nanda, "Are sparse autoencoders useful? a case study in sparse probing," *arXiv preprint arXiv:2502.16681*, 2025.

[38] M. Chaudhary and A. Geiger, "Evaluating open-source sparse autoencoders on disentangling factual knowledge in gpt-2 small," 2024. [Online]. Available: https://arxiv.org/abs/2409.04478

[39] E. Farrell, Y.-T. Lau, and A. Conmy, "Applying sparse autoencoders to unlearn knowledge in language models," *arXiv preprint arXiv:2410.19278*, 2024.