SampleFix: Learning to Generate Functionally Diverse Fixes

Hossein Hajipour[1], Apratim Bhattacharyya[2], Cristian-Alexandru Staicu[1], and Mario Fritz[1]

[1] CISPA Helmholtz Center for Information Security, Germany
[2] Max Planck Institute for Informatics, Germany

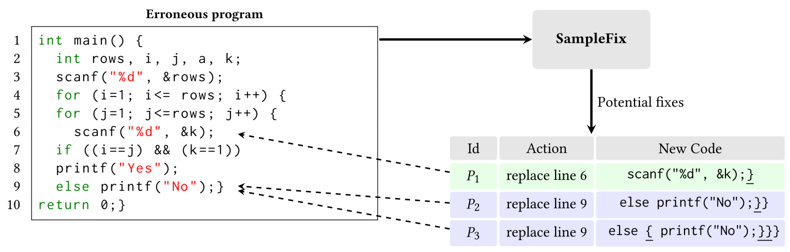
Abstract. Automatic program repair holds the potential of dramatically improving the productivity of programmers during the software development process and correctness of software in general. Recent advances in machine learning, deep learning, and NLP have rekindled the hope to eventually fully automate the process of repairing programs. However, previous approaches that aim to predict a single fix are prone to fail due to uncertainty about the true intend of the programmer. Therefore, we propose a generative model that learns a *distribution* over potential fixes. Our model is formulated as a deep conditional variational autoencoder that can efficiently sample fixes for a given erroneous program. In order to ensure *diverse* solutions, we propose a novel regularizer that encourages diversity over a semantic embedding space. Our evaluations on common programming errors show for the first time the generation of diverse fixes and strong improvements over the state-of-the-art approaches by fixing up to 45% of the erroneous programs. We additionally show that for the 65% of the repaired programs, our approach was able to generate multiple programs with diverse functionalities.

Keywords: Program repair · Generative models · Conditional variational autoencoder.

1 Introduction

Software development is a time-consuming and expensive process. Unfortunately, programs written by humans typically come with bugs, so significant effort needs to be invested to obtain code that is only likely to be correct. Debugging is also typically performed by humans and can contain mistakes. This is neither desirable nor acceptable in many critical applications. Therefore, automatically locating and correcting program errors [11] offers the potential to increase productivity as well as improve the correctness of software.

Advances in deep learning [17,18], computer vision [9,26], and NLP [30,3] have dramatically boosted the machine's ability to automatically learn representations of natural data such as images and natural language contents for various tasks. Deep learning models also have been successful in learning the distribution over continuous [29,16] and discrete data [21,14], to generate new and diverse data points [10]. These advances in machine learning and the advent of large corpora of source code [1] provide new opportunities toward harnessing deep learning methods to understand, generate, or debug programs.



```
1  int main() {
2    int rows, i, j, a, k;
3    scanf("%d", &rows);
4    for (i=1; i<= rows; i++) {
5    for (j=1; j<=rows; j++) {
6      scanf("%d", &k);
7    if ((i==j) && (k==1))
8    printf("Yes");
9    else printf("No");}
10 return 0;}
```

Id	Action	New Code
P_1	replace line 6	`scanf("%d", &k);}`
P_2	replace line 9	`else printf("No");}}`
P_3	replace line 9	`else { printf("No");}}}`

Fig. 2: SampleFix captures the inherent ambiguity of the possible fixes by sampling multiple potential fixes for the given erroneous real-world program. Potential fixes with the same functionality are highlighted with the same color and the newly added tokens are underlined.

Prior works in automatic program repair predominantly rely on expert-designed rules and error models that describe the space of the potential fixes [27,8]. Such hand-designed rules and error models are not easily adaptable to the new domains and require a time-consuming process.

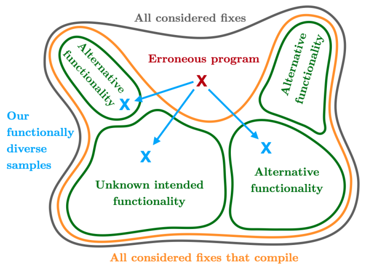

All considered fixes
Erroneous program
Alternative functionality
Alternative functionality
Our functionally diverse samples
Unknown intended functionality
Alternative functionality
All considered fixes that compile

Fig. 1: Our SampleFix approach with diversity regularizer promotes sampling of diverse fixes, that account for the inherent uncertainty in the automated debugging task.

In contrast, learning-based approaches provide an opportunity to adapt such models to the new domain of errors. Therefore, there has been an increasing interest to carry over the success stories of deep learning in NLP and related techniques to employ learning-based approaches to tackle the "common programming errors" problem [13,12]. Such investigations have included compile-time errors such as missing scope delimiters, adding extraneous symbols, using incompatible operators. Novice programmers and even experienced developers often struggled with these types of errors [25], which is usually due to lack of attention to the details of programs and/or programmer's inexperience.

Recently, Gupta et al. [13] proposed a deep sequence to sequence model called DeepFix where, given an erroneous program, the model predicts the locations of the errors and a potential fix for each predicted location. The problem is formulated as a deterministic task, where the model is trained to predict a single fix for each error. However, different programs – and therefore also their fixes – can express the same functionality. Besides, there is also uncertainty about the intention of the programmer. Figure 1 illustrates the issue. Given an erroneous program (buggy program), there is a large number of programs within a certain edit distance. A subset of these, will result in successful compilation. The remaining programs will still implement different functionalities and – without additional information or assumptions – it is impossible to tell which program/-functionality was intended. In addition, previous work [28] also identified overfitting as one of the major challenges for learning-based automatic program repair. We believe that one of the culprits for this is the poor objectives used in the training process, e.g., training a model to generate a particular target fix.

Let us consider the example in Figure 2 from the dataset of DeepFix [13]. This example program is incorrect due to the imbalanced number of curly brackets. In a traditional scenario, a compiler would warn the developer about this error. For example, when trying to compile this code with GCC, the compiler terminates with the error "expected declaration or statement at end of input", indicating line 10 as the error location. Experienced developers would be able to understand this cryptic message and proceed to fixing the program. Based on their intention, they can decide to add a curly bracket either at line 6 (patch P_1) or at line 9 (patch P_2). Both these solutions would fix the compilation error in the erroneous program, but the resulting solutions have different semantics.

Hence, we propose a deep generative framework to automatically correct programming errors by learning the distribution of potential fixes. We investigate different solutions to model the distribution of the fixes and sample multiple fixes, including different variants of Conditional Variation Autoencoders (CVAE) and beam search decoding. It turns out (as we will also show in our experiments) CVAE and beam search decoding are complementary, while CVAE is computationally more efficient in comparison to beam search decoding. Furthermore, we encourage diversity in the candidate fixes through a novel regularizer which penalizes similar fixes for an identical erroneous program and significantly increases the effectiveness of our approach. The candidate fixes in Figure 2 are generate by our approach, illustrating its potential for generating both diverse and correct fixes. For a given erroneous program, our approach is capable of generating diverse fixes to resolve the syntax errors.

To summarize, the contributions of this paper are as follows, 1. We propose an efficient generative method to automatically correct common programming errors by learning the distribution over potential fixes. 2. We propose a novel regularizer to encourage the model to generate diverse fixes. 3. Our generative model together with the diversity regularizer shows an increase in the diversity and accuracy of fixes, and a strong improvement over the state-of-the-art approaches.

2 Related Work

Our work builds on the general idea of sequence-to-sequence models as well as ideas from neural machine translation. We phrase our approach as a variational auto-encoder and compare it to prior learning-based program repair approaches. We review the related work in order below:

2.1 Neural Machine Translation

Sutskever et al. [30] introduces neural machine translation and casts it as a sequence-to-sequence learning problem. The popular encoder-decoder architecture is introduced to map the source sentences into target sentences. One of the major drawbacks of this model is that the sequence encoder needs to compress all of the extracted information into a fixed-length vector. Bahdanau et al. [3] addresses this issue by using attention mechanism in the encoder-decoder architecture, where it focuses on the most relevant part of encoded information by learning to search over the encoded vector. In our work, we employ a sequence-to-sequence model with attention to parameterize our generative model. This model gets an incorrect program as input and maps it to many potential fixes by drawing samples on the estimated distribution of the fixes.

2.2 Variational Autoencoders

The variational autoencoders [16,24] is a generative model designed to learn deep directed latent variable based graphical models of large datasets. The model is trained on the data distribution by maximizing the variational lower bound of the log-likelihood as the objective function. Bowman et al. [5] extend this framework by introducing an RNN-based variational autoencoder to enable the learning of latent variable based generative models on text data. The proposed model is successful in generating diverse and coherent sentences. To model conditional distributions for the structured output representation Sohn et al. [29] extended variational autoencoders by introducing an objective that maximizes the conditional data log-likelihood. In our approach, we employ an RNN-based conditional variational autoencoder to model the distribution of the potential fixes given erroneous programs. Variational autoencoder approaches enable the efficient sampling of accurate and diverse fixes.

2.3 Learning-based Program Repair

Recently there has been a growing interest in using learning-based approaches to automatically repair the programs [22]. Long and Rinard [20] proposed a probabilistic model by designing code features to rank potential fixes for a given program. Pu et al. [23] employ an encoder-decoder neural architecture to automatically correct programs. In these works and many learning-based programming repair approaches, enumerative search over programs is required to resolve all errors. However, our proposed framework is capable of predicting the location and potential fixes by passing the whole program to the model. Besides this, unlike our approach, which only generates fixes for the given erroneous program, Pu et al. [23] need to predict whole program statements to resolve the errors.

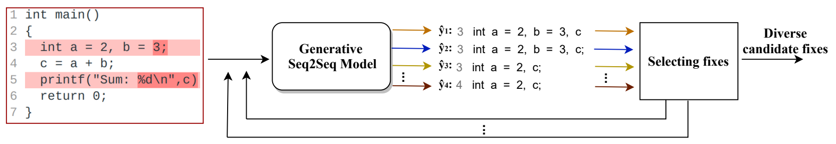
There are two important program repair tasks explored in the literature: fixing syntactic errors and fixing semantic ones. While in the current work we propose a technique for fixing syntactic errors, we believe that our observation about the diversity of the fix has implications for the approaches aimed at repairing semantic bugs as well. Most of the recent work in this domain aim to predict a unique fix, often extracted from a real-world repository. For example, Getafix [2], a recent approach for automatically repairing six types of semantic bugs, is evaluated on a set of 1,268 unique fixes written by developers. Similarly, DLfix [19] considers a bug to be fixed only if it exactly matches a patch provided by the developer. While this is an improved methodology in the spirit of our proposal it is highly dependent on the performance of the test suite oracle which may not always capture the developer's intent.

DeepFix [13], RLAssist [12], and DrRepair [32] uses neural representations to repair syntax errors in programs. In detail, DeepFix [13] uses a sequence-to-sequence model to directly predict a fix for incorrect programs. In contrast, our generative framework is able to generate multiple fixes by learning the distribution of potential correctness. Therefore, our model does not penalize, but rather encourages diverse fixes. RLAssist [12] repairs the programs by employing a reinforcement learning approach. They train an agent that navigate over the program to locate and resolve the syntax errors. In this work, they only address the typographic errors, rely on a hand-designed action space, and meet problems due to the increasing size of the action space. In contrast, our method shows improved performance on typographic errors and also generalizes to issues with missing variable declaration errors by generating diverse fixes.

In a recent work, Yasunaga and Liang [32] proposed DrRepair to resolve the syntax error by introducing a program feedback graph. They connect the relevant symbols in the source code and the compile error messages and employ the graph neural network on top to model the process of the program repair. In this work, they rely on the compiler error messages which can be helpful, but it also limits the generality of the method. However, our proposed approach does not rely on additional information such as compiler error messages, and it resolves the errors by directly modeling the underlying distribution of the potential correct fixes.

3 SampleFix: Generative Model for Diversified Code Fixes

Repairing the common program errors is a challenging task due to ambiguity in potential corrections and lack of representative data. Given a single erroneous program and a certain number of allowed changes, there are multiple ways to fix the program resulting in different styles and functionality. Without further information, the true, intended style and/or functionality remains unknown. In order to account for this inherent ambiguity, we propose a deep generative model to learn a distribution over potential fixes given the erroneous program – in contrast to predicting a single fix. We frame this challenging learning problem as a conditional variational autoencoders (CVAE). However, standard sampling procedures and limitations of datasets and their construction make learning and generation of diverse samples a challenge. We address this issue by a beam search decoding scheme in combination with a novel regularizer that encourages diversity of the samples in the embedding space of the CVAE.



```
1 int main()
2 {
3   int a = 2, b = 3;
4   c = a + b;
5   printf("Sum: %d\n",c)
6   return 0;
7 }
```

```
ŷ1: 3  int a = 2, b = 3, c
ŷ2: 3  int a = 2, b = 3, c;
ŷ3: 3  int a = 2, c;
ŷ4: 4  int a = 2, c;
```

Fig. 3: Overview of SampleFix at inference time, highlighting the generation of diverse fixes.

Figure 3 provides an overview of our proposed approach at inference time. For a given erroneous program, the generative model draws T intermediate, candidate fixes $\hat{y}$ from the learned conditional distribution. We use a compiler to select a subset of promising intermediate candidate fixes based on the number of remaining errors. This procedure is applied iteratively until arrive at a set of candidate fixes within the maximum number of prescribed changes. We then select a final set of candidate fixes that compile, have unique syntax according to our measure described below (Subsection 3.5).

In the following, we formulate our proposed generative model with the diversity regularizer and provide details of our training and inference process.

3.1 Conditional Variational Autoencoders for Generating Fixes

Conditional Variational Autoencoders (CVAE) [29], model conditional distributions $p_\theta(\mathbf{y}|\mathbf{x})$ using latent variables $\mathbf{z}$. The conditioning introduced through $\mathbf{z}$ enables the modelling of complex multi-modal distributions. As powerful transformations can be learned using neural networks, $\mathbf{z}$ itself can have a simple distribution which allows for efficient sampling. This model allows for sampling from $p_\theta(\mathbf{y}|\mathbf{x})$ given an input sequence $\mathbf{x}$, by first sampling latent variables $\hat{\mathbf{z}}$ from the prior distribution $p(\mathbf{z})$. During training, amortized variational inference is used and the latent variables $\mathbf{z}$ are learned using a recognition network $q_\phi(\mathbf{z}|\mathbf{x},\mathbf{y})$, parametrized by ϕ. In detail, the variational lower bound of the model (Equation 1) is maximized,

$$\log(p(\mathbf{y}|\mathbf{x})) \geq \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x},\mathbf{y})} \log(p_\theta(\mathbf{y}|\mathbf{z},\mathbf{x})) - D_{\mathrm{KL}}(q_\phi(\mathbf{z}|\mathbf{x},\mathbf{y}), p(\mathbf{z}|\mathbf{x})). \tag{1}$$

Penalizing the divergence of $q_\phi(\mathbf{z}|\mathbf{x},\mathbf{y})$ to the prior in Equation 1 allows for sampling from the prior $p(\mathbf{z})$ during inference. In practice, the variational lower bound is estimated using Monte-Carlo integration,

$$\hat{\mathcal{L}}_{\mathrm{CVAE}} = \frac{1}{T}\sum_{\mathrm{i}=1}^{T} \log(p_\theta(\mathbf{y}|\hat{\mathbf{z}}_{\mathrm{i}},\mathbf{x})) - D_{\mathrm{KL}}(q_\phi(\mathbf{z}|\mathbf{x},\mathbf{y}), p(\mathbf{z}|\mathbf{x})) \ . \tag{2}$$

where, $\hat{\mathbf{z}}_\mathrm{i} \sim q_\phi(\mathbf{z}|\mathbf{x},\mathbf{y})$, and T is the number of samples. We cast our model for resolving program errors in the Conditional Variational Autoencoder framework. Here, the input $\mathbf{x}$ is the erroneous program and $\mathbf{y}$ is the fix.

However, the plain CVAE as described in [29] suffers from diversity issues. Usually, the drawn samples do not reflect the true variance of the posterior $p(\mathbf{y}|\mathbf{x})$. This would amount to the correct fix potentially missing from our candidate fixes. To mitigate this problem, next we introduce an objective that aims to enhance the diversity of our candidate fixes.

3.2 Enabling Diverse Samples using a Best of Many Objective

Here, we introduce the diversity enhancing objective that we use. Casting our model in the Conditional Variational Autoencoder framework would enable us to sample a set of candidate fixes for a given erroneous program. However, the standard variational lower bound objective does not encourage diversity in the candidate fixes. This is because the average likelihood of the candidate fixes is considered. In detail, as the average likelihood is considered, all candidate fixes must explain the "true" fix in training set well. This discourages diversity and constrains the recognition network, which is already constrained to maintain a Gaussian latent variable distribution. In practice, the learned distribution fails to fully capture the variance of the true distribution. To encourage diversity, we employ "Many Samples" (MS) objective proposed by Bhattacharyya et al. [4],

$$\hat{\mathcal{L}}_\mathrm{MS} = \log\Big(\frac{1}{T}\sum_{\mathrm{i}=1}^{T} p_\theta(\mathbf{y}|\hat{\mathbf{z}}_\mathrm{i},\mathbf{x})\Big) - D_\mathrm{KL}(q_\phi(\mathbf{z}|\mathbf{x},\mathbf{y}), p(\mathbf{z}|\mathbf{x})) \ . \tag{3}$$

In comparison to Equation 2, this objective (Equation 3) encourages diversity in the model by allowing for multiple chances to draw highly likely candidate fixes. This enables the model to generate diverse candidate fixes, while maintaining high likelihood. In practice, due to numerical stability issues, we use "Best of Many Samples" (BMS) objective, which is an approximation of 3. This objective retains the diversity enhancing nature of Equation 3 while being easy to train,

$$\hat{\mathcal{L}}_\mathrm{BMS} = \max_\mathrm{i}\big(\log(p_\theta(\mathbf{y}|\hat{\mathbf{z}}_\mathrm{i},\mathbf{x}))\big) - D_\mathrm{KL}(q_\phi(\mathbf{z}|\mathbf{x},\mathbf{y}), p(\mathbf{z}|\mathbf{x})) \ . \tag{4}$$

3.3 DS-SampleFix: Encouraging Diversity with a Diversity-sensitive Regularizer

To increase the diversity using Equation 4 we need to use a substantial number of samples during training. This is computationally prohibitive especially for large models, as memory requirements and computation time increases linearly in the number of such samples. On the other hand, for a small number of samples, the objective behaves similarly to the standard CVAE objective as the recognition network has fewer and fewer chances to draw highly likely samples/candidate fixes, thus limiting diversity. Therefore, in order to encourage the model to generate diverse fixes even with a limited number of samples, we propose a novel regularizer that aims to increase the distance between the two closest candidate fixes (Equation 5). This penalizes generating similar candidate fixes for a given erroneous program and thus encourages diversity in the set of candidate fixes. In comparison to Equation 4, we observe considerable gains even with the use of only $T = 2$ candidate fixes. In detail, we maximize the following objective

$$\begin{aligned}\hat{\mathcal{L}}_{\text{DS-BMS}} = \max_{\mathrm{i}} \big(\log(p_\theta(\mathbf{y}|\hat{\mathbf{z}}_{\mathrm{i}}, \mathbf{x}))\big) + \min_{\mathrm{i,j}} d(\hat{\mathbf{y}}^{\mathrm{i}}, \hat{\mathbf{y}}^{\mathrm{j}}) \\ -D_{\mathrm{KL}}(q_\phi(\mathbf{z}|\mathbf{x}, \mathbf{y}), p(\mathbf{z}|\mathbf{x})) \ .\end{aligned} \tag{5}$$

Distance Metric. Here, we discuss the distance metric d in Equation 5. Note, that the samples $\left\{\hat{\mathbf{y}}^{\mathrm{i}}, \hat{\mathbf{y}}^{\mathrm{j}}\right\}$ can be of different lengths. Therefore, we first pad the shorter sample to equalize lengths. Empirically, we find that the Euclidean distance performs best. This is mainly because, in practice, Euclidean distance is easier to optimize.

3.4 Beam Search Decoding for Generating Fixes

Beam search decoding is a classical model to generate multiple outputs from a sequence-to-sequence model [31,7]. Given the distributions $p_\theta(\mathbf{y}|\mathbf{x})$ of a sequence-to-sequence model we can generate multiple outputs by unrolling the model in time and keeping the top-K tokens at each time step, where K is the beam width. In our generative model, we employ beam search algorithm to sample multiple fixes. In detail, we decode with beam width of size K for each sample $\mathbf{z}$ and in total for T samples from $p(\mathbf{z})$. We set $T = 100$ during inference.

3.5 Selecting Diverse Candidate Fixes

We extend the iterative repair procedure introduced by Gupta et al. [13] in the context of our proposed generative model, where the iterative procedure now leverages multiple candidate fixes. Given an erroneous program, the generative model outputs T candidate fixes. Each fix contains a potential erroneous line with the corresponding fix. So in each iteration we only edit one line of the given program. To select the best fixes, we take the candidate fixes and the input erroneous program, reconcile them to create T updated programs. We evaluate these fixes using a compiler, and select up to the best N fixes, where $N \leq T$. We only select the unique fixes which do not introduce any additional error messages. In the next iterations, we feed up to N programs back to the model. These programs are updated based on the selected fixes of the previous iteration. We keep up to N programs with the lower number of error messages over the iterations. At the end of the repairing procedure, we obtain multiple potential candidate fixes. In the experiments where we are interested in a single repaired program, we pick the best fix with the highest probability score according to our deep generative model.

3.6 Model Architecture and Implementation Details

To ensure a fair comparison, our generative model is based on the sequence-to-sequence architecture, similar to Gupta et al. [13]. Figure 4 shows the architecture of our approach in detail. Note that the recognition network is available to encode the fixes to latent variables $\mathbf{z}$ only during training. All of the networks in our framework consists of 4-layers of LSTM cells with 300 units. The network is optimized using Adam optimizer [15] with the default setting. We use $T = 2$ samples to train our models, and $T = 100$ samples during inference. To process the program through the networks, we tokenize the programs similar to the setting used by Gupta et al. [13].

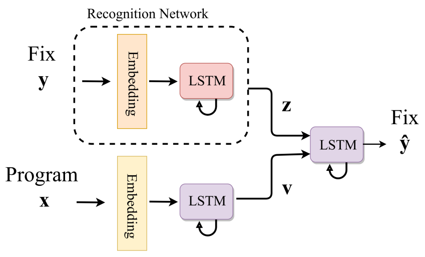


Fig. 4: Overview of network architecture.

During inference, the conditioning erroneous program $\mathbf{x}$ is input to the encoder, which encodes the program to the vector $\mathbf{v}$. To generate multiple fixes using our decoder, the code vector $\mathbf{v}$ along with a sample of $\mathbf{z}$ from the prior $p(\mathbf{z})$ is input to the decoder. For simplicity, we use a standard Gaussian $\mathcal{N}(0, \mathbf{I})$ prior, although more complex priors can be easily leveraged. The decoder is unrolled in time and output logits $(p_\theta(\mathbf{y}|\hat{\mathbf{z}}_\mathrm{i}, \mathbf{x}))$.

4 Experiments

We evaluate our approach on the task of repairing common programming errors. We evaluate the diversity and accuracy of our sampled error corrections as well as compare our proposed method with the state of the art.

4.1 Dataset

We use the dataset published by Gupta et al. [13] as it's sizable and includes real-world data. It contains C programs written by students in an introductory programming course. The dataset consists of 93 different tasks that were written by students in an introductory programming course. The programs were collected using a web-based system [6]. These programs have token lengths in the range $[75, 450]$, and contain typographic and missing variable declaration errors. To tokenize the programs and generate training and test data different type of tokens, such as types, keywords, special characters, functions, literals and variables are used. The dataset contains two sets of data which are called synthetic and real-world data. The synthetic data contains the erroneous programs which are synthesized by mutating correct programs written by students. The real-world data contains 6975 erroneous programs with 16766 error messages.

4.2 Evaluation

Table 2: Results of performance comparison of DeepFix, RLAssist, DrRepair, Beam search (BS), SampleFix , DS-SampleFix, and DS-SampleFix + BS. Typo, Miss Dec, and All refer to typographic, missing variable declarations, and all of the error messages respectively. Speed denotes computational time for sampling 100 fixes. ✔denotes successfully compiled programs, while 🐞 refers to resolved error messages.

Models	Typo		Miss Dec		All		Speed (s)
	✔	🐞	✔	🐞	✔	🐞	
DeepFix [13]	23.3%	30.8%	10.1%	12.9%	33.4%	40.8%	-
RLAssist [12]	*26.6%*	*39.7%*	-	-	-	-	-
DrRepair [32]	-	-	-	-	34.0%	-	-
Beam search (BS)	25.9%	42.2%	**20.3%**	47.0%	44.7%	63.9%	4.82
SampleFix	24.8%	38.8%	16.1%	22.8%	40.9%	56.3%	0.88
DS-SampleFix	27.7%	40.9%	16.7%	24.7%	44.4%	61.0%	0.88
DS-SampleFix + BS	**27.8%**	**45.6%**	19.2%	**47.9%**	**45.2%**	**65.2%**	1.17

We evaluate our approach on synthetic and real-world data. To evaluate our approach on the synthetic test set we randomly select 20k pairs. This data contains pairs of erroneous programs with the intended fixes. To evaluate our approach on real-world data we use a real-world set of erroneous programs. Unlike synthetic test set, we don't have access to the intended fix(es) in the real-world data. However, we can check the correctness of the program using the evaluator (compiler). Following the prior work, we train two networks, one for typographic errors and another to fix missing variables declaration errors. Note that there might be an overlap between the error resolved by the network for typographic errors and the network for missing variables declaration errors, so we also provide the overall results of the resolved error messages.

Table 1: Results of performance comparison of DeepFix, Beam search (BS), SampleFix ,and DS-SampleFix on synthetic data. Typo, Miss Dec, and All refer to typographic, missing variable declarations, and all of the errors respectively.

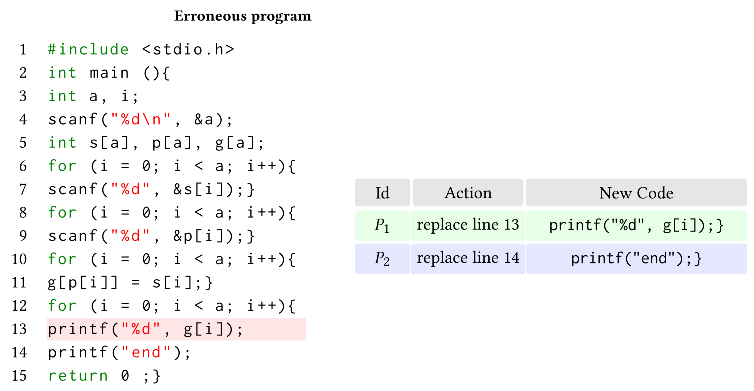
Models	Typo	Miss Dec	All
DeepFix	84.7%	78.8%	82.0%
Beam search (BS)	91.8%	89.5%	90.7%
SampleFix	86.8%	86.5%	86.6%
DS-SampleFix	95.6%	88.1%	92.2%

Synthetic Data. Table 1 shows the comparison of our proposed approaches, Beam search (BS), SampleFix and DS-SampleFix, with DeepFix [13] on the synthetic data in the first iteration. In this table (Table 1), we can see that our approaches outperform DeepFix in generating intended fixes for the typographic and missing variable declaration errors. Beam search (BS), SampleFix and DS-SampleFix generate 90.7%, 86.6%, and 92.2% of the intended fixes respectively.

Real-World Data. In Table 2 we compare our approaches, with state-of-the-art approaches (DeepFix [13], RLAssist [12], and DrRepair [32]) on the real-world data. In our experiments (Table 2) we show the performance of beam search decoding, CVAEs (SampleFix), and our proposed diversity-sensitive regularizer (DS-SampleFix). Furthermore, we show that DS-SampleFix can still take advantage of beam search algorithm (DS-SampleFix + BS). To do that, for each sample z we decode with beam width of size 5, and to sample 100 fixes we draw 20 samples from $p(z)$. We also provide the sampling speed in terms of sampling 100 fixes for a given program using an average over 100 runs. The running time results show that CVAE-based models are at least 4x faster than beam search in sampling the fixes. In this experiment, we feed the programs up to 5 iterations.

Table 2 shows that our approaches outperform DeepFix [13], RLAssist [12], and DrRepair [32] in resolving the error messages. This shows that generating multiple diverse fixes can lead to substantial improvement in performance. Beam search, SampleFix, DS-SampleFix, and DS-SampleFix + BS resolve 63.9%, 56.3%, 61.0%, and 65.2% of the error messages respectively. Overall, our DS-SampleFix + BS is able to resolve all compile-time errors of the 45.2% of the programs - around 12% points improvement over DeepFix and 11% points improvement over DrRepair. Furthermore, the performance advantage of DS-SampleFix over SampleFix shows the effectiveness of our novel regularizer.

Note that DrRepair [32] has achieved further improvements by relying on the compiler. While utilizing the compiler output seems to be beneficial, it also limits the generality of the approach. For a fair comparison, we report the performance of DrRepair without the compiler output, but consider informing our model by the compiler output an interesting avenue of future work.


Erroneous program

```
1  #include <stdio.h>
2  int main (){
3  int a, i;
4  scanf("%d\n", &a);
5  int s[a], p[a], g[a];
6  for (i = 0; i < a; i++){
7  scanf("%d", &s[i]);}
8  for (i = 0; i < a; i++){
9  scanf("%d", &p[i]);}
10 for (i = 0; i < a; i++){
11 g[p[i]] = s[i];}
12 for (i = 0; i < a; i++){
13 printf("%d", g[i]);
14 printf("end");
15 return 0 ;}
```

Id	Action	New Code
P_1	replace line 13	`printf("%d", g[i]);}`
P_2	replace line 14	`printf("end");}`

Fig. 5: An example illustrating that our DS-SampleFix can generate diverse fixes. Left: Example of a program with a typographic error. The error, i.e., missing bracket, is highlighted at line 13. Right: Our DS-SampleFix proposes multiple fixes for the given error (line number with the corresponding fix), highlighting the ability of DS-SampleFix to generate diverse and accurate fixes.

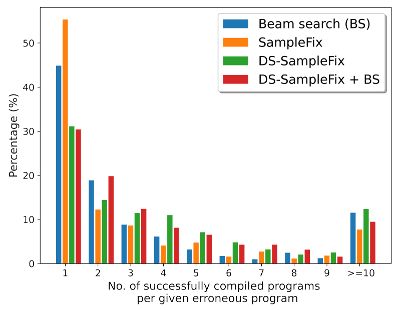
Qualitative Example. We illustrate diverse fixes generated by our DS-SampleFix in Figure 5 using a code example with typographic errors, with the corresponding two output samples of DS-SampleFix. In the examples given in Figure 5, there is a missing closing curly bracket after line 13. We can see that DS-SampleFix generates multiple correct fixes to resolve the error in the given program. This indicates that our approach is capable of handling the inherent ambiguity and uncertainty in predicting fixes for the erroneous programs. The two fixes in Figure 5 are unique and compileable fixes that implement different functionalities for the given erroneous program. Note that generating multiple diverse fixes gives the programmers the opportunity of choosing the desired fix(es) among the compileable ones, based on their intention.

Generating Functionally Diverse Programs. Given an erroneous program, our approach can generate multiple potential fixes that result in a successful compilation. Since we do not have access to the user's intention, it is desirable to suggest multiple potential fixes with diverse functionalities. Here, we evaluate our approach in generating multiple programs with different functionalities.

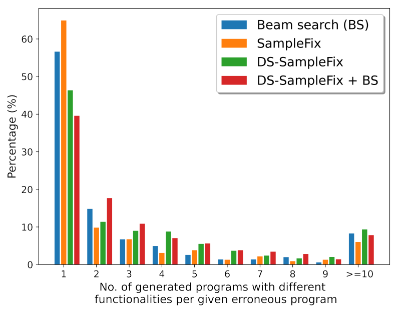
In order to assess different functionalities, we use the following approach based on tests. The dataset of Gupta et al. [13] consists of 93 different tasks. The description of each task, including the input/output format, is provided in the dataset. Based on the input/output format, we can provide input examples for each task. To measure the diversity in functionality of the programs in each task, we generate 10 input examples. For instance, given a group of programs for a specific task, we can run each program using the input examples and get the outputs. We consider two programs to have different functionalities if they return different outputs given the same input example(s).

In order to generate multiple programs we use our iterative selecting strategy (Subsection 3.5). In each iteration, we keep up to N programs with the less number of error messages over the iterations. At the end of the repairing procedure, we obtain multiple repaired programs. As discussed (Figure 1), a subset of these programs will successfully compile. In this experiment, we use the real-world test set, and we set $N = 50$ as this number is large enough to allow us to study the diversity of the fixes, without incurring an unnecessarily large load on our infrastructure. Our goals in the remaining of this section are: 1. For each erroneous program, to measure the number of generated unique fixes that successfully compile. 2. For each erroneous program, to measure the number of generated programs with different functionalities.

Figure 6a and Figure 6b show the syntactic diversity of the generated programs, and the diversity in functionality of these programs, respectively. In Figure 6a we show the percentage of the successfully compiled programs with unique fixes for a given erroneous program. The x-axis refers to the number of generated and successfully compiled unique programs, and y-axis to the percentage of repaired programs for which these many unique fixes were generated. For example, for almost 20% of the repaired programs, DS-SampleFix + BS generates two



(a) Diversity of the generated programs.



(b) Diversity of the functionality of the generated programs.

Fig. 6: The results show the performance of Beam search (BS), SampleFix , DS-SampleFix , and DS-SampleFix + BS. (a) Percentage of the number of the generated successfully compiled, unique programs for the given erroneous programs. (b) Percentage of the successfully compiled programs with different functionalities for the given erroneous programs.

unique fixes. Overall, we can see that DS-SampleFix and DS-SampleFix + BS generate more diverse programs in comparison to the other approaches.

Figure 6b shows the percentage of the successfully compiled programs with different functionalities, for a given erroneous program. Here, the x-axis refers to the number of the generated functionally different programs, and y-axis refers to the percentage of erroneous programs with at least one fix, for which we could generate that many diverse fixes. One can observe that in many cases, e.g., up to 60% of the times for SampleFix, the methods generate programs corresponding to a single functionality. However, in many other cases they generate functionally diverse fixes. For example, in almost 10% of the cases, DS-SampleFix generate 10 or more fixes with different functionalities. In Figure 6b we can see that all of the approaches have higher percentage for generating program with the same functionality in comparison to the results in Figure 6a. This indicates that for some of the given erroneous programs, we generate multiple unique programs with approximately the same functionality. These results show that

Table 3: Results of performance comparison of Beam Search (BS), SampleFix , DS-SampleFix , and DS-SampleFix +BS on generating diverse programs. Diverse Prog refers to the percentage of cases where the models generate at least two or more successfully compiled unique programs. Diverse Func denotes the percentage of cases where the models generate at least two or more programs with different functionalities.

Models	Diverse Prog	Diverse Func
Beam search	55.6%	45.1%
SampleFix	44.6%	34.9%
DS-SampleFix	68.8%	53.4%
DS-SampleFix + BS	**69.5%**	**60.4%**

DS-SampleFix and DS-SampleFix + BS generate programs with more diverse functionalities in comparison to the other approaches.

In Table 3 we compare the performance of our approaches in generating diverse programs and functionalities. We provide results for all of our four approaches, i.e., Beam search (BS), SampleFix, DS-SampleFix, and DS-SampleFix + BS. We consider that an approach can generate diverse programs if it can produce two or more successfully compiled, unique programs for a given erroneous program. Similarly, we say that the approach produces functionally diverse programs if it can generate two or more programs with observable differences in functionality for a given erroneous program. Here we consider the percentage out of the total number of erroneous programs for which the model generates at least one successfully compiled program. The results of this table show that our DS-SampleFix + BS approach generates programs with more diverse functionalities in comparison to the other approaches.

5 Conclusion

We propose a novel approach to correct common programming errors. We recognize and model the inherent ambiguity and uncertainty when predicting multiple fixes. In contrast to previous approaches, our approach is able to learn the distribution over candidate fixes rather than the most likely fix. We achieve increased diversity of the sampled fixes by a novel diversity-sensitive regularizer. We show that our approach is capable of generating multiple diverse fixes with different functionalities. Furthermore, our evaluations on synthetic and real-world data show improvements over state-of-the-art methods.

References

1. Allamanis, M., Barr, E.T., Devanbu, P., Sutton, C.: A survey of machine learning for big code and naturalness. ACM Computing Surveys (CSUR) (2018)
2. Bader, J., Scott, A., Pradel, M., Chandra, S.: Getafix: learning to fix bugs automatically. Proc. ACM Program. Lang. **3**(OOPSLA) (2019)
3. Bahdanau, D., Cho, K., Bengio, Y.: Neural machine translation by jointly learning to align and translate (2015)
4. Bhattacharyya, A., Schiele, B., Fritz, M.: Accurate and diverse sampling of sequences based on a "best of many" sample objective. In: CVPR (2018)
5. Bowman, S.R., Vilnis, L., Vinyals, O., Dai, A.M., Jozefowicz, R., Bengio, S.: Generating sentences from a continuous space. In: SIGNLL Conference on Computational Natural Language Learning (CoNLL) (2016)
6. Das, R., Ahmed, U.Z., Karkare, A., Gulwani, S.: Prutor: A system for tutoring CS1 and collecting student programs for analysis (2016)
7. Deshpande, A., Aneja, J., Wang, L., Schwing, A.G., Forsyth, D.: Fast, diverse and accurate image captioning guided by part-of-speech. In: CVPR (2019)
8. D'Antoni, L., Samanta, R., Singh, R.: Qlose: Program repair with quantitative objectives. In: CAV (2016)
9. Girshick, R.: Fast r-cnn. In: ICCV (2015)

10. Gottschlich, J., Solar-Lezama, A., Tatbul, N., Carbin, M., Rinard, M., Barzilay, R., Amarasinghe, S., Tenenbaum, J.B., Mattson, T.: The three pillars of machine programming. In: MAPL (2018)
11. Goues, C.L., Pradel, M., Roychoudhury, A.: Automated program repair. Commun. ACM **62**(12), 56–65 (2019)
12. Gupta, R., Kanade, A., Shevade, S.: Deep reinforcement learning for programming language correction. In: AAAI (2019)
13. Gupta, R.R., Pal, S., Kanade, A., Shevade, S.K.: Deepfix: Fixing common c language errors by deep learning. In: AAAI (2017)
14. Jang, E., Gu, S., Poole, B.: Categorical reparameterization with gumbel-softmax. In: ICLR (2017)
15. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. In: ICLR (2015)
16. Kingma, D.P., Welling, M.: Auto-encoding variational bayes. In: ICLR (2014)
17. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: NIPS (2012)
18. Lee, H., Grosse, R., Ranganath, R., Ng, A.Y.: Unsupervised learning of hierarchical representations with convolutional deep belief networks. Communications of the ACM (2011)
19. Li, Y., Wang, S., Nguyen, T.N.: Dlfix: Context-based code transformation learning for automated program repair. In: International Conference on Software Engineering (ICSE) (2020)
20. Long, F., Rinard, M.: Automatic patch generation by learning correct code. In: ACM SIGPLAN Notices (2016)
21. Maddison, C.J., Mnih, A., Teh, Y.W.: The concrete distribution: A continuous relaxation of discrete random variables (2016)
22. Monperrus, M.: Automatic software repair: a bibliography. ACM Computing Surveys (CSUR) (2018)
23. Pu, Y., Narasimhan, K., Solar-Lezama, A., Barzilay, R.: sk_p: a neural program corrector for moocs. In: ACM SIGPLAN (2016)
24. Rezende, D.J., Mohamed, S.: Variational inference with normalizing flows. In: ICML (2015)
25. Seo, H., Sadowski, C., Elbaum, S., Aftandilian, E., Bowdidge, R.: Programmers' build errors: a case study (at google). In: ICSE (2014)
26. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. In: ICLR (2015)
27. Singh, R., Gulwani, S., Solar-Lezama, A.: Automated feedback generation for introductory programming assignments. In: PLDI (2013)
28. Smith, E.K., Barr, E.T., Goues, C.L., Brun, Y.: Is the cure worse than the disease? overfitting in automated program repair. In: Foundations of Software Engineering (ESEC/FSE) (2015)
29. Sohn, K., Lee, H., Yan, X.: Learning structured output representation using deep conditional generative models. In: NIPS (2015)
30. Sutskever, I., Vinyals, O., Le, Q.V.: Sequence to sequence learning with neural networks. In: NIPS (2014)
31. Wang, L., Schwing, A., Lazebnik, S.: Diverse and accurate image description using a variational auto-encoder with an additive gaussian encoding space. In: NIPS (2017)
32. Yasunaga, M., Liang, P.: Graph-based, self-supervised program repair from diagnostic feedback. In: ICML (2020)