Fix Bugs with Transformer through a Neural-Symbolic Edit Grammar

Anonymous ACL submission

Abstract

We introduce NSEdit (neural-symbolic edit), a novel Transformer-based code repair method. Given only the source code that contains bugs, NSEdit predicts an editing sequence that can fix the bugs. The edit grammar is formulated as a regular language, and the Transformer uses it as a neural-symbolic scripting interface to generate editing programs. We modify the Transformer and add a pointer network to select the edit locations. An ensemble of rerankers are trained to re-rank the editing sequences generated by beam search. We fine-tune the rerankers on the validation set to reduce overfitting. NSEdit is evaluated on various code repair datasets and achieved a new state-of-the-art accuracy (24.04%) on the Tufano small dataset of the CodeXGLUE benchmark. NSEdit performs robustly when programs vary from packages to packages and when buggy programs are concrete. We conduct detailed analysis on our methods and demonstrate the effectiveness of each component.

1 Introduction

Neural networks pretrained on source code (Feng et al., 2020; Kanade et al., 2020; Guo et al., 2020; Ahmad et al., 2021) are bringing substantial gains on many code understanding tasks such as code classification, code search, and code completion (Lu et al., 2021). However, code repair remains challenging because it requires the model to have robust syntactic and semantic understanding of a given program even when it contains bugs. The difference between a buggy program and its fixed version often lies in small details that must be fixed exactly, which further requires the model to have precise knowledge about programming.

Code repair with large language models often formulate the problem as a Neural Machine Translation problem (NMT), where the input buggy code is "translated" into its fixed version as the output. We categorize existing work based on two design factors: the *translation target* and the *buggy code representation*. For the translation target, the model can predict the fixed code directly (Tufano et al., 2019; Phan et al., 2021; Yasunaga and Liang, 2021) or generate some form of edit that can be applied to the buggy code to fix it (Yao et al., 2021; Chen et al., 2019; Zhu et al., 2021; Yin et al., 2018). For the buggy code representation, we can use a tokenized sequence of buggy code for sequence-to-sequence (Seq2Seq) prediction (Chen et al., 2019; Bhatia et al., 2018) or program analysis representations such as abstract syntax tree (AST), data-flow graph (DFG), and error messages (Yao et al., 2021; Allamanis et al., 2021; Berabi et al., 2021).

We propose NSEdit, a Transformer-based (Vaswani et al., 2017) model that predicts the editing sequence given only the source code. We use both encoder and decoder of the Transformer: the encoder processes the buggy code, and the decoder predicts the editing sequence given an edit grammar. We design the edit grammar as a regular language, and the Transformer uses it as a domain-specific language (DSL) to write scripts that can fix the bugs when executed. The grammar consists of two actions, *delete* and *insert*, which are added to the vocabulary of the language model as new tokens. The decoder has two modes: *word/action mode* predicts the two action tokens along with word tokens, and *location mode* selects the location of the edit. A pointer network implements the location selection mode, and we slice the encoder memory as the embedding of the edit location to enable content-based retrieval, instead of representing a location as a static word embedding in the vocabulary. We use beam search to generate predictions at inference time. Given diversity-versus-quality problem encountered in all beam-search-based sequence generation methods (Kool et al., 2019), we train rerankers to improve the quality of sequences generated (Ng et al., 2019; Lee et al., 2021; Ramesh et al., 2021). An overview of the architecture is provided in Figure 1. We will publish our source code.

We now introduce our decisions and hypotheses with respect to the two design factors of translation target and buggy code representations in the context of existing literature.

Translation target Using the fixed code as translation target is straight-forward to implement, with the added benefit that input and target are both code sequences, which can be easier to model and, compared to edits, more similar to the code corpus that large language models are pre-trained on. However, as our results show, predicting the fixed code directly may encourage the model to learn the copying behavior which causes overfitting and does not reflect the goal of editing the code to make changes. Existing edit prediction approaches often rely on a graph representation of the buggy code (Yao et al., 2021). This implicitly assumes that there exists a graph representation for the buggy code in the first place, which may not be true if the bug causes syntactical errors. Existing edit prediction approaches may also require significant architecture design and involve multiple stages of edit prediction (Zhu et al., 2021; Hashimoto et al., 2018; Yin et al., 2018). Instead, we design an edit grammar and rely on Transformer to do what it does best as a language model: learn the edit grammar and output editing instructions with this grammar. As to the architectural changes, we add a pointer network to predict the location of an edit, which is an essential modification. Our use of rerankers is an orthogonal change that adds to the performance significantly but not the complexity of the main Transformer model. The edit grammar can be seen as a DSL for edits, given which the Transformer generates a short program to edit the input sequence. This places our work alongside the burgeoning neural-symbolic literature to use neural networks to write executable scripts (Chen et al., 2021; Mao et al., 2019), vastly expanding the algorithmic capacity of deep learning systems.

Buggy code representation We decide to train the model end-to-end given only the buggy code and fixed code without any auxiliary program analysis information. In light of recent finding that Transformers are universal sequence-to-sequence function approximators (Yun et al., 2020), we want to prove that Transformers are powerful enough to learn the syntax and data flow through pre-training on large code corpus and can do so robustly even when bugs are present in the input. When syntactical structures are not given, the Transformer model has to learn the syntax, which may provide additional signal that helps the model learn the semantics of the code corpus (Manning et al., 2020). When human programmers debug, they look at the code directly and only use AST/DFG as a mental model, and, as our results show, Transformer can learn to edit the code directly as well. Representing buggy code with program analysis graphs can incorporate important static and dynamic analysis information. However, for code repair in particular, bugs may cause syntactical errors that prevent extraction of program analysis graphs. Program analysis tools may impose restrictions on the inputted programs (e.g. language-dependent tools), while our problem formulation is more general and portable. Lastly, it is a known challenge that neural program models encounter generalizability issues when semantic-preserving program transformations are encountered (Rabin et al., 2021).

The main contributions of our paper are: (1) Our proposed method NSEdit achieves the state-of-the-art performance on the CodeXGLUE code repair benchmark (Lu et al., 2021). We show that pre-trained Transformer, given only the buggy code without program analysis representation or auxiliary information, can reach SOTA performance in code repair formulated as a sequence-to-sequence neural machine translation problem. (2) We formulate NSEdit grammar that is a regular language and one of the simplest edit representation in the literature. The two-mode decoder and finite state machine together ensure that the model follows the grammar. The Transformer uses the grammar as a neural-symbolic API to generate executable scripts that edit the code. We show that predicting editing sequences leads to superior performance, even when programs vary from packages to packages. (3) We use a pointer network to achieve content-based edit location selection. We slice the encoder memory to obtain the latent representation of a potential edit location. (4) We use an ensemble of rerankers to re-order the top-K editing sequences produced by beam search and significantly improve all exact match accuracy. We apply a novel technique to fine-tune the rerankers on validation set which effectively reduces over-fitting of rerankers. (5) We show that the reranking score can be used to improve the precision of editing sequences with efficient trade-off of recall.

2 Related work

Code repair with deep learning In addition to the code repair methods we discussed in the Introduction, we see a diverse array of methods proposed in recent years. Getafix (Johannes et al., 2019) presents a novel hierarchical clustering algorithm that summarizes fix patterns into a hierarchy and uses a simple ranking technique based on the context of a code change to select the most appropriate fix for a given bug. Vasic et al. (2019a) presents multi-headed pointer networks to localize and fix the variable misuse bugs. Recently, Dinella et al. (2020) learns a sequence of graph transformations to detect and fix a broad range of bugs in Javascript: given a buggy program modeled by a graph structure, the model makes a sequence of predictions including the position of bug nodes and corresponding graph edits to produce a fix. DeepDebug (Drain et al., 2021) trains a backtranslation Transformer model and uses various program analysis information obtained from test suites to fine-tune the model.

Neural machine translation with Transformer Transformer and its derivative models such as BERT (Devlin et al., 2018) and GPT (Radford et al., 2018) form a family of large language models that dominate neural machine translation and deep natural language processing. The models consists of many parameters, and the performance of the model scales with the size of the model (Brown et al., 2020). Recently, Transformer has been adapted to domains other than natural language processing, such as image classification (Dosovitskiy et al., 2020) and protein structure modeling (Rao et al., 2021), demonstrating Transformer as a general-purpose architecture.

3 Methods

Problem setup Our code repair dataset contains pairs (x, y) of strings, where x is the source code that contains bugs (buggy code), and y is the fixed code. Given the buggy code as the input sequence x, the goal of NSEdit is to generate the correct sequence of edits e that can transform x into y.

For notations, we use variables such as x to denote strings or constants. We use bold variables such as $\mathbf{x}$ to denote tokens and tensors. We use hatted variables such as $\hat{\mathbf{x}}$ or $\hat{x}$ to denote model predictions. We use uppercase bold variables such as $\mathbf{X}$ to denote a probability distribution.

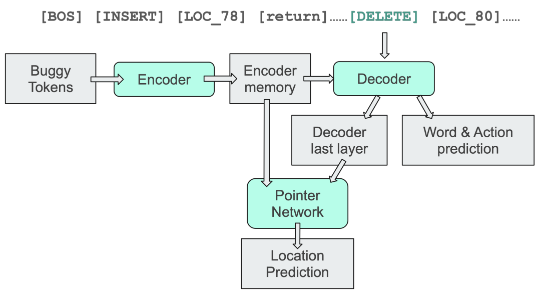

Figure 1: An illustration of the main NSEdit model architecture. There are two modes in the decoder. One mode predicts words and actions, and the other mode selects locations with a pointer network. The pointer network takes the penultimate layer output of the decoder and compares it with the encoder memory by dot product in order to select edit location.

Overview of NSEdit Training Training NSEdit consists of a pipeline of stages. We tokenize the buggy-fixed pair (x, y) with pre-trained CodeBERT tokenizer before obtaining ground truth edits e. We load the pre-trained CodeBERT encoder (Feng et al., 2020) and CodeGPT decoder (Lu et al., 2021). We fine-tune the NSEdit model f to predict editing sequences with teacher forcing. After training the model, we use beam search to generate the top-5 editing sequences (hypotheses). We train two rerankers with different architectures to classify which editing sequence is correct among the beam search hypotheses. The two rerankers and the original beam search score are combined with an ensemble model to produce the final reranking. Lastly, we fine-tune the rerankers on the beam search hypotheses on the validation set to reduce over-fitting. The beam search hypotheses reranked by the fine-tuned ensemble are the final predictions.

3.1 Tokenization and editing sequence generation

We tokenize the both buggy code and fixed code with a Byte Pair Encoding (BPE) (Sennrich et al., 2015) tokenizer that is pre-trained on CodeBERT code corpus. To compute the editing sequences, we use a variation of Ratcliff-Obershelp algorithm (Ratcliff and Metzener, 1988) implemented in Python's difflib library. The editing sequences are computed on tokenized sequences instead of raw strings.

To formulate the NSEdit grammar formally, an editing sequence consists of two types of actions: *delete*(i, j) and *insert*$(i, \mathbf{s})$. The *delete*(i, j) action deletes the subsequence in $[i, j)$ from the buggy sequence $\mathbf{x}$. The *insert*$(i, \mathbf{s})$ action inserts a sequence of tokens $\mathbf{s}$ before location i in the buggy sequence $\mathbf{x}$. As a result, NSEdit grammar contains three types of tokens in an editing sequence: action, word and location tokens. The finite state machine that describes NSEdit grammar is provided in Figure 2. An example editing sequence can be `[DELETE] [LOC_1] [LOC_2] [INSERT] [LOC_2] @Override public`. More example editing sequences are provided in Appendix E.

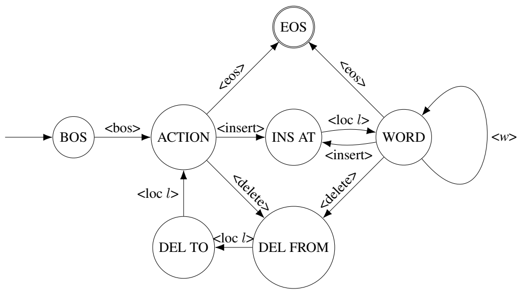

Figure 2: The transition diagram of the finite state machine for the NSEdit grammar used to generate editing sequences. The start state is the state BOS. The accept state is the state EOS.

3.2 Training Transformer to predict bug fix editing sequences with teacher forcing

We use the Transformer model (Vaswani et al., 2017) to perform sequence-to-sequence prediction. The NSEdit model computes $f(\mathbf{x}) = \hat{\mathbf{e}}$, where $\mathbf{x}$ is the buggy token sequence and $\hat{\mathbf{e}}$ is the predicted editing sequence. The encoder processes buggy code $\mathbf{x}$ and outputs the encoder memory $\mathbf{m}$, formally shown in Equation 1. For input $\mathbf{x}$ with L tokens and model with h hidden units, the encoder memory has shape (L, h), omitting the batch dimension. The decoder takes $\mathbf{m}$ and the current editing sequence token $\mathbf{e}_i$ as the input and autoregressively predicts the next token $\hat{\mathbf{e}}_{i+1}$ by maximum likelihood, as shown in Equation 2 and 3, where $[\cdot]$ denotes the slicing operator.

$$\mathbf{m} = \mathit{encoder}(\mathbf{x}) \tag{1}$$

$$\hat{\mathbf{E}}_{i+1} = \mathit{decoder}(\mathbf{m}, \mathbf{e}_i) \tag{2}$$

$$\hat{\mathbf{e}}_{i+1} = \arg\max_{w \in W} \hat{\mathbf{E}}_{i+1}[w] \tag{3}$$

We use teacher forcing as the training procedure (Williams and Zipser, 1989; Lamb et al., 2016). This means that in Equation 2, the ground truth edit token $\mathbf{e}_i$ is inputted into the decoder, but not the predicted token $\hat{\mathbf{e}}_i$. For Seq2Seq models, teacher forcing decouples prediction $\hat{\mathbf{e}}_i$ from $\hat{\mathbf{e}}_{i+1}$ during back propagation, thus it is more robust against vanish/exploding gradient problems common in recurrent neural networks.

We fine-tune pre-trained CodeBERT (Feng et al., 2020) and CodeGPT (Lu et al., 2021). We modify the decoder to have two modes, a *word/action mode* that predicts edit actions and inserted words, and a *location mode* that predicts edit locations.

The original CodeBERT tokenizer has 50265 word tokens in the vocabulary, and we add <*delete*> and <*insert*> tokens to the vocabulary. When predicting words or actions, the decoder outputs a probability vector $\hat{\mathbf{w}}$ over a set W of 50267 elements by passing the logits output $\mathbf{c}$ into the softmax function, shown in Equation 4.

$$\hat{\mathbf{W}} = \frac{\mathit{exp}(\mathbf{c})}{\sum_{j \in W} \mathit{exp}(\mathbf{c}_i)} \tag{4}$$

When predicting locations, instead of further expanding the vocabulary to add 513 location tokens and predict them along with words and actions, the decoder uses a pointer network in place of the last layer of the decoder (Figure 1).

The pointer network is a feed forward neural network. It transforms the output from the penultimate layer of the decoder into a latent representation $\mathbf{v}$ (Vinyals et al., 2015; Vasic et al., 2019b). In order to determine the location of the edit, we compute the dot product between $\mathbf{v}$ and $\mathbf{m}$ before a softmax function over all edit locations, as shown in equation 5. As the result, the pointer network outputs a probability vector $\hat{\mathbf{L}}$ over all edit locations at index $0, 1, 2 \ldots L$ for a buggy code with L tokens.

$$\hat{\mathbf{L}} = \frac{\mathit{exp}(\mathbf{v}^T\mathbf{m})}{\sum_{j=0}^{L} \mathit{exp}(\mathbf{v}^T\mathbf{m}_j)} \tag{5}$$

Since ground truth is available with teacher forcing, we determine which decoder mode to use given the type of ground truth token $\mathbf{e}_{i+1}$.

We slice the encoder memory as the embedding $\mathbf{m}[l]$ to replace the embedding of a location token <*loc l*> as the input to the decoder in Equation 2. As the result, the input $\mathbf{m}[l]$ and output $\mathbf{v}$ of the decoder for locations are both content-based representations, rather than a fixed location embedding that does not change when location context changes with the input program. We use cross entropy loss for both word/action prediction and location prediction and add them together with equal coefficients.

3.3 Generating beam search hypotheses during inference

During inference, when ground truth is not available, we generate sequence predictions with beam search (Reddy et al., 1977; Graves, 2012; Sutskever et al., 2014). Every partially generated sequence is assigned a probability Π that is the product of every token probability, and the Top-K most probable editing sequences (hypotheses) are outputted, where $K = 5$. Formal definitions of our beam search procedure is provided in Appendix A.

The finite state machine can uniquely determine the next token type based on the previous token given the transition function in Figure 2. Formally, we modify Equation 2 to use *fsm* to determine the token type as an input to the decoder

$$\hat{\mathbf{E}}_{i+1} = \mathit{decoder}\big(\mathbf{m}, \hat{\mathbf{e}}_i, \mathit{fsm}(\hat{e}_i)\big) \tag{6}$$

We mask the probability of an invalid token to be zero, thereby ensuring valid NSEdit grammar syntax. We slice the encoder memory based on the predicted edit location $\hat{\mathbf{e}}_i$ during inference. Formally, when the current input is an edit location, $\hat{\mathbf{e}}_i = \arg\max_{j=0,1,2..L} \hat{\mathbf{E}}_i[j] = \text{<}\mathit{loc\ l}\text{>} = \mathbf{m}[l]$ in Equation 6.

3.4 Reranking the beam search hypotheses

Our results show that top-5 accuracy is significantly higher than top-1 accuracy (Table 6), meaning that the correct edit can be produced among the 5 beam search hypotheses but not ranked the most probable by the original beam search probability Π in Equation 11. Beam search with models trained with teacher-forcing can produce a diverse set of hypotheses, and the quality of the hypotheses may be improved (Kool et al., 2019). To do so, we rerank the beam search hypotheses with rerankers (Ng et al., 2019; Lee et al., 2021). We formulate this problem as a classification problem: given K hypotheses produced by the beam search that have a correct prediction, the objective of the reranker is to classify which of the K hypotheses is the correct one. The reranking score for a hypothesis $\hat{\mathbf{e}}_k$ is computed as

$$\mathit{score}(\hat{\mathbf{e}}_k) = \log \frac{\exp(\mathit{reranker}(\hat{\mathbf{e}}_k)/T)}{\sum_{i=1,2,..K} \exp(\mathit{reranker}(\hat{\mathbf{e}}_i)/T)} \tag{7}$$

where $T = 0.5$ is a temperature term that controls the smoothness of the probability distribution (Lee et al., 2021). Note that each hypothesis $\hat{\mathbf{e}}_k$ goes through the same *reranker* function where each $\mathit{reranker}(\hat{\mathbf{e}}_k)$ has a scalar output. This model can be seen as a special case of Siamese model (Koch et al., 2015). We use cross entropy loss to train rerankers on the beam search hypotheses that are produced on the training set by the main NSEdit model.

We train two rerankers with different architectures: one with both Transformer encoder and decoder, the same architecture as the main NSEdit model, and the other with encoder only. The Transformer reranker outputs reranking score at the end of the sequence, and the encoder-only reranker outputs at the <*cls*> token. Formally, for a beam hypotheses $\hat{\mathbf{e}}$ of length L, we compute the reranking scores

$$\mathit{reranker}_{\mathit{Transformer}}(\hat{\mathbf{e}}) = \mathit{ff}_1(\hat{\mathbf{E}}_{L-1}) \tag{8}$$

$$\mathit{reranker}_{\mathit{Encoder}}(\hat{\mathbf{e}}) = \mathit{ff}_2(\mathbf{m}[\text{<}\mathit{cls}\text{>}]) \tag{9}$$

where $\hat{\mathbf{E}}_{L-1}$ is computed given by Equation 6 and $\mathbf{m}$ by Equation 1. The *ff* function is a simple one-layer feed-forward neural network.

For every beam search hypotheses $\hat{\mathbf{e}}$, we have three ranking scores: the original beam search log probability score Π (Equation 11) and two reranking scores. We use a linear ensemble model to blend the ranking scores:

$$s = \log \Pi + c_1 \mathit{score}_{\mathit{Transformer}} + c_2 \mathit{score}_{\mathit{Encoder}} \tag{10}$$

where c_1, c_2 are hyperparameters to be tuned (Jahrer et al., 2010; Breiman, 1996). To search for the best hyperparameters, we train the rerankers on the training set, and we pick the configuration with the highest validation accuracy.

Results show that rerankers tend to overfit on the training set. To mitigate reranker's overfitting issue, we fine-tune the rerankers on the validation set for b epochs (Tennenholtz et al., 2018). To tune hyperparameter b, we re-split the validation set by 75:25, fine-tune the reranker on the 75% split for b epochs, and pick the b with the highest accuracy on the 25% split. We see that tuning for one epoch ($b = 1$) performs the best. To produce the final fine-tuned ensemble reranker, we fine-tune both rerankers on 100% of the validation set for one epoch, and combine them with the same ensemble hyperparameters c_1, c_2 found before fine-tuning.

Length	Normalization	**NSEdit (ours)**	Baseline	CodeBERT†	GraphCodeBERT†	PLBART†	CoTexT†
Small	Abstract	**24.04**	16.30	16.40	17.3	19.21	<u>22.64</u>
Small	Concrete	**23.86**	<u>17.75</u>	-	-	-	-
Medium	Abstract	<u>13.87</u>	8.91	5.16	9.1	8.98	**15.36**
Medium	Concrete	**13.46**	<u>9.59</u>	-	-	-	-

Table 1: Top-1 exact match accuracy of NSEdit and other code repair models, evaluated on the Tufano et al. (2019) datasets. NSEdit achieved the state-of-the-art result on Tufano small abstract dataset, a part of CodeXGLUE benchmark. The results from CodeXGLUE benchmark or original papers are marked with †. Tufano abstract dataset normalizes variable names, method names and type names. The best and second best results are bold and underlined.

4 Experiments and Results

4.1 NSEdit achieves SOTA performance on CodeXGLUE code repair benchmark

NSEdit achieved the state-of-the-art (SOTA) performance (24.04%) on the Tufano et al. (2019) code repair dataset as a part of the CodeXGLUE benchmark (Lu et al., 2021). We report our results on Tufano datasets in comparison with other code repair methods (Feng et al., 2020; Guo et al., 2020; Ahmad et al., 2021; Phan et al., 2021) currently on the CodeXGLUE benchmark in Table 1. Compared to other methods, our method NSEdit is the only one that predicts any form of edits, while all others predict fixed programs directly. Some example bug fixes correctly produced by our NSEdit model are provided in Appendix E.

In Table 1, the *Baseline* model is a Transformer with the CodeBERT encoder and a randomly initialized six-layer Transformer decoder. Compared to the complete NSEdit model, Baseline has four differences: (1) the prediction target is fixed code by default. (2) the decoder does not have a pointer network for location mode. (3) CodeGPT is not used to initialize the decoder. (4) rerankers are not used. We will reuse this Baseline model in the following experiments.

4.2 Predicting editing sequences performs better than predicting fixed code

We formulate a novel NSEdit grammar to predict editing sequences as the target, rather than the fixed code. To confirm that predicting editing sequences yields better performance, we initialize two Baseline models without CodeGPT and rerankers, the same as in Section 4.1. One Baseline model predicts editing sequences and the other predicts fixed code. The only difference in the architecture is the pointer network needed to predict locations in editing sequences. We report the results on Tufano abstract datasets in Table 2. Predicting edits has better performance, possibly because editing sequence is shorter than fixed code, and it discourages copying behavior by focusing on the changes.

Translation target	Length	Top-1	Top-5
Editing sequences	Small	**21.17**	**37.93**
	Medium	**13.20**	**19.17**
Fixed code	Small	16.30	30.42
	Medium	8.91	17.14

Table 2: Exact match accuracy of two Baseline models when predicting editing sequences and fixed code on Tufano abstract dataset. The Baseline model is the main NSEdit model without CodeGPT and rerankers in order to isolate the effect of translation target.

4.3 NSEdit performs robustly against package-to-package variations

We note that the Tufano et al. (2019) training, validation and test sets have overlapping Java packages. To investigate the effect of package-to-package variations, we curate an in-house dataset from the publicly available 10K Github Java packages (Allamanis and Sutton, 2013), which will be public. Our dataset generation process resembles Tufano et al. (2019): we partition the dataset given the buggy program length and normalize variable names, method names and type names in abstract code, while retaining the original names in concrete code. We implement a strict policy to separate training, validation and test set packages. Other than the same packages, closely related packages that share same naming prefix, e.g. “spring-cloud-stream-samples” and “spring-cloud-stream”, are considered related and also separated in either training, validation or test set. The dataset consists of 138575/12983/9282 train/valid/test samples. We report the accuracy of Baseline models in Table 3.

Recall that we hypothesize previously that predicting fixed code directly encourages the model to learn the copying behavior. We see that when pack-

Packages	Norm.	Target	Top-1		Top-5	
			Val.	Test	Val.	Test
Mixed	Abs.	Code	**26.38**	**9.78**	**41.67**	**21.99**
		Edit	26.14	9.07	38.63	18.51
	Conc.	Code	27.61	3.45	37.45	8.77
		Edit	**30.91**	**7.11**	**38.52**	**12.11**
Separate	Abs.	Code	**12.72**	10.82	**27.50**	26.45
		Edit	12.51	**11.46**	27.03	**27.05**
	Conc.	Code	4.94	3.98	11.93	11.15
		Edit	**9.10**	**9.04**	**16.30**	**18.42**

Table 3: Exact match accuracy of Baseline models that predict on editing sequences and fixed code on our in-house dataset (small). Mixed dataset mixes the training and validation set packages and leave the test set separate, and otherwise all three sets have separate packages. We see that when packages are separate or when input programs have original variable names, it overfits less to predict editing sequences than fixed code directly. CodeGPT and rerankers are not used.

ages are mixed, all models overfit with large gap between validation and test accuracy. Furthermore, mixing packages causes all models to have reduced accuracy, likely because overfitting is a significant performance bottleneck. When predicting code, accuracy on concrete code (27.61%, row 3) is slightly better than accuracy on abstract code (26.38%, row 1) on validation set with mixed training/validation packages, but only half at test time with unseen packages (3.45% v.s. 9.78%), possibly because for concrete code, the model is given more context and it is easier to copy from similar programs, which makes overfitting more severe. Predicting edits does not suffer the same performance drop at test time for concrete code (7.11%, row 4) compared to abstract code (9.07%, row 2), which supports our hypothesis that predicting edits discourages copying behavior. When packages are separate and code is concrete, predicting editing sequences (9.04%, row 8) doubles the test time top-1 accuracy of predicting fixed code directly (3.98%, row 7). This is the most valuable use case in applications, and predicting edits has even greater advantage than predicting fixed code.

4.4 NSEdit is a general bug fix method in different languages and settings

The NSEdit grammar is language agnostic. To confirm that NSEdit works on languages other than Java, we evaluate our NSEdit model on ETH Py150 dataset (Kanade et al., 2020; Raychev et al., 2016). The Python dataset contains five classification tasks, and we experiment on three of them that are related to code repair: variable-misuse classification, wrong binary operator, and swapped operand. We process ETH Py150 according to our problem setup and report the performance of NSEdit in Table 4. We confirm that NSEdit is a general method can edit buggy programs in different languages and settings.

Task	Top-1	Top-5
Variable-misuse classification	83.08	87.14
Wrong binary operator	58.44	75.32
Swapped operand	67.38	69.23

Table 4: Exact match accuracy of NSEdit on three code repair tasks of the ETH Py150 dataset. NSEdit can repair programs in different languages and settings.

4.5 Pre-trained encoder and decoder can be fine-tuned together on editing sequences

Fine-tuning pre-trained models on code repair contributes to the performance of our model. To investigate the effect of different backbone models, we change the pre-trained backbones used to initialize the weights of the Transformer before fine-tuning. We report accuracy on Tufano datasets in Table 5.

Length	Pre-trained backbone	Top-1	Top-3	Top-5
Small	No backbone	15.89	25.00	27.66
	CodeBERT	21.17	33.40	37.93
	CodeBERT+CodeGPT	**22.35**	**33.20**	**36.35**
Medium	No backbone	7.32	10.89	11.81
	CodeBERT	13.20	17.68	19.17
	CodeBERT+CodeGPT	**13.72**	**18.87**	**20.17**

Table 5: Exact match accuracy of NSEdit when different pre-trained backbone models are used to initialize the weights before fine-tuning on Tufano abstract dataset. The pre-trained encoder and decoder can be fine-tuned together in the same model. The decoder, despite not pre-trained on editing sequences, can be fine-tuned to predict edits. Reranking is not performed to isolate the effect. All models predict editing sequences.

The CodeBERT encoder backbone is trained on multiple programming languages, including Java. Even when the input sequences contain bugs and program analysis auxiliary information is not provided, CodeBERT can robustly extract program information and improve the performance of NSEdit after fine-tuning. The CodeGPT decoder backbone also improves the performance of NSEdit. CodeBERT and CodeGPT are two independently published models pre-trained on different datasets, and our results confirm that even when large language models are pre-trained in different settings, they can be integrated into the same Transformer model. Notably, CodeGPT is pre-trained on code, and the model can effectively transfer knowledge to predict editing sequences.

4.6 Rerankers bring significant performance improvements

The use of rerankers significantly improves the performance of NSEdit to the state-of-the-art level. We investigate the effect of different settings of rerankers in Table 6.

Reranker settings	Top-1	Top-2	Top-3	Top-4	Top-5
NSEdit without rerankers	22.35	29.19	33.20	35.22	36.35
Transformer	18.47	26.80	31.55	34.62	36.35
Ensemble	22.76	29.07	32.63	35.08	36.35
Fine-tuned Transformer	21.17	28.67	32.84	35.34	36.35
Fine-tuned ensemble	**24.04**	**30.54**	**34.10**	**35.70**	**36.35**

Table 6: Top-5 exact match accuracy of incremental ablations in reranker settings on Tufano small abstract dataset. Ensemble improves all accuracies compared to a single Transformer reranker. Fine-tuning on validation set improves all accuracies as well. The combination of both achieves the highest accuracy.

The final reranker ablation *Fine-tuned ensemble* achieves the best accuracy among all settings and is presented in the Methods section. Notably, we see that the use of Transformer reranker has lower accuracy than without reranker. Ensemble rerankers and validation-set fine-tuning both improved performance separately and in combination. Further details about reranker ablation experiment settings are provided in Appendix B.

4.7 Ensemble reranker efficiently trades off precision and recall

Rerankers rerank the beam search hypotheses as discussed in Section 3.4. A beam search hypothesis has higher reranking score if it is predicted to be more likely. Therefore, we can use the reranking score computed in Equation 10 as a confidence metric to trade off precision and recall of the editing sequence generated by beam search. If the reranking score is lower than a threshold, we discard the editing sequence predicted (negative), and otherwise we use it (positive). We sweep the threshold parameter and generate the precision-recall tradeoff plot for Tufano small abstract dataset in Figure 3. Note that for accuracy reported previously, the model makes a prediction every time (always positive) and recall is 100%.

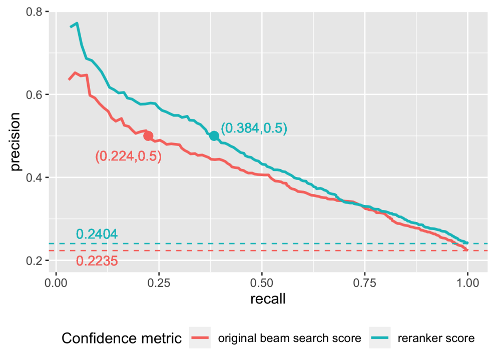

Figure 3: Precision versus recall plot on Tufano small abstract dataset using reranking score and original beam search score as the confidence metric. Among the two, reranking score trades off precision and recall more efficiently, reaching the same precision with higher recall.

When the model is not required to make a prediction for every buggy sequence, precision can be improved with trade-off of recall by using the reranking score as a confidence metric. This trade-off is usually desirable in bug fix applications. Specifically, when recall is lowered from 100% to 38.4%, precision can be increased from 24.04% to 50%. In comparison, we also use beam search's original sum of log probability score computed in Appendix Equation 11 as the confidence metric, and we see that reranking score is more efficient because its recall is higher at every precision level. Specifically, the original score's recall needs to be lowered to 22.4% in order to reach the same precision level of 50%, which is almost half of the reranking score's recall. This result additionally confirms that our rerankers learn an extrinsic signal that differs from the main model's intrinsic confidence.

5 Discussions

NSEdit has achieved a new state-of-the-art performance on the code repair task of the CodeXGLUE benchmark (Lu et al., 2021; Tufano et al., 2019). For code repair, it is more effective to predict editing sequences than fixed code.

As closing thoughts, we want to draw attention that our edit grammar generates a neural-symbolic interface that is a special case of a general paradigm. Integration of deep learning systems with traditional symbolic systems can be achieved by formulating a domain-specific language (DSL) and training a Transformer to use the DSL as a programmable interface.

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333*.

Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. 2021. Self-supervised bug detection and repair. *arXiv preprint arXiv:2105.12787*.

Miltiadis Allamanis and Charles Sutton. 2013. Mining Source Code Repositories at Massive Scale using Language Modeling. In *The 10th Working Conference on Mining Software Repositories*, pages 207–216. IEEE.

Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin Vechev. 2021. Tfix: Learning to fix coding errors with a text-to-text transformer. In *International Conference on Machine Learning*, pages 780–791. PMLR.

Sahil Bhatia, Pushmeet Kohli, and Rishabh Singh. 2018. Neuro-symbolic program corrector for introductory programming assignments. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 60–70. IEEE.

Leo Breiman. 1996. Bagging predictors. *Machine learning*, 24(2):123–140.

Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harri Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Zimin Chen, Steve James 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*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. 2020. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *International Conference on Learning Representations*.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. 2020. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*.

Dawn Drain, Colin B Clement, Guillermo Serrato, and Neel Sundaresan. 2021. Deepdebug: Fixing python bugs using stack traces, backtranslation, and code skeletons. *arXiv preprint arXiv:2105.09352*.

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*.

Alex Graves. 2012. Sequence transduction with recurrent neural networks. *arXiv preprint arXiv:1211.3711*.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*.

Tatsunori B Hashimoto, Kelvin Guu, Yonatan Oren, and Percy Liang. 2018. A retrieve-and-edit framework for predicting structured outputs. *arXiv preprint arXiv:1812.01194*.

Michael Jahrer, Andreas Töscher, and Robert Legenstein. 2010. Combining predictions for accurate recommender systems. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 693–702.

Bader Johannes, Scott Andrew, Pradel Michael, and Chandra Satish. 2019. Getafix: learning to fix bugs automatically. *Proceedings of ACM on Programming Languages*, (10).

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *International Conference on Machine Learning*, pages 5110–5121. PMLR.

Gregory Koch, Richard Zemel, Ruslan Salakhutdinov, et al. 2015. Siamese neural networks for one-shot image recognition. In *ICML deep learning workshop*, volume 2. Lille.

Wouter Kool, Herke Van Hoof, and Max Welling. 2019. Stochastic beams and where to find them: The gumbel-top-k trick for sampling sequences without replacement. In *International Conference on Machine Learning*, pages 3499–3508. PMLR.

Alex M Lamb, Anirudh Goyal Alias Parth Goyal, Ying Zhang, Saizheng Zhang, Aaron C Courville, and Yoshua Bengio. 2016. Professor forcing: A new algorithm for training recurrent networks. In *Advances in neural information processing systems*, pages 4601–4609.

Ann Lee, Michael Auli, and Marc'Aurelio Ranzato. 2021. Discriminative reranking for neural machine translation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 7250–7264.

Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E Gonzalez, and Ion Stoica. 2018. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*.

Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.

Christopher D Manning, Kevin Clark, John Hewitt, Urvashi Khandelwal, and Omer Levy. 2020. Emergent linguistic structure in artificial neural networks trained by self-supervision. *Proceedings of the National Academy of Sciences*, 117(48):30046–30054.

Jiayuan Mao, Chuang Gan, Pushmeet Kohli, Joshua B Tenenbaum, and Jiajun Wu. 2019. The neuro-symbolic concept learner: Interpreting scenes, words, and sentences from natural supervision. *arXiv preprint arXiv:1904.12584*.

Nathan Ng, Kyra Yee, Alexei Baevski, Myle Ott, Michael Auli, and Sergey Edunov. 2019. Facebook fair's wmt19 news translation task submission. *arXiv preprint arXiv:1907.06616*.

Long Phan, Hieu Tran, Daniel Le, Hieu Nguyen, James Anibal, Alec Peltekian, and Yanfang Ye. 2021. Cotext: Multi-task learning with code-text transformer. *arXiv preprint arXiv:2105.08645*.

Md Rafiqul Islam Rabin, Nghi DQ Bui, Ke Wang, Yijun Yu, Lingxiao Jiang, and Mohammad Amin Alipour. 2021. On the generalizability of neural program models with respect to semantic-preserving program transformations. *Information and Software Technology*, 135:106552.

Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training.

Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. 2021. Zero-shot text-to-image generation. *arXiv preprint arXiv:2102.12092*.

Roshan Rao, Jason Liu, Robert Verkuil, Joshua Meier, John F Canny, Pieter Abbeel, Tom Sercu, and Alexander Rives. 2021. Msa transformer. *bioRxiv*.

John W Ratcliff and David E Metzener. 1988. Pattern-matching-the gestalt approach. *Dr Dobbs Journal*, 13(7):46.

Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016. Probabilistic model for code with decision trees. *ACM SIGPLAN Notices*, 51(10):731–747.

D Raj Reddy et al. 1977. Speech understanding systems: A summary of results of the five-year research effort. *Department of Computer Science. Camegie-Mell University, Pittsburgh, PA*, 17:138.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2015. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112.

Guy Tennenholtz, Tom Zahavy, and Shie Mannor. 2018. Train on validation: squeezing the data lemon. *arXiv preprint arXiv:1802.05846*.

Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(4):1–29.

Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. 2019a. Neural program repair by jointly learning to localize and repair. In *International Conference on Learning Representations*.

Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. 2019b. Neural program repair by jointly learning to localize and repair. *arXiv preprint arXiv:1904.01720*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer networks. *arXiv preprint arXiv:1506.03134*.

Ronald J Williams and David Zipser. 1989. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2):270–280.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2019. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*.

Ziyu Yao, Frank F Xu, Pengcheng Yin, Huan Sun, and Graham Neubig. 2021. Learning structural edits via incremental tree transformations. *arXiv preprint arXiv:2101.12087*.

Michihiro Yasunaga and Percy Liang. 2021. Break-it-fix-it: Unsupervised learning for program repair. *arXiv preprint arXiv:2106.06600*.

Pengcheng Yin, Graham Neubig, Miltiadis Allamanis, Marc Brockschmidt, and Alexander L Gaunt. 2018. Learning to represent edits. *arXiv preprint arXiv:1810.13337*.

Chulhee Yun, Srinadh Bhojanapalli, Ankit Singh Rawat, Sashank J Reddi, and Sanjiv Kumar. 2020. Are transformers universal approximators of sequence-to-sequence functions? In *8th International Conference on Learning Representations, 2020*.

Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A syntax-guided edit decoder for neural program repair. *arXiv preprint arXiv:2106.08253*.

A Beam search formal definition

During beam search, we maintain K incomplete subsequences (i.e. beams) $B_{i,k}$, where $B_{0,k} = [$<*bos*>$]$ for all $k = 1, 2...K$. For any $B_{i,k}$, we compute the probability $\Pr(\hat{e}_{i+1,k}|\hat{e}_{1,k}, \hat{e}_{2,k}, ...\hat{e}_{i,k}) = \hat{\mathbf{E}}_{i+1,k}[\hat{e}_{i+1,k}]$ given Equation 6. The probability of the concatenated sequence $B_{i+1,k} = B_{i,k} + [\hat{e}_{i+1,k}] = [\hat{e}_{1,k}, \hat{e}_{2,k}, ...\hat{e}_{i,k}, \hat{e}_{i+1,k}]$ is the product of the probabilities of all tokens as they are generated. Formally,

$$\Pr(\hat{e}_1, \hat{e}_2...\hat{e}_{i+1}) = \prod_{j=1}^{i+1} \Pr(\hat{e}_j|\hat{e}_1, ...\hat{e}_{j-1}) \quad (11)$$

We denote this product as Π in Equation 10. Taking logarithm on both sides, we have

$$\begin{aligned} \log \Pr(B_{i,k} + [\hat{e}_{i+1,k}]) = &\log \Pr(B_{i,k}) + \\ &\log \hat{\mathbf{E}}_{i+1,k}[\hat{e}_{i+1,k}] \end{aligned} \quad (12)$$

We take the top-K most probable tokens $\hat{e}_{i+1,k} \in W$ for all beam $k = 1, 2, ...K$, given

$$\underset{(k,\hat{e}_{i+1,k}) \in \{1,2,...K\} \times W}{\arg \operatorname{topK}} \log \Pr(B_{i,k} + [\hat{e}_{i+1,k}]) \quad (13)$$

where any previous step beam may have multiple concatenated beams in top-K with different tokens. The selected top-K tokens $\hat{e}_{i+1,k}$ form a new set of K beams for the next iteration, until <*eos*> is predicted or maximum length is reached. Note that since $\arg \operatorname{topK}$ is applied to $\{1, 2, ...K\} \times W$, it is possible for different tokens to append to the same previous beam and are all included in the top-k beams for the next step. Also note that beam $B_{i+1,k}$ may not contain beam $B_{i,k}$ as a subsequence.

B Reranker ablation experiments details

In this section, we provide more details on the settings of the reranker ablation experiments presented in Table 6.

The *NSEdit without rerankers* version does not use rerankers and directly reports beam search accuracy based on the original beam search score Π in Equation 11. We compare the reranked accuracy with this baseline.

The *Transformer* version trains a single Transformer reranker to rerank the beam search hypotheses, ignoring the original beam search score. We see that the accuracy is lower than the accuracy of *NSEdit without rerankers*.

The *Ensemble* version trains both rerankers and blends the three ranking scores by optimizing validation accuracy. We see that the ensemble reranker improves over *Transformer* reranker but not better than *NSEdit without rerankers*.

The *Fine-tuned Transformer* version trains a single Transformer reranker and tunes it on validation set for one epoch. We see that it significantly outperforms the *Transformer* version without fine-tuning, which suggests that fine-tuning on validation prevents the reranker from overfitting. The accuracy of this ablation outperforms *NSEdit without rerankers*.

The final version *Fine-tuned ensemble* uses a blended ensemble of two rerankers fine-tuned on the entire validation set, as described in the Methods section.

C Training and model hyperparameters

We implement our Transformer architecture in PyTorch, except the pre-trained CodeBERT and CodeGPT models, which we load from HuggingFace (Wolf et al., 2019) . The learning rate is set to be 1e-4 multiplied by the number of GPUs. When CodeGPT weights are loaded, we halve the learning rate of pre-trained parameters and quadruple the learning rate of randomly initialized parameters. AdamW optimizer (Loshchilov and Hutter, 2017) with triangular learning rate scheduler is used in all experiments. The NSEdit main model is trained for at most 60 epochs and early stopping is applied if the accuracy does not improve. Automatic mixed precision (AMP) is enabled. Training of the model together with rerankers on Tufano datasets takes around a day on a machine with 4 V100 Nvidia GPUs.

In beam search, a length penalized score is computed for the partially generated sequences at every step according to Wu et al. (2016). After the scores are computed, the finite state machine set the invalid tokens to have zero probability according to the edit grammar as described in Section 3.3.

Rerankers are trained on buggy programs for which the beam search produces at least one correct editing sequence. We train rerankers for 12 epochs, with the same learning rate setup as the main NSEdit model. We fine-tune the reranker on validation set for 1 epoch with 1 GPU.

We find the best hyperparameters with Ray tune (Liaw et al., 2018) or grid search. The ensemble reranker on Tufano datasets have coefficients reported in Table 7. Each coefficient is searched among 10 candidates in logarithmic interval $[0.01, 100]$, then another 20 candidates in a narrower linear interval $[0.1, 2]$.

Length	Normalization	Transformer c_1	Encoder c_2
Small	Abstract	0.4	0.4
	Concrete	1.0	0.7
Medium	Abstract	0.2	0.3
	Concrete	0.3	0.1

Table 7: The ensemble reranker hyper-parameters found through grid search for Tufano datasets. The ensemble reranker is a linear model with Equation 10.

D Tufano dataset editing sequence statistics

In Table 8, we summarize statistics about the ground truth editing sequences in Tufano et al. (2019) datasets. We see that the editing sequences in Tufano medium, compared to Tufano small dataset, have more number of edits, longer insertion length and longer overall editing sequence length by mean and median. This suggests that the bug fixes in Tufano medium dataset are overall more difficult to predict correctly.

	Tufano	Mean	Median
Number of edits	small	2.02	2
	medium	2.45	2
	combined	2.24	2
Insertion length	small	3.60	1
	medium	6.22	2
	combined	4.99	1
Editing sequence length	small	10.8	8
	medium	14.4	10
	combined	12.7	8

Table 8: The statistics of the ground truth editing sequences on Tufano datasets. We see that the editing sequences from Tufano medium, compared to Tufano small dataset, have more number of edits, longer insertion length and longer overall editing sequence length by mean and median.

E Example bug fixes

We provide some examples of bug fix that are correctly produced by NSEdit in Figures 4 to 12.

```
- public void write(byte b[]) throws IOException {
?                        --
+ public void write(byte[] b) throws IOException {
?                      ++
    assertOpen();
    super.write(b);
  }
```

Figure 4: Example bug fix. Coding style improvement. The predicted editing sequence is [INSERT][LOC_6][][DELETE][LOC_7][LOC_9][INSERT][LOC_9])

```
  @Override public Iterator<?> downstreams(){
    WindowGroupedFlux<T> g=window;
+   if (g == null) {
-   if (g == null)   return Collections.emptyList().iterator();
?  ---------------
+     return Collections.emptyList().iterator();
+   }
    return Collections.singletonList(g).iterator();
  }
```

Figure 5: Example bug fix. Coding style improvement. The predicted editing sequence is [DELETE][LOC_31][LOC_34][INSERT][LOC_34] {return[INSERT][LOC_41]}

```
  public void addPoint(Point2D point){
    ArcPoint newPoint=new ArcPoint(point,false);
-   HistoryItem historyItem=new AddArcPathPoint<S,T>(arc,newPoint);
?                                               ---
+   HistoryItem historyItem=new AddArcPathPoint<>(arc,newPoint);
    historyItem.redo();
    historyManager.addNewEdit(historyItem);
  }
```

Figure 6: Example bug fix. Coding style improvement. The predicted editing sequence is [DELETE][LOC_37][LOC_40][DELETE][LOC_64][LOC_66]

```
  public long getConsoleReportingInterval(){
-   System.out.println(reportingIntervalConsole.getValue());
    return reportingIntervalConsole.getValue();
  }
```

Figure 7: Example bug fix. Remove unnecessary logging statement. The predicted editing sequence is [DELETE][LOC_9][LOC_24]

```
- public String getName(){
+ @Override public String getName(){
? ++++++++++
    return CypherPsiImplUtil.getName(this);
  }
```

Figure 8: Example bug fix. Missing annotation. The predicted editing sequence is [DELETE][LOC_1][LOC_2][INSERT][LOC_2]@Override public

```
  @Override public boolean apply(PickleEvent pickleEvent){
    String picklePath=pickleEvent.uri;
    if (!lineFilters.containsKey(picklePath)) {
-     return true;
?            ^^^
+     return false;
?            ^^^^
    }
    for (  Long line : lineFilters.get(picklePath)) {
      for (    PickleLocation location : pickleEvent.pickle.getLocations()) {
        if (line == location.getLine()) {
          return true;
        }
      }
    }
    return false;
  }
```

Figure 9: Example bug fix. Logical error. The predicted editing sequence is `[DELETE][LOC_44][LOC_45][INSERT][LOC_45] false`

```
  /**
   * Returns the preferred fragment size.
   * @param format target format
   * @return the preferred fragment size
   * @throws IOException if failed to compute size by I/O error
   * @throws InterruptedException if interrupted
   * @throws IllegalArgumentException if some parameters were {@code null}
   */
  public long getPreferredFragmentSize(FragmentableDataFormat<?> format) throws
      ↪ IOException, InterruptedException {
    if (format == null) {
      throw new IllegalArgumentException("format␣must␣not␣be␣null");
    }
    long min=getMinimumFragmentSize(format);
-   if (min <= 0) {
?            -
+   if (min < 0) {
      return -1;
    }
    long formatPref=format.getPreferredFragmentSize();
    if (formatPref > 0) {
      return Math.max(formatPref,min);
    }
    return Math.max(preferredFragmentSize,min);
  }
```

Figure 10: Example bug fix. Logical error. The predicted editing sequence is `[DELETE][LOC_140][LOC_141][INSERT][LOC_141] <`

```
  public boolean equals(AudioQuality quality){
    if (quality == null)   return false;
-   return (quality.samplingRate == this.samplingRate & quality.bitRate ==
    ↪ this.bitRate);
+   return (quality.samplingRate == this.samplingRate && quality.bitRate ==
    ↪ this.bitRate);
?                                                        +
  }
```

Figure 11: Example bug fix. Wrong operator. The predicted editing sequence is `[DELETE][LOC_35][LOC_36][INSERT][LOC_36] &&`

```
  @Override public void run(){
    this.ownerThread=Thread.currentThread();
    Log.debug("Starting␣event␣loop","name",name);
    setStatus(LoopStatus.BEFORE_LOOP);
    try {
      beforeLoop();
    }
   catch (  Throwable e) {
-     Log.error("Error occured before loop is started","name",name,"error",e);
+     Log.error("Error occurred before loop is started","name",name,"error",e);
?                          +
      setStatus(LoopStatus.FAILED);
      return;
    }
    setStatus(LoopStatus.LOOP);
    while (status == LoopStatus.LOOP) {
      if (Thread.currentThread().isInterrupted()) {
        break;
      }
      try {
        insideLoop();
      }
   catch (    Throwable e) {
        Log.error("Event␣loop␣exception␣in␣" + name,e);
      }
    }
    setStatus(LoopStatus.AFTER_LOOP);
    afterLoop();
    setStatus(LoopStatus.STOPPED);
    Log.debug("Stopped␣event␣loop","name",name);
  }
```

Figure 12: Example bug fix. Spelling error. The predicted editing sequence is `[DELETE][LOC_68][LOC_70][INSERT][LOC_70] occurred`