Spotting Code Mutation for Predictive Mutation Testing

Yifan Zhao
Key Lab of HCST (PKU), MOE; SCS
Peking University
Beijing, China
zhaoyifan@stu.pku.edu.cn

Yizhou Chen
Key Lab of HCST (PKU), MOE; SCS
Peking University
Beijing, China
yizhouchen@stu.pku.edu.cn

Zeyu Sun
National Key Laboratory of Space Integrated Information System, Institute of Software, Chinese Academy of Sciences
Beijing, China
zeyu.zys@gmail.com

Qingyuan Liang
Key Lab of HCST (PKU), MOE; SCS
Peking University
Beijing, China
liangqy@stu.pku.edu.cn

Guoqing Wang
Key Lab of HCST (PKU), MOE; SCS
Peking University
Beijing, China
guoqingwang@stu.pku.edu.cn

Dan Hao*
Key Lab of HCST (PKU), MOE; SCS
Peking University
Beijing, China
haodan@pku.edu.cn

ABSTRACT

Mutation testing is widely used to measure the test adequacy of a project. Despite its popularity, mutation testing is time-consuming and extremely expensive. To mitigate this problem, researchers propose Predictive Mutation Testing (PMT). Existing PMT approaches build classification models based on statistical program features or source code of programs to predict mutation testing results. Previous statistical feature-based PMT models need expensive overhead to collect dynamic features and neglect the rich information inherent in code text. Previous text-based PMT models extract essential code elements as input and outperform the feature-based models. However, they encode code text in a plain way. Therefore, they cannot sensitively capture subtle differences in mutants and they have difficulty in capturing the correlation between mutants and tests. To address these challenges, we propose a new model, SODA. SODA uses a new learning strategy, Mutational Semantic Learning, to make our model spot code mutation and its impact on test behavior. In particular, we employ a new sampling strategy to reinforce the corresponding relationship between mutants and tests by sampling same-mutant contrastive groups. Then we employ contrastive learning to make our model capture subtle differences in mutants. We conduct experiments to investigate the performance of SODA. The results demonstrate that both in the cross-project and cross-version scenarios, SODA achieves state-of-the-art classification performance (improves upon baselines by 5.32%-114.92% in kill-F1 score, 0.04%-25.54% in survive-F1 score, 4.25%-60.43% in accuracy) and has the lowest mutation score error.

*Dan Hao is the corresponding author. HCST is the abbreviation for "High Confidence Software Technologies". MOE is the abbreviation for "Ministry of Education". SCS is the abbreviation for "School of Computer Science".

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3695491

CCS CONCEPTS

• **Software and its engineering** → *Software maintenance tools.*

KEYWORDS

contrastive learning, mutation testing

ACM Reference Format:
Yifan Zhao, Yizhou Chen, Zeyu Sun, Qingyuan Liang, Guoqing Wang, and Dan Hao. 2024. Spotting Code Mutation for Predictive Mutation Testing. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3691620.3695491

1 INTRODUCTION

Mutation testing is widely used to measure the test adequacy of a project, and mutation analysis is recognized as one of the strongest test-adequacy criteria [2, 16, 39]. Mutation testing tools are utilized to make small changes to source code, resulting in mutants with seeded artificial faults that could mimic true faults [17, 24]. A test suite is run against these mutants to assess test adequacy by checking whether it can detect the seeded faults. If the test suite identifies a mutant behaving differently from the original program, the mutant is deemed "killed"; otherwise, it is deemed "survived". The "Mutation Score (MS)", defined as the ratio of killed mutants, serves as a measure of test adequacy. A test suite is considered more reliable if it achieves a higher mutation score, indicating its effectiveness in detecting a larger proportion of artificial faults.

Despite its popularity, mutation testing is time-consuming and extremely expensive [13]. For example, Google [39] and Meta [3] have adopted mutation testing to improve their test quality. However, in Google, the code base has approximately 2 billion lines of code, with more than 150 million test executions per day [39]. Mutating such a large complex software system and running all tests against the generated tremendous number of mutants is impractical. To mitigate this problem, researchers propose Predictive Mutation Testing (PMT) [49] to facilitate scalable mutation testing. PMT aims to predict the outcome of mutation testing without test execution. Specifically, a PMT model directly predicts whether a test suite can kill a given mutant, and then aggregates these predictions across all mutants to calculate the predicted mutation score.

Mutated method

```
1  private static Object remove(final Object array, final int index) {
2      final int length = getLength(array);
-      if (index < 0 || index >= length) {
3+     if (index <= 0 || index >= length) {
4          throw new IndexOutOfBoundsException("Index: " + index + ", Length: " + length);
5      }
6
7      final Object result = Array.newInstance(array.getClass().getComponentType(), length - 1);
8      System.arraycopy(array, 0, result, 0, index);
9      if (index < length - 1) {
10         System.arraycopy(array, index + 1, result, index, length - index - 1);
11     }
12
13     return result;
14 }
```

Test method-1

```
1  public void testRemoveAllNullObjectArray() {
2      ArrayUtils.remove((Object[]) null, 0);
3  }
```

Test method-2

```
1  public void testRemoveDoubleArray() {
2      double[] array;
3      array = ArrayUtils.remove(new double[] {1}, 0);
4      assertTrue(Arrays.equals(ArrayUtils.EMPTY_DOUBLE_ARRAY, array));
5      ...
6  }
```

Figure 1: Motivating example of two mutant-test pairs formed by the same mutant with two tests from the *commons-lang* project (commit d1a45e9). The mutated line is highlighted with color (i.e., the line is mutated from the red line to the green line). Test method-2 kills the mutant (i.e., detects the changed behavior) while Test method-1 does not.

Existing PMT approaches utilize machine learning algorithms and train classification models based on statistical features or source code of programs to predict mutation testing results. Previous statistical feature-based PMT models [31, 49] need expensive overhead to collect dynamic features without considering the rich information contained in code text, and fall short of practical utility [1, 12]. To address this issue, researchers have adopted more lightweight code text as input. They extract informative elements such as key code components (e.g., source method name) [20], contextual code information [12], and feed these elements into deep neural networks to predict mutation testing results. These text-based PMT models perform better than statistical feature-based PMT models [20].

However, these text-based PMT models face two challenges. First, **they cannot sensitively capture subtle differences in mutants.** A mutated method and its corresponding original method are often similar at the textual level but greatly differ in terms of program semantics. A mutation operator often makes small changes to the source code, leaving most parts of the code unchanged. As the example shown in Figure 1, the code is mutated by changing "<" to "<=". Therefore, the code representation of the mutant and source code will be very similar, making it hard for the model to learn the impact of the mutation operator and discriminate between the mutant and source code. Second, **they have difficulty in capturing the correlation between mutants and tests.** Even identical mutants may yield varying results for different test methods, because different test methods may cover distinct parts and paths of a program, leading to different program behavior. For example, in Figure 1, the test method *testRemoveAllNullObjectArray* does not kill the mutant, since the input array is null and its length is 0. Therefore, the *IndexOutOfBoundsException* is thrown regardless of the index value. However, the test method *testRemoveDoubleArray* kills the mutant, since the array length equals 1 and the *IndexOutOfBoundsException* should not have been thrown, but is thrown due to the change of the conditional statement. With the above challenges unaddressed, existing PMT models achieve unsatisfactory performance.

In order to address the above challenges, we propose a new model, **SODA** (**S**p**O**tting Code Mutation for Pre**D**ictive Mut**A**tion Testing). Different from previous PMT models, our model focuses on the mutated parts of the code and their impact on test behavior through contrasting killed and survived mutant-test pairs. In particular, we propose a new learning strategy, i.e., Mutational Semantic Learning, to make our model spot small code changes (i.e., code mutation) and their corresponding impact on tests. Mutational Semantic Learning comprises two steps, i.e., contrastive group sampling and contrastive learning. We employ contrastive group sampling to address the second challenge, i.e., we use the proposed sampling strategy to reinforce the corresponding relationship between mutants and tests. Each contrastive group consists of two mutant-test pairs. We construct same-mutant contrastive groups by sampling mutant-test pairs for the same mutant (e.g., the two pairs in Figure 1). Therefore, the model concentrates on the different part of the input (i.e., the test code) when comparing the two pairs in each same-mutant contrastive group. Focusing on the test code, the model learns to pay attention to the different test inputs and how the inputs trigger the altered program semantics, i.e., the impact of code mutation on the test results. To address the first challenge, we employ contrastive learning [5, 11, 19]. Since tests are written to verify the correctness of program semantics, survived pairs indicate the corresponding tests detect the unchanged part of the program semantics, while killed pairs indicate the corresponding tests detect the changed part of it. Our model learns to spot the changed program semantics by maximizing the representation agreement among the mutant-test pairs within the same categories (e.g., maximizing the representation similarity of killed mutant-test pairs) while minimizing it otherwise (e.g., minimizing the representation similarity between killed and survived mutant-test pairs). Finally, we train the model on the PMT classification task, refining the representation to better adapt our model to the PMT task.

We conduct experiments on Defects4J [16] to investigate the performance of SODA, the results demonstrate that both in the cross-project and cross-version scenarios, SODA achieves state-of-the-art classification performance and has the lowest predicted mutation score error. In particular, SODA outperforms the state-of-the-art model MutationBERT by 11.09% in kill-F1 score[1], 7.67% in survive-F1 score[2], 9.63% in accuracy in the cross-version scenario, and outperforms MutationBERT by 15.11% in kill-F1 score, 10.22% in survive-F1 score, 12.55% in accuracy in the cross-project scenario. In the cross-version scenario, SODA achieves the lowest mutation score error, 0.0292, smaller than half of MutationBERT's result. To help understand our model, we further conduct an ablation study, which substantiates the positive contribution of our contrastive group sampling and contrastive learning strategies.

This paper makes the following contributions:

- We propose a new model SODA for PMT. We propose Mutational Semantic Learning and the intuition behind it is to learn the impact of small code changes through contrasting same-mutant contrastive groups.
- We conduct a comprehensive experiment to verify the effectiveness of the proposed approach. The results show that

[1]F1 score for predicting killed mutants
[2]F1 score for predicting survived mutants

SODA significantly improves PMT effectiveness in both cross-version and cross-project scenarios.
- We make our code and data publicly available at **https://github.com/yifan-CodeDir/SODA**.

2 METHODOLOGY

Our approach aims to predict if a given test can kill a given mutant. Different from previous PMT models, our model spots the mutated parts of the code and their impact on test behavior via Mutational Semantic Learning. Figure 2 shows the overview of our approach. The workflow of our approach consists of three stages. In the first stage (Data Collection & Preprocessing in Section 2.1), we collect and preprocess the generated mutant data to construct text input for each mutant-test pair. In the second stage (Mutational Semantic Learning in Section 2.2), we train the representation layers of our model. Specifically, first, we construct same-mutant contrastive groups by sampling same-mutant positive (i.e., killed) mutant-test pairs from the training dataset. Second, we train the model to learn to represent mutant-test pairs by contrasting these groups. In the third stage (PMT Learning & Prediction in Section 2.3), we train the classification layer to classify the mutant-test pairs based on the representation layers learned in the previous stage.

Note that PMT could be conducted on two levels: the mutant-test suite level [1, 31, 49] and the mutant-test level [12, 20]. The former focuses on predicting whether a mutant can be detected by a whole test suite, while the latter focuses on predicting whether a mutant can be detected by a single test (i.e., predicting the results of mutant-test pairs). The latter produces more fine-grained results and could indicate the former's results: if at least one test is predicted to kill the mutant, the whole test suite is predicted to kill the mutant. Therefore, in this paper, we focus on predicting the results of mutant-test pairs and also present the results when aggregating them to the mutant-test suite level.

2.1 Data collection & Preprocessing

Data collection. The input to our model consists of each mutant-test pair, with the objective of accurately predicting the label of each pair (i.e., 0 for survived and 1 for killed). To collect such data, we start with a set of projects and their corresponding test suites. We use mutation tools to generate a series of mutants and run the test suites against them. The execution results of mutation testing are recorded as ground truth. In order to facilitate test-level prediction, we record fine-grained results, i.e. the results of running each test against each mutant. Consequently, this process yields a dataset comprising numerous mutant-test pairs, each annotated with its respective label. We further perform data selection and only retain the mutant-test pairs where the test covers the mutant. Because tests that do not cover the mutant will certainly not kill the mutant, and the prediction results may be overestimated if such tests are not removed according to previous work [1]. Finally, we get a set of mutant-test pairs for our model.

Preprocessing. To represent the code text of the mutant-test pairs, we implement the following preprocessing steps as shown in the example in Figure 3. First, we extract the source code of a mutated method. Then we perform a token-level comparison between the original and mutated methods. The differences are surrounded with special tokens "<BEFORE>", "<AFTER>", and "<ENDDIFF>" to denote the different tokens before and after mutation [12]. Subsequently, the mutated method is concatenated with each test to construct each mutant-test pair, where the mutated method and the test are separated with a special token "<SEP>" to indicate their different roles. Each mutant-test pair is labeled "killed" if the test detects the abnormal behavior of the mutant and is labeled "survived" otherwise. Hence, at the end of this stage, we get a series of processed mutant-test pairs along with their labels as our database.

2.2 Mutational Semantic Learning

In this stage, we aim to train the model to learn the changed program semantics and its impact on tests. In particular, to reinforce the relationship between mutants and tests and thus learn the impact of changed program semantics on tests, we employ same-mutant contrastive group sampling. We sample contrastive groups as training instances for contrastive learning. Each contrastive group consists of two mutant-test pairs. This process allows the model to learn from the nuanced differences between tests that interact with the same mutant. We present the details in Section 2.2.1 Then, we employ contrastive learning to analyze the differences between killed and survived mutant-test pairs and thus learn the changed program semantics. Specifically, we minimize the representation distance if the two mutant-test pairs in a contrastive group belong to the same category and maximize it otherwise, as described in Section 2.2.2. This process helps the model learn to distinguish between the changed and unchanged semantics of mutants reflected by killed and survived mutant-test pairs more effectively.

2.2.1 Contrastive Group Sampling. Contrastive group sampling aims to sample contrastive groups to serve as training instances for contrastive learning. Our objective is to construct informative groups of mutant-test pairs from which the model can effectively learn the changed semantics and its impact on tests.

The complete set of mutant-test pairs is composed of two types of mutant-test pairs, i.e., "killed" and "survived" depending on whether the test detects the changed program semantics. The two kinds of pairs indicate different runtime situations and we aim to learn the differences between them from the aspect of program semantics. In the hyperspace of vectorized representation, our intuition is to push away the vectorized representation of mutant-test pairs from different categories and pull together those from the same categories, so that when doing classification the different categories of mutant-test pairs can be discriminated easily. Therefore, we construct contrastive groups of mutant-test pairs to learn their commonalities if they are in the same categories and differences otherwise. Our representation training goal can be formally written as follows:

$$R^* = argmin\, G(R) \tag{1}$$

$$\begin{aligned} G(R) = &\sum_{i=1}^{N}\sum_{j=1}^{N} \mathbf{1}_{i\neq j}\mathbf{1}_{y_i=y_j}\, \mathrm{dis}(R(mt_i), R(mt_j)) \\ &- \sum_{i=1}^{N}\sum_{j=1}^{N} \mathbf{1}_{i\neq j}\mathbf{1}_{y_i\neq y_j}\, \mathrm{dis}(R(mt_i), R(mt_j)) \end{aligned} \tag{2}$$

where R represents the representation layers of the model, N is the total number of mutant-test pairs, $\mathbf{1}$ refers to an indicator function indicating whether a condition is satisfied, y_i represents the label of

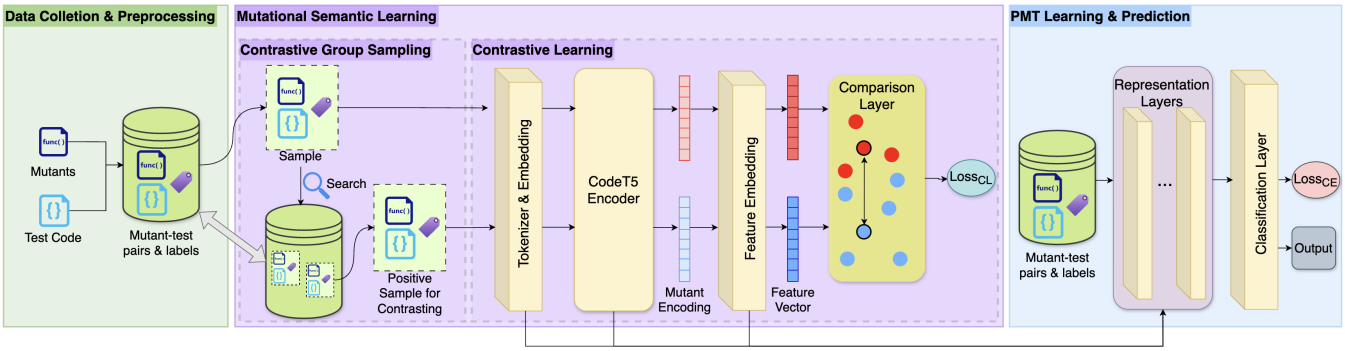


Figure 2: Overview of SODA

Mutated method

```
1  private static Object remove(final Object array, final int index) {
2      final int length = getLength(array);
-      if (index < 0 || index >= length) {
3+     if (index <= 0 || index >= length) {
4          throw new IndexOutOfBoundsException("Index: " + index + ", Length: " + length);
5      }
6
7      final Object result = Array.newInstance(array.getClass().getComponentType(), length - 1);
8      System.arraycopy(array, 0, result, 0, index);
9      if (index < length - 1) {
10         System.arraycopy(array, index + 1, result, index, length - index - 1);
11     }
12
13     return result;
14 }
```

Test method

```
1  public void testRemoveAllNullObjectArray() {
2      ArrayUtils.remove((Object[]) null, 0);
3  }
```

(a) An example of a mutant-test pair

Input Representation

```
1  <CLS>
2  private static Object remove(final Object array, final int index) {
3      final int length = getLength(array);
4      if (index <BEFORE>< <AFTER><=<ENDDIFF> 0 || index >= length) {
5          throw new IndexOutOfBoundsException("Index: " + index + ", Length: " + length);
6      }
7
8      final Object result = Array.newInstance(array.getClass().getComponentType(), length - 1);
9      System.arraycopy(array, 0, result, 0, index);
10     if (index < length - 1) {
11         System.arraycopy(array, index + 1, result, index, length - index - 1);
12     }
13
14     return result;
15 }
16 <SEP>
17 public void testRemoveAllNullObjectArray() {
18     ArrayUtils.remove((Object[]) null, 0);
19 }
```

(b) Input representation of the example

Figure 3: Data preprocess example

the mutant-test pair mt_i (i.e., 0 for survived and 1 for killed), dis$(\cdot)$ is a measurement calculating the distance between two vectors.

To achieve our goal, we first sample different kinds of contrastive groups for mutant-test pairs to learn their commonalities and differences. Each contrastive group consists of two mutant-test pairs. In particular, we consider two kinds of contrastive groups for mutant-test pairs, i.e., "killed-killed" and "killed-survived" groups, and aim to minimize the distance between the representation of the former while maximizing the distance between the representation of the latter. Note that we do not include "survived-survived" groups since the killed mutant-test pairs need to satisfy more strict constraints concerning execution, infection, propagation, and detection [45] and therefore contain more semantic information and stronger patterns, while the survived mutant-test pairs contain less semantic information. Formally, the contrastive label for the contrastive group (mt_i, mt_j) is as follows:

$$L_{CL}^{ij}\left(mt_i, mt_j\right) = \begin{cases} 0 & y_i \neq y_j \\ 1 & y_i = y_j = 1 \end{cases} \tag{3}$$

where y_i and y_j are labels (i.e., 0 for survived and 1 for killed) for the mutant-test pair mt_i and mt_j, respectively.

To sample contrastive groups, we then employ a same-mutant contrastive group sampling approach. We aim to learn the relationship between mutants and tests by contrasting different categories of mutant-test pairs for the same mutant, thereby highlighting the significant role of tests.

As shown in Algorithm 1, for each mutant-test pair mt_i, we aim to find an appropriate mutant-test pair mt_j to form a contrastive group (mt_i, mt_j) (Line 3-21). With a certain probability α, we start our search process (Line 4). Our search process begins by querying the database for a killed mutant-test pair that includes the same mutant as in mt_i but is paired with a different test that can kill the mutant (Line 5-8). If such a pair is found, it is selected as mt_j (Line 7). If no such pair exists, we then seek a killed mutant-test pair mt_j that shares the same mutated method as mt_i (Line 9-12). This includes pairs where the same mutation operator is applied to the same method but at a different position, or where different mutation operators are applied to the same method. With the same mutated method, the two mutants in mt_i and mt_j indicate different changed program semantics for the same method and thus help the model better capture the impact of code mutation. If we are still unable to find such a mutant-test pair, we resort to randomly selecting a killed mutant-test pair from the project (Line 13-15). In addition to the outlined search process, we occasionally select a killed mutant-test pair at random with a specific probability to aid the model in escaping local optima (Line 17-19). This strategy is grounded in two main reasons. First, we assume that mutant-test pairs with higher textual similarity (i.e., identical mutated method) pose greater challenges in discrimination. Conversely, a randomly selected mutant-test pair will typically exhibit lower textual similarity, making it easier to discriminate. This approach allows the model to initially "warm up" with simpler contrastive groups and then tackle more complex ones. Second, incorporating randomly selected mutant-test pairs enables the model to compare different mutated methods, thereby discovering the commonalities and differences

between different methods and their inputs, which enhances the model's generalizability. With the preceding process, the found mt_j is grouped with mt_i and added to $batch_{cg}$ (Line 20). The algorithm returns $batch_{cg}$ for further representation learning (Line 22).

Algorithm 1: Algorithm for Contrastive Group Sampling

Input: A batch of mutant-test pairs $batch_{mt}$, random probability α, set of all killed mutant-test pairs MT_k
Output: A batch of contrastive groups $batch_{cg}$

```
1  Function Sample_Contrastive_Group(batch_mt, α, MT_k):
2      batch_cg = []
3      for each mutant-test pair mt_i in batch_mt do
4          if rand(0,1) < α then
5              m = get_mutant(mt_i)
6              if has_kill_mutant_test_pair(m, MT_k) then
7                  mt_j = get_kill_mutant_test_pair(m, MT_k)
8              end
9              else if has_other_mutant_for_same_method(m,
                MT_k) then
10                 m' = get_mutant_for_same_method(m, MT_k)
11                 mt_j = get_kill_mutant_test_pair(m', MT_k)
12             end
13             else
14                 mt_j = random_sample(MT_k)
15             end
16         end
17         else
18             mt_j = random_sample(MT_k)
19         end
20         Append (mt_i, mt_j) to batch_cg
21     end
22     return batch_cg
```

2.2.2 Mutant Representation Learning through Contrastive Learning. After constructing the contrastive groups, the next step is to use these groups to train a model to distinguish between different categories of mutant-test pairs. We begin by employing the widely used pre-trained language model, CodeT5 [46], to encode the code text. We select CodeT5 for its outstanding performance on code-related tasks [4, 9, 23, 28, 48]. Since our task focuses on understanding code semantics and is a classification task rather than a generation task, we use the encoder part of CodeT5 to encode the mutant-test pairs.

For each contrastive group constructed from the preceding process, each mutant-test pair in the contrastive group is first tokenized and embedded to produce an initial representation of code. Then the initial representation is fed into the CodeT5 encoder to get an encoding of the mutant-test pair, denoted as $\boldsymbol{e}$. The feature embedding layer is then applied, aiming to further project the encoded representation to a low-dimensional hyperspace for distance calculation. Therefore, we use a multi-layer perceptron to produce a feature vector $\boldsymbol{f}$.

$$\boldsymbol{f} = \sigma(W_2 * \sigma(W_1 * \boldsymbol{e} + \boldsymbol{b_1}) + \boldsymbol{b_2}) \tag{4}$$

where σ is the ReLU activation function, W_1, W_2 are the trainable weight matrices and $\boldsymbol{b_1}$, $\boldsymbol{b_2}$ are the bias vectors.

In the comparison layer, the feature vector is linearly projected to a 2-dimensional contrastive representation vector.

$$\boldsymbol{v} = W_3 * \boldsymbol{f} + \boldsymbol{b_3} \tag{5}$$

where W_3 is the trainable weight matrix and $\boldsymbol{b_3}$ is the bias vector.

The representation of different mutant-test pairs in different categories is compared in the comparison layer. In particular, the contrastive loss is calculated based on vector $\boldsymbol{v}$, which is then used to update the encoding and feature embedding layers based on backpropagation to discriminate the mutant-test pairs. Formally, the contrastive loss Loss_{CL} for the contrastive group (mt_i, mt_j) is calculated as follows:

$$\begin{aligned}\text{Loss}_{CL}\left(\boldsymbol{v_i}, \boldsymbol{v_j}, L_{CL}^{ij}\right) = L_{CL}^{ij} \cdot \text{sim}\left(\boldsymbol{v_i}, \boldsymbol{v_j}\right)^2 \\ + \left(1 - L_{CL}^{ij}\right) \cdot \max\left(0, M - \text{sim}\left(\boldsymbol{v_i}, \boldsymbol{v_j}\right)\right)^2\end{aligned} \tag{6}$$

where $\boldsymbol{v_i}$ and $\boldsymbol{v_j}$ are the two contrastive representation vectors for mt_i and mt_j, respectively, L_{CL}^{ij} is the contrastive label shown in equation (3) indicating whether mt_i and mt_j are in the same category, $\text{sim}(\cdot)$ is the euclidean distance between the two vectors, M is the maximum margin indicating the threshold for dissimilarity.

2.3 PMT Learning & Prediction

In this stage, we aim to tune the model to adapt it to the PMT task. Besides leveraging the representation layers trained in the previous stage to provide informative input for the classification task, we also aim to incorporate the original information from the plain text. Therefore, we employ an additional embedding layer to embed the encoded text $\boldsymbol{e}$ and produce a classification vector $\boldsymbol{c}$. The feature vector $\boldsymbol{f}$ and the classification vector $\boldsymbol{c}$ are concatenated, and fed as input to a multi-layer perceptron to produce the final prediction result. Specifically, we use the following formula to calculate the prediction result.

$$\boldsymbol{c} = \sigma_1(W_5 * \sigma_1(W_4 * (\boldsymbol{e}) + \boldsymbol{b_4}) + \boldsymbol{b_5}) \tag{7}$$

$$\hat{y} = \sigma_2(W_6 * ([\boldsymbol{f} \oplus \boldsymbol{c}]) + \boldsymbol{b_6}) \tag{8}$$

where σ_1 and σ_2 are the ReLU and sigmoid activation functions, respectively, W_4, W_5, W_6 are the trainable weight matrices and $\boldsymbol{b_4}$, $\boldsymbol{b_5}$, $\boldsymbol{b_6}$ are the bias vectors, $[a \oplus b]$ means concatenating vector a with b.

We use the cross-entropy loss to train the model to predict mutation testing results. Cross-entropy loss is widely used in classification tasks [30], and we perform weighted cross-entropy loss [12] to better handle the well-known imbalanced data problem in PMT [49]. The formula for calculating cross-entropy loss is as follows:

$$\text{Loss}_{CE}(y, \hat{y}) = -(w_1 y \log(\hat{y}) + w_2(1 - y)\log(1 - \hat{y})) \tag{9}$$

where $\hat{y}$ is the predicted killing probability of the mutant-test pair, y is the true label of the mutant-test pair (0 for survived and 1 for killed), w_1 and w_2 are the weights for the positive samples (i.e., killed mutant-test pairs) and negative samples (i.e., survived mutant-test pairs), respectively, depending on the proportion of each class in the training set.

After the training phase, the predicted probability can be used to predict mutation testing results. Specifically, the model can predict the probability of a test killing a mutant, i.e., mutant-test level PMT. The model can also be applied to mutant-test suite level PMT by

aggregating all predicted test results for a given mutant. We use the threshold aggregation strategy since previous work [12] has shown its effectiveness. Formally, we use the following formula to predict the mutation testing result of a test suite T for a given mutant m:

$$p(m,T) = \begin{cases} 0 & \forall t \in T\ s.t.\ \hat{y}(m,t) < 0.1 \\ 1 & \exists t \in T\ s.t.\ \hat{y}(m,t) \geq 0.1 \end{cases} \quad (10)$$

where $\hat{y}(m,t)$ denotes the predicted killing probability of the mutant-test pair (m,t), $p(m,T) = 0$ denotes the mutant m is predicted to be "survived" under test suite T and "killed" otherwise. We choose 0.1 as the threshold because with 0.1 our model achieves the best performance on the validation dataset as shown in Section 4.5.

3 EXPERIMENT SETUP

In this section, we present the setup of the experiment, which aims to investigate the performance of our approach by comparing it against the state-of-the-art PMT approaches. In particular, as our approach and the state-of-the-art PMT approaches are all learning-based techniques, the evaluation is conducted in two scenarios, i.e., the cross-version and cross-project scenarios. Besides, we also conduct an ablation study and a parameter-tuning experiment to help better understand our approach. To sum up, we aim to answer the following five research questions.

RQ1. How does our approach perform in a cross-version scenario? In this RQ, we aim to investigate the effectiveness of SODA when trained with within-project data. For each project, we use mutant testing results on older versions to train a model and mutation testing results on the latest version to test the model [3].

RQ2. How does our approach perform in a cross-project scenario? In this RQ, we aim to investigate the effectiveness of SODA when trained with cross-project data. For each project, we use mutation testing results on other projects to train the model.

RQ3. How do different processes contribute to the effectiveness of our approach? In this RQ, we aim to investigate how the two key processes, i.e., contrastive group sampling and contrastive learning, contribute to the overall effectiveness of SODA. That is, we remove one of the processes each time and observe the effectiveness change of the modified SODA.

RQ4. How can our approach help reduce the time cost for mutation testing? Since PMT aims to reduce the time for mutation testing, in this RQ, we aim to investigate to what extent can we save time if applying PMT models to mutation testing. We compare the time costs to run the mutation tool and PMT models and present the predicted mutation score error of the PMT models.

RQ5. How do the parameters influence the effectiveness of our approach? In this RQ, we aim to investigate the impact of setting different thresholds for aggregation in test suite level PMT. Therefore, we tune the threshold from a discrete set on the validation dataset and observe the effectiveness change of the PMT models to select the best threshold for aggregation.

3.1 Baselines

We include the state-of-the-art PMT models Seshat [20] and MutationBERT [12] as our baselines. We do not include statistical feature-based PMT approaches [31, 49] since they are less effective than Seshat as shown in previous work [20]. However, to make a more comprehensive comparison available, we provide the results of a statistical feature-based baseline EPMT [1] on our website [42]. We here briefly introduce the included baselines.

Seshat is a deep-learning model utilizing Bi-directional GRU to process text and perform encoding. The model input is extracted from the code, considering essential textual inputs, i.e., test and source method names, code tokens of mutated statements, and mutation operators. Seshat is the first PMT model proposed to use textual input rather than statistical features. It shows promising results and outperforms feature-based PMT.

MutationBERT is proposed based on the pre-trained model CodeBERT [8]. It aims to include more code context to enhance the accuracy of PMT since the code beyond simply mutated lines and method names also contains useful information. Therefore, they construct text input by including the code surrounding a mutated line and the body of the test method, and then use the textual input and labels to fine-tune CodeBERT. MutationBERT is the state-of-the-art PMT model, showing better PMT results than Seshat [12].

Table 1: Subject programs

Project	Version	LoC	# Tests	Date	# Mut. Gen
commons-lang	1	21,788	2,291	2013-07-26	22.793
	10	20,433	2,198	2012-09-27	19,767
	20	18,967	1,876	2011-07-03	19,073
	30	17,660	1,733	2010-03-16	18,144
	40	17,435	1,643	2009-10-22	17,972
	50	17,760	1,720	2007-10-31	18,151
	60	16,920	1,590	2006-10-31	17,819
jfreechart	1	96,382	2,193	2010-02-09	81,006
	5	89,347	2,033	2008-11-24	75,024
	10	84,482	1,805	2008-06-10	71,052
	15	84,134	1,782	2008-03-19	70,647
	20	80,508	1,651	2007-10-08	67,479
	25	79,823	1,617	2007-08-28	66,766
gson	15	7,826	1,029	2017-05-31	5,044
	10	7,693	996	2016-05-17	4,775
	5	7,630	984	2016-02-02	4,722
	1	5,418	720	2010-11-02	2,295
commons-cli	30	2,497	354	2010-06-17	1,592
	20	1,989	148	2008-07-28	1,118
	10	2,002	112	2008-05-29	1,151
	1	1,937	94	2007-05-15	1,118
jackson-core	25	25,218	573	2019-01-16	30,010
	20	21,480	384	2016-09-01	25,257
	15	18,652	346	2016-03-21	21,599
	10	18,930	330	2015-07-31	22,089
	5	15,687	240	2014-12-07	18,610
	1	15,882	206	2013-08-28	16,982
commons-csv	15	1,619	290	2017-12-11	1,173
	10	1,276	200	2014-06-09	1,043
	5	1,236	189	2014-03-13	996
	1	806	54	2012-03-27	695

3.2 Subjects

We use six projects from Defects4J v2.0.0 [16] as our experimental subjects following previous work [20]. Defects4J is a widely-used dataset for many software testing tasks, such as mutation testing [12, 20], fault localization [21, 22, 26], program repair [6, 14, 27], etc. Each project has evolved over time, resulting in multiple versions. We select program versions that are multiples of 5 or 10 following previous work [20] to facilitate the cross-version scenario evaluation, and list them following time order. Note that a smaller version number does not always indicate an older version, e.g., in *commons-lang* and *jfreechart* smaller version numbers indicate

[3] We use "the latest version" to refer to the most recent version of a project in Defects4J, and use "older versions" to refer to the other versions of a project in Defects4J

more recent versions. Major is used as the Java mutation tool [15]. It consists of two main components, i.e., a compiler-integrated mutator and a mutation analyzer, to generate mutants and is widely used in mutation testing [20, 31, 43, 49]. Table 1 shows the information about our subject programs, including project name, version, LoC, number of tests, release date, and number of generated mutants.

To conduct evaluation in the cross-version scenario, for each project, we use the data from its latest version as testing data, data from its second latest version as validation data, and data from the other older versions as training data. To conduct evaluation in the cross-project scenario, we only consider the data from the latest version for each project. In particular, we use the five projects with the largest number of samples (i.e., *commons-lang*, *jfreechart*, *gson*, *jackson-core*, *commons-csv*) to conduct five-fold validation. Each time we use one of the projects as testing data, another project as validation data, and the rest three projects as training data.

3.3 Evaluation Metrics

We employ the widely used metrics for binary classification tasks, i.e., precision, recall, F1 score, and accuracy, as our evaluation metrics. Moreover, since survived and killed mutants are both important in PMT, we report kill precision, kill recall, kill-F1 score, survive precision, survive recall, and survive-F1 score to comprehensively understand the performance of our method. Besides, we include mutation score error as another metric [20, 49]. Mutation score is used to estimate the test adequacy of a program, which can be calculated by $MS = \frac{|K|}{|K|+|S|}$, where $|K|$ and $|S|$ are the number of killed mutants and survived mutants, respectively. The calculation formula of mutation score error is $E_{MS} = |MS_p - MS|$, where E_{MS} is the mutation score error, MS_p is the predicted mutation score, and MS is the ground truth of mutation score.

3.4 Implementation

We implement Seshat and MutationBERT utilizing their published replication packages. The parameters are set following their work [12, 20]. For SODA, in the Mutational Semantic Learning stage, we set the maximum number of training epochs to 50 and use the AdamW optimizer [25] with a batch size of 64. The learning rate is initialized to 5×10^{-5}. In the PMT Learning stage, we set the maximum number of training epochs to 30 and use the AdamW optimizer with a batch size of 128 and a learning rate of 1×10^{-5}. All three models' loss functions converge using these settings.

The CodeBERT and CodeT5 both have a maximum input token length limit of 512. Therefore, for CodeBERT, we truncate the token sequence of the mutant-test pair if its length exceeds 512 tokens following previous work [12]. For our method utilizing CodeT5, we use a context window surrounding the mutated line of code to handle this situation. Specifically, we use a 256 token-length context window where the mutated line lies in the central location to represent the mutant if its length exceeds 256 tokens. We truncate the token sequence of the test if its length exceeds 256 tokens. Then we combine the code and test to form the final input.

All the experiments are conducted on a workstation with 2 Intel Xeon Gold 5218R CPUs, 256GB RAM, and four 24G GPUs of GeForce RTX 3090, running Ubuntu 18.04 x64 OS. We implement our approach based on PyTorch V1.11.1 [37].

4 RESULTS AND ANALYSIS

4.1 RQ1. Cross-version scenario

We compare the performance of Seshat, MutationBERT, and SODA in the cross-version scenario, where data generated from the older versions are used to train and the data generated from the latest versions are used to test. Besides, we present prediction performance from two granularities, i.e., test matrix (i.e., mutant-test level prediction) and test suite (i.e., mutant-test suite level prediction), where the former provides fine-grained results, and the latter provides results aggregated based on the former. Table 2 presents the evaluation results with precision, recall, F1 score, and accuracy metrics. From the table, SODA performs the best both in terms of predicting test matrix and test suite results among the three models.

In particular, for predicting test matrix, SODA outperforms Seshat and MutationBERT. SODA improves upon Seshat by 7.09% in kill precision, 5.76% in kill recall, and 7.34% in kill-F1 score, respectively. In terms of predicting survived mutant-test pairs, SODA improves upon Seshat by 2.37% in survive precision, 3.03% in survive recall, and 2.94% in survive-F1 score, respectively. Besides, SODA achieves 4.25% higher accuracy than Seshat. SODA outperforms MutationBERT in terms of all metrics except kill recall. In particular, SODA improves upon MutationBERT by 20.02% in kill precision, and 11.09% in kill-F1 score, but performs 2.65% worse than MutationBERT in kill recall. This indicates that SODA works more conservatively, i.e., it tends to guarantee the predicted killed mutants are true positives while may produce more false negatives (i.e., some killed mutants are falsely predicted to survive). The effect can be eliminated by further running a confirmation check before presenting predicted-survived mutants to the user [12]. A confirmation check by running each predicted-survived mutant could be a practical application for PMT, ensuring the list presented to the developer is free of incorrect predictions [12]. This approach guarantees that the information provided is actionable and saves developers time in verifying the tool's results. Due to space limitations, we provide the results on our website [42]. Besides, SODA improves upon MutationBERT by 0.17% in survive precision, 13.13% in survive recall, 7.67% in survive-F1 score, and 9.63% in accuracy.

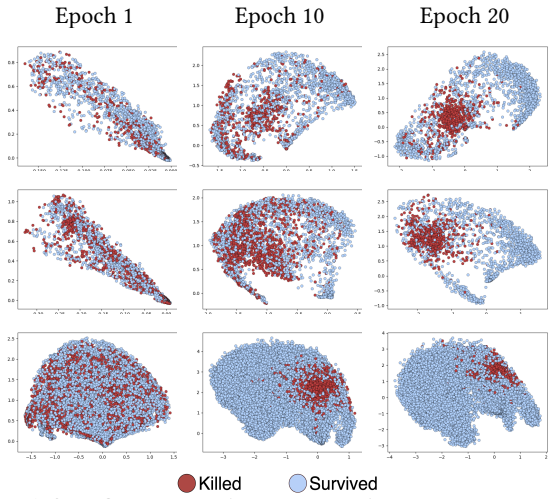
When it comes to predicting test suite level results, SODA consistently outperforms Seshat and MutationBERT. Thanks to its better performance at a finer granularity (test matrix granularity), it achieves the best predictive performance when the results are aggregated at the test suite level. In particular, SODA improves upon Seshat by 17.99% in kill-F1 score, 25.54% in survive-F1 score, and 20.57% in accuracy. Besides, SODA improves upon MutationBERT by 6.70% in kill-F1 score, 14.55% in survive-F1 score, and 8.23% in accuracy. For kill precision, kill recall, survive precision, and survive recall, SODA all outperforms Seshat and MutationBERT by a large margin. Note that different from the test matrix level, SODA performs better than MutationBERT in terms of kill recall at the test suite level, probably due to the following compounding effect. Although SODA may falsely predict some killed mutant-test pairs to survive, it accurately predicts at least one killed mutant-test pair within the same test suite. Consequently, the entire test suite is still correctly predicted to kill the mutant, contributing to high kill recall. In summary, SODA performs the best in the cross-version scenario no matter for the test matrix granularity or test suite granularity.

Table 2: Performance comparison in the cross-version scenario

Granularity	Model	Kill Precision	Kill Recall	Kill F1 Score	Survive Precision	Survive Recall	Survive F1 Score	Accuracy
Test Matrix	Seshat	0.7125	0.7918	0.7322	0.9128	0.8716	0.8880	0.8461
	MutationBERT	0.6357	**0.8603**	0.7075	0.9328	0.7938	0.8490	0.8046
	SODA	**0.7630**	0.8374	**0.7860**	**0.9344**	**0.8980**	**0.9141**	**0.8821**
Test Suite	Seshat	0.8518	0.6954	0.7636	0.5371	0.7393	0.6144	0.7219
	MutationBERT	0.8724	0.8233	0.8444	0.6574	0.7136	0.6734	0.8042
	SODA	**0.9230**	**0.8801**	**0.9010**	**0.7390**	**0.8072**	**0.7714**	**0.8704**

To better understand the process of Contrastive Learning (CL), we plot the 2-dimensional representation vector v on a surface and form a scatter plot in Figure 4. In the figure, we plot the change of the vector v in the CL process for project *commons-cli*, *commons-csv*, and *gson*, as shown in the first, second, and third rows, respectively. For each project, we plot the v when training is completed in epoch 1, epoch 10, and epoch 20, as shown in the first, second, and third columns, respectively. The killed mutant-test pairs are represented by the red points while the survived mutant-test pairs are represented by the blue points. From the figure, initially, the model has not yet learned how to discriminate between different categories of samples. Thus, the points are scattered chaotically, making it hard to classify them into their correct categories. However, as the training process progresses, the model begins to discern the differences between various categories of samples, leading to the increasing separation of the blue and red points. Specifically, the red points begin to cluster together while the blue points are increasingly pushed away from them. After 20 epochs, it becomes easy to separate most of the killed and survived mutant-test pairs since their representation is distributed differently. Therefore, SODA demonstrates enhanced proficiency in accurately classifying mutant-test pairs as killed within a certain scope in the hyperspace.

Figure 4: The distribution of representation vector at different epochs in the CL process. The first, second, and third rows for *commons-cli*, *commons-csv*, and *gson*, respectively.



4.2 RQ2. Cross-project scenario

Table 3 shows the performance comparison of the three models in the cross-project scenario. In this scenario, data generated from other subjects are used to train the model. From the table, SODA still achieves the best overall predictive performance. In particular, when predicting the test matrix level results, SODA outperforms Seshat by 70.46% in kill-F1 score, 0.04% in survive-F1 score, and 8.84% in accuracy. Besides, SODA outperforms MutationBERT by 15.11% in kill-F1 score, 10.22% in survive-F1 score, and 12.55% in accuracy. Compared to the performance in the cross-version scenario shown in Table 2, the predictive performance of the three models all drops from over 80% accuracy to less than 70% accuracy. This is consistent with results from previous work [12], because in the cross-version scenario, different versions of the same project have similar code base and tests, therefore the training data and testing data have similar data distribution but in the cross-project scenario they do not. When the results are aggregated to the test suite level, SODA outperforms Seshat by 114.92% in kill-F1 score, 5.41% in survive-F1 score, and 60.43% in accuracy. SODA outperforms MutationBERT by 5.32% in kill-F1 score, 9.72% in survive-F1 score, and 4.51% in accuracy. However, SODA performs worse than Seshat in terms of survive recall at both the test matrix and test suite levels, indicating that SODA produces more false positives. Here we observe a tendency of predicting mutants to survive for Seshat caused by data imbalance. At the test matrix level, 24% of mutant-test pairs are killed, while at the test suite level, 56% of mutants are killed. Therefore, it is an imbalanced dataset with different majority classes at different prediction levels. Since at the test matrix level survived mutant-test pairs constitute the majority class, Seshat tends to falsely predict mutants to be survived (more false negatives). Therefore Seshat has a high survive recall but low kill recall. At the test suite level, where the survived mutants constitute the minority class, Seshat maintains a high survive recall but exhibits a low kill recall and accuracy, which falls below 50%. However, SODA is more robust to the imbalanced data due to the weighted loss strategy, with a nearly 70% accuracy at the test suite level.

4.3 RQ3. Ablation Study

To investigate the effectiveness of our Contrastive Group Sampling (CGS) process and Contrastive Learning (CL) process, we further conduct an ablation study in the cross-project scenario. The results are shown in Table 4. Specifically, to investigate the contribution of CGS, we remove it and sample the contrastive groups by random sampling, denoted as SODA-w/o CGS in the table (i.e., the second row). To investigate the contribution of CL, we remove the CL process, meaning that we do not use contrastive learning and directly fine-tune CodeT5 with the collected PMT data. The variant is denoted as SODA-w/o CL in the table (i.e., the third row).

Table 3: Performance comparison in cross-project scenario

Granularity	Model	Kill Precision	Kill Recall	Kill F1 Score	Survive Precision	Survive Recall	Survive F1 Score	Accuracy
Test Matrix	Seshat	0.3598	0.3227	0.3128	0.7113	**0.7522**	0.7213	0.6068
	MutationBERT	0.4030	0.6019	0.4632	0.7706	0.5924	0.6546	0.5868
	SODA	**0.4620**	**0.6577**	**0.5332**	**0.8101**	0.6584	**0.7216**	**0.6604**
Test Suite	Seshat	0.7131	0.2681	0.3553	0.3775	**0.7908**	0.4855	0.4309
	MutationBERT	0.7332	0.7398	0.7251	0.4838	0.4972	0.4664	0.6615
	SODA	**0.7378**	**0.8169**	**0.7636**	**0.5889**	0.4865	**0.5117**	**0.6913**

From the table, both of the two processes positively contribute to the effectiveness of SODA. Without CGS, the performance of SODA decreases by 5.53% in kill-F1 score, 0.47% in survive-F1 score, and 2.62% in accuracy. This indicates that the sampling strategy for the contrastive group is important. By sampling same-mutant contrastive groups as training instances, the model can more effectively learn to discriminate between different categories of mutant-test pairs. It achieves this by determining whether tests trigger the altered program semantics, thereby enhancing its ability to classify with greater accuracy. Without CL, the performance of SODA decreases by 11.27% in kill-F1 score, 11.65% in survive-F1 score, and 13.11% in accuracy. This indicates that contrastive learning helps SODA learn a better representation of mutant-test pairs by pushing the representation of different categories of samples away and pulling together those of the same categories. With contrastive learning, the model significantly outperforms mere fine-tuning on downstream data, achieving substantial performance improvements. CL helps CodeT5 to better capture the fine-grained code edit operations and better understand the impact of the mutant operators on the corresponding methods and tests. In summary, both of the two processes positively contribute to the effectiveness of SODA and play important roles in improving SODA's performance.

4.4 RQ4. Efficiency

To investigate the efficiency of PMT models, we compare their inference time. To compare the time cost of predictive mutation testing with traditional mutation testing, we also report the execution time of Major to compute the full test matrix. The time costs for each tool to run are listed in Table 5, where the second column lists the execution time of Major, the third to fifth columns list the inference time of Seshat, MutationBERT, and SODA, respectively. Due to space limitations, we only list the inference time and leave the training time on our website [42]. Inference time is more important since the training process can be conducted offline.

From the table, Seshat is the most efficient model, costing less than 4 minutes to perform inference, while SODA costs less than 45 minutes to complete prediction. However, Major costs at most 1889 minutes to compute the full test matrix. MutationBERT and SODA take more time to predict mutation testing results, because they are built on larger pre-trained code models, and they have much longer input to process than Seshat (i.e., more contextual information). However, despite SODA being the least efficient model, it still achieves 30.06×, 24.38×, 21.71×, 20.16×, 124.96×, 41.00× speed up compared to Major on the six projects respectively.

Despite its high efficiency, how do the PMT models actually perform in predicting the mutation score? We use another metric, mutation score error, to measure their accuracy in predicting the mutation score. The averaged prediction results in the cross-version and cross-project scenarios are shown in Table 6. From the table, SODA predicts mutation score with the highest accuracy, with a mutation score error of 0.0292 and 0.1242 in the cross-version and cross-project scenarios, respectively. However, Seshat predicts mutation score with the worst accuracy, its mutation score error is 3.22 times and 2.40 times larger than SODA's in cross-version and cross-project scenarios, respectively. Therefore, while Seshat demonstrates notable efficiency and reduced overhead as a PMT model, SODA remains an attractive option, offering a balanced compromise between overhead and accuracy in predicting mutation scores, which can be proved by the confirmation check results [42]. With confirmation check, Seshat and SODA have comparable overheads, while SODA still achieves higher accuracy than Seshat.

4.5 RQ5. Parameter Tuning

To better understand the impact of threshold setting on the effectiveness of our model, we perform a parameter-tuning study by tuning the threshold and investigate the change in SODA's effectiveness. We tune the parameter on the validation set and use the best parameter setting in the aggregation strategy for RQ1 and RQ2.

Table 7 presents the performance of choosing different thresholds for aggregating results at the test suite level on the validation set. We tune the threshold from the set $\{0.10, 0.25, 0.50, 0.75, 0.90\}$. We only report kill-F1 score, survive-F1 score, and accuracy due to space limitations. From the table, we observe that all of the three models perform the best when the threshold is 0.10. Specifically, Seshat achieves the best kill-F1 score and accuracy at 0.10, while achieving the best survive-F1 score at 0.25. Since the kill-F1 score significantly decreases at 0.25 while the survive-F1 score remains stable at 0.10 and 0.25, we choose 0.10 as the threshold for Seshat. MutationBERT and SODA all achieve the highest kill-F1 score, survive-F1 score, and accuracy when the threshold is 0.10. On the validation set, SODA consistently outperforms Seshat and MutationBERT with different choices of threshold.

5 DISCUSSION

In this section, we explore the impact of false negatives and false positives, analyze evaluation results on more recent code bases, and examine the causes of incorrect predictions.

Table 8 shows the F_β score in the cross-version scenario (cross-project results are presented on the website [42] due to space limitations). Since F_β score assigns different weights to precision and recall, we present the results to further discuss the trade-offs between false negatives and false positives. We not only select the

Table 4: Ablation study results

	Kill Precision	Kill Recall	Kill F1 Score	Survive Precision	Survive Recall	Survive F1 Score	Accuracy
SODA	**0.4620**	**0.6577**	**0.5332**	**0.8101**	0.6584	**0.7216**	**0.6604**
SODA-w/o CGS	0.4566	0.6221	0.5037	0.7979	**0.6685**	0.7182	0.6431
SODA-w/o CL	0.4095	0.6428	0.4731	0.7754	0.5663	0.6375	0.5738

Table 5: Efficiency comparison

Project	Time to Run			
	Major (s)	Seshat (s)	MutationBERT (s)	SODA (s)
commons-lang	12,924	33	410	430
jfreechart	64,719	203	2,491	2,655
gson	16,738	69	723	771
commons-cli	1,290	5	57	64
jackson-core	113,343	78	763	907
commons-csv	5,289	9	114	129

Table 6: Average Mutation Score Error

	Mutation Score Error	
	Cross-version	Cross-project
Seshat	0.1233	0.4226
MutationBERT	0.0626	0.1488
SODA	**0.0292**	**0.1242**

Table 7: Results for tuning aggregation threshold

Model	Threshold	Kill F1 Score	Survive F1 Score	Accuracy
Seshat	0.10	**0.358**	0.541	**0.469**
	0.25	0.329	**0.542**	0.459
	0.50	0.260	0.531	0.430
	0.75	0.194	0.522	0.403
	0.90	0.169	0.518	0.393
MutationBERT	0.10	**0.763**	**0.592**	**0.703**
	0.25	0.735	0.580	0.679
	0.50	0.690	0.570	0.645
	0.75	0.509	0.542	0.552
	0.90	0.491	0.546	0.544
SODA	0.10	**0.834**	**0.659**	**0.783**
	0.25	0.816	**0.659**	0.767
	0.50	0.757	0.639	0.718
	0.75	0.671	0.612	0.654
	0.90	0.602	0.592	0.607

most commonly used β values 0.5 and 2 [7, 29, 44], but also include 0.2 and 5 for a wider range of evaluation. Note that $\beta < 1$ means giving more weight to precision, while $\beta > 1$ means giving more weight to recall. From the table, SODA consistently outperforms the baselines with varying β values, both from the perspectives of killed and survived mutants. SODA only performs slightly worse than MutationBERT when the kill recall is considered five times as important as kill precision (as shown by the kill F-5 score). For the confirmation check scenario, false positives are more important than false negatives, since false negatives could be further checked by running predicted-survived mutants while false positives could not, leaving some survived mutants undetected. In other words, survive F-2 score and survive F-5 score are more important measurements, and SODA performs well on both of them.

Table 9 presents evaluation results for newer versions of the six projects that are not included in the Defects4J dataset, assessing how model performance evolves over time (we evaluate models on these new versions with the trained models in RQ1). Due to compatibility issues with JUnit 5, we use the most recent versions compatible with JUnit 4 for mutation analysis with Major. Version details and cross-project results of these new versions are available on our website [42]. At the test matrix level, SODA outperforms Seshat and MutationBERT, achieving improvements in kill-F1 score (4.78% and 3.64%), survive-F1 score (6.45% and 8.49%), and accuracy (7.65% and 11.50%), respectively. These advantages largely persist at the test suite level, although SODA shows a slightly worse kill-F1 score compared to MutationBERT. Note that compared to earlier results in RQ1, SODA's performance in the test matrix level prediction declines by 19.63% in kill-F1 score, 9.55% in survive-F1 score, and 12.44% in accuracy. This decline also extends to the test suite level prediction. The performance drop is attributed to the extensive time gap, ranging from 2 to 12 years, between the training and testing data sets. This has resulted in significant differences in code and tests, leading to distinct distributions between the training and testing data. Consequently, the knowledge acquired from the training data inherently has limited applicability to the testing data, which restricts the improvement of SODA's performance. Despite these issues, SODA's lower kill recall compared to MutationBERT (indicating more false negatives) can potentially be mitigated by incorporating a confirmation check, as previously discussed in RQ1.

We analyze the overlap of mutant-test pairs correctly and incorrectly predicted between SODA and the other techniques and provide the Venn diagrams on our website [42]. From the figures, SODA has the smallest area of incorrect prediction and the largest area of correct prediction. To further understand the main reasons for incorrect prediction, we perform a case study by manually checking some samples. We find that insufficient context and missed clues are the two main reasons for incorrect prediction [12]. Since we only include the mutated methods and tests as input, SODA may not have sufficient information when some tests indirectly invoke the mutated methods (insufficient context) or SODA misses some code clues to make correct predictions (missed clues). Some examples are also provided on our website [42].

6 RELATED WORK

Mutation analysis is considered the strongest method for evaluating test suite efficacy [2], and has received focused and growing attention in both academic circles [13, 36, 43] and industry practice [3, 38–40]. Since the high computation cost of mutation testing makes it impractical in large complex systems, researchers have put dedicated efforts into reducing the cost of mutation testing. For example, mutation selection focuses on using a subset of mutant operators [33–35, 41] or choosing a subset of mutants [10, 47] to

Table 8: F_β score in the cross-version scenario

Granularity	Model	Kill F-0.2 Score	Kill F-0.5 Score	Kill F-2 Score	Kill F-5 Score	Survive F-0.2 Score	Survive F-0.5 Score	Survive F-2 Score	Survive F-5 Score
Test Matrix	Seshat	0.7135	0.7184	0.7580	0.7834	0.9105	0.9016	0.8774	0.8726
	MutationBERT	0.6403	0.6608	0.7767	**0.8391**	0.9245	0.8938	0.8139	0.7975
	SODA	**0.7641**	**0.7702**	**0.8097**	0.8303	**0.9325**	**0.9256**	**0.9040**	**0.8992**
Test Suite	Seshat	0.8441	0.8135	0.7208	0.7001	0.5421	0.5645	0.6804	0.7267
	MutationBERT	0.8699	0.8602	0.8310	0.8247	0.6578	0.6610	0.6942	0.7095
	SODA	**0.9213**	**0.9140**	**0.8883**	**0.8817**	**0.7414**	**0.7516**	**0.7924**	**0.8043**

Table 9: Performance on newer versions for the six projects

Granularity	Model	Kill F1 Score	Survive F1 Score	Accuracy
Test Matrix	Seshat	0.6029	0.7767	0.7175
	MutationBERT	0.6095	0.7621	0.7131
	SODA	**0.6317**	**0.8268**	**0.7724**
Test Suite	Seshat	0.6879	0.4596	0.6182
	MutationBERT	**0.8300**	0.5381	0.7626
	SODA	0.8281	**0.6117**	**0.7691**

estimate the results of full mutation analysis. Kaufman et al. [18] propose a measure of mutant usefulness and prioritize mutants accordingly to reduce the time required for improving test completeness. In industry, to reduce the time cost of mutation testing, researchers from Google [38, 39] propose a diff-based probabilistic approach to reduce the number of mutants by omitting those uninteresting or uncovered lines. Researchers from Meta [3] propose to semi-automatically learn new mutation operators to produce valuable mutants. Then they apply the learned mutation operators to generate mutants that are proven to expose a lack of testing. Google's approach complements ours, and combining our PMT with their mutant reduction methods could be a future research direction. Additionally, adapting our PMT to Meta's strategy with new mutation operators is another viable avenue for exploration.

Some researchers propose to build classification models to predict mutation testing results, which is the most relevant to our work. Zhang et al. [49] first propose the task of PMT. They propose dynamic and static features which are all program statistical information based on PIE theory [45]. Then they build a random forest model to predict the mutation testing results. Mao et al. [31] conduct a comprehensive study based on the work by Zhang et al. [49]. They consider more features, more classification algorithms, and more subjects. The empirical results show that package-level features are more important and random forest has advantages in accuracy as being a classification algorithm. Aghamohammadi et al. [1] examine the impact of unreached mutants in PMT. An unreached mutant is that the mutated line of the mutant is not executed by any test cases [32]. Therefore, an unreached mutant certainly survives, and adding them to the final results of PMT may cause a bias, making the presented PMT results better than they really are. Therefore, they propose to remove those unreached mutants and revisit the PMT performance. Since collecting program statistical information is expensive, Kim et al. [20] propose to use information from code text to predict mutation testing results. They extract useful code text such as a mutated line and use Bi-directional GRU to process the text information. They build a deep neural network model named Seshat, which is shown to outperform previous approaches. Jain et al. [12] propose to use a pre-trained code model to predict mutation testing results. They use contextual information from test methods and mutants as input and fine-tune CodeBERT [8] with PMT data. Different from all the approaches above, we propose to use Mutational Semantic Learning to learn the representation of mutant-test pairs by contrasting different categories of pairs and thus learn the fine-grained textual features.

7 THREATS TO VALIDITY

Threats to internal validity mainly come from the implementation of the previous approaches and our approach. To reduce the threats, we use the code published by prior work [12, 20], and use the mature machine learning library PyTorch to implement our approach. We release our code for ease of inspection and replication.

Threats to external validity mainly come from the subjects and the generated mutants and tests. To reduce the threats, we use the dataset published from previous work [20]. The subjects come from Defects4J, which is a widely used dataset for many software engineering tasks and is considered representative of real-world projects, enhancing our confidence in the generalizability of our findings. However, using only the six projects from the Defects4J dataset could potentially threaten the external validity of our findings. To address this, we conduct experiments on newer versions of the six projects that are not included in the Defects4J dataset, and we plan to validate our methodology on additional projects in future work[4]. Besides, since Major currently only supports JUnit 3 and JUnit 4, our experiments exclude projects that utilize JUnit 5 for testing, which may also contribute to threats to external validity.

Threats to construct validity mainly come from the metrics we use to evaluate approaches. To reduce the threats, we employ widely used metrics for classification tasks in machine learning, i.e., precision, recall, and F1 score. We explore a range of β values when calculating the F_β score to discuss the impact of false negatives and false positives. We also include a domain-specific metric, mutation score error, to more comprehensively evaluate the approaches.

8 CONCLUSION

In this paper, we propose SODA, the first mutational semantic-sensitive PMT approach. SODA learns the mutant-test pair representation by sampling contrastive groups and comparing mutant-test pairs of different categories. We evaluate SODA in cross-version and cross-project scenarios and the results show that SODA achieves state-of-the-art performance. Using SODA, developers can more accurately get PMT results with over 20× speed up.

ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China (Grant Nos. 62232001, 62372005, 62402482).

[4]We have begun preliminary evaluations on the *commons-codec* project, which serves as an out-of-sample test. The initial results detailed on our website [42] are promising.

REFERENCES

[1] Alireza Aghamohammadi and Seyed-Hassan Mirian-Hosseinabadi. 2021. An ensemble-based predictive mutation testing approach that considers impact of unreached mutants. *Software Testing, Verification and Reliability* 31, 7 (2021), e1784.

[2] James H Andrews, Lionel C Briand, Yvan Labiche, and Akbar Siami Namin. 2006. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering* 32, 8 (2006), 608–624.

[3] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. 2021. What it would take to use mutation testing in industry—a study at facebook. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 268–277.

[4] Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar T Devanbu, and Baishakhi Ray. 2022. Natgen: generative pre-training by "naturalizing" source code. In *Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering*. 18–30.

[5] Yizhou Chen, Zeyu Sun, Zhihao Gong, and Dan Hao. 2024. Improving Smart Contract Security with Contrastive Learning-based Vulnerability Detection. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–11.

[6] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering* 47, 9 (2019), 1943–1959.

[7] Mohamed El Kerdawy, Mohamed El Halaby, Afnan Hassan, Mohamed Maher, Hatem Fayed, Doaa Shawky, and Ashraf Badawi. 2020. The automatic detection of cognition using eeg and facial expressions. *Sensors* 20, 12 (2020), 3516.

[8] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[9] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. 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*. 935–947.

[10] Rahul Gopinath, Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. 2015. How hard does mutation analysis have to be, anyway?. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 216–227.

[11] Beliz Gunel, Jingfei Du, Alexis Conneau, and Veselin Stoyanov. 2020. Supervised Contrastive Learning for Pre-trained Language Model Fine-tuning. In *International Conference on Learning Representations*.

[12] Kush Jain, Uri Alon, Alex Groce, and Claire Le Goues. 2023. Contextual Predictive Mutation Testing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 250–261.

[13] Yue Jia and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2010), 649–678.

[14] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1161–1173.

[15] René Just. 2014. The Major mutation framework: Efficient and scalable mutation analysis for Java. In *Proceedings of the 2014 international symposium on software testing and analysis*. 433–436.

[16] René Just, Darioush Jalali, and Michael D Ernst. 2014. 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*. 437–440.

[17] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 654–665.

[18] Samuel J Kaufman, Ryan Featherman, Justin Alvin, Bob Kurtz, Paul Ammann, and René Just. 2022. Prioritizing mutants to guide mutation testing. In *Proceedings of the 44th International Conference on Software Engineering*. 1743–1754.

[19] Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. 2020. Supervised contrastive learning. *Advances in neural information processing systems* 33 (2020), 18661–18673.

[20] Jinhan Kim, Juyoung Jeon, Shin Hong, and Shin Yoo. 2022. Predictive mutation analysis via the natural language channel in source code. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 4 (2022), 1–27.

[21] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*. 169–180.

[22] Yi Li, Shaohua Wang, and Tien Nguyen. 2021. Fault localization with code coverage representation learning. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 661–673.

[23] Bo Lin, Shangwen Wang, Zhongxin Liu, Yepang Liu, Xin Xia, and Xiaoguang Mao. 2023. Cct5: A code-change-oriented pre-trained model. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1509–1521.

[24] Richard J Lipton and Frederick G Sayward. 1979. *Mutation analysis*.

[25] Ilya Loshchilov and Frank Hutter. 2018. Fixing weight decay regularization in adam. (2018).

[26] Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. 2021. Boosting coverage-based fault localization via graph-based representation learning. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 664–676.

[27] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*. 101–114.

[28] Parvez Mahbub, Ohiduzzaman Shuvo, and Mohammad Masudur Rahman. 2023. Explaining software bugs leveraging code structures in neural machine translation. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 640–652.

[29] Etienne Manderscheid and Matthias Lee. 2023. Predicting customer satisfaction with soft labels for ordinal classification. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 5: Industry Track)*. 652–659.

[30] Anqi Mao, Mehryar Mohri, and Yutao Zhong. 2023. Cross-entropy loss functions: Theoretical analysis and applications. In *International Conference on Machine Learning*. PMLR, 23803–23828.

[31] Dongyu Mao, Lingchao Chen, and Lingming Zhang. 2019. An extensive study on cross-project predictive mutation testing. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 160–171.

[32] Pedro Reales Mateo and Macario Polo Usaola. 2015. Reducing mutation costs through uncovered mutants. *Software Testing, Verification and Reliability* 25, 5-7 (2015), 464–489.

[33] Aditya P Mathur. 1991. Performance, effectiveness, and reliability issues in software testing. In *1991 The Fifteenth Annual International Computer Software & Applications Conference*. IEEE Computer Society, 604–605.

[34] A Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H Untch, and Christian Zapf. 1996. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 5, 2 (1996), 99–118.

[35] A Jefferson Offutt, Gregg Rothermel, and Christian Zapf. 1993. An experimental evaluation of selective mutation. In *Proceedings of 1993 15th international conference on software engineering*. IEEE, 100–107.

[36] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation testing advances: an analysis and survey. In *Advances in computers*. Vol. 112. Elsevier, 275–378.

[37] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[38] Goran Petrović and Marko Ivanković. 2018. State of mutation testing at google. In *Proceedings of the 40th international conference on software engineering: Software engineering in practice*. 163–171.

[39] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. 2021. Practical mutation testing at scale: A view from google. *IEEE Transactions on Software Engineering* 48, 10 (2021), 3900–3912.

[40] Ana B Sánchez, José A Parejo, Sergio Segura, Amador Durán, and Mike Papadakis. 2024. Mutation Testing in Practice: Insights from Open-Source Software Developers. *IEEE Transactions on Software Engineering* (2024).

[41] Akbar Siami Namin, James H Andrews, and Duncan J Murdoch. 2008. Sufficient mutation operators for measuring test effectiveness. In *Proceedings of the 30th international conference on Software engineering*. 351–360.

[42] SODA. 2024. Replication package. https://github.com/yifan-CodeDir/SODA. Accessed: 2024-09-10.

[43] Zhao Tian, Junjie Chen, Qihao Zhu, Junjie Yang, and Lingming Zhang. 2022. Learning to construct better mutation faults. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.

[44] Siddhanth Tripathi, Sinchana Shetty, Somil Jain, and Vanshika Sharma. 2021. Lung disease detection using deep learning. *Int. J. Innov. Technol. Explor. Eng* 10, 8 (2021), 1–10.

[45] Jeffrey M. Voas. 1992. PIE: A dynamic failure-based technique. *IEEE Transactions on software Engineering* 18, 8 (1992), 717.

[46] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. 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. 8696–8708.

[47] W Eric Wong and Aditya P Mathur. 1995. Reducing the cost of mutation testing: An empirical study. *Journal of Systems and Software* 31, 3 (1995), 185–196.

[48] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1482–1494.

[49] Jie Zhang, Ziyi Wang, Lingming Zhang, Dan Hao, Lei Zang, Shiyang Cheng, and Lu Zhang. 2016. Predictive mutation testing. In *Proceedings of the 25th international symposium on software testing and analysis*. 342–353.