SYNC: EFFICIENT NEURAL CODE SEARCH THROUGH STRUCTURALLY GUIDED HARD NEGATIVE CURRICULA

Anonymous authors
Paper under double-blind review

ABSTRACT

Efficient code snippet search using natural language queries can be a great productivity tool for developers (beginners and professionals alike). Recently neural code search has been popular, where a neural method is used to embed both the query (NL) and the code snippet (PL) into a common representation space; which is further used to obtain the most relevant PL satisfying the intent in the query. Transformers-based pre-trained language models (such as CodeBERT, GraphCodeBERT, and UniXCoder) have been especially effective to learn such representation. These models often make mistakes such as retrieving snippets with incorrect data types, and incorrect method names or signatures; even when exposed to the underlying structural information of the code (such as Abstract Syntax Tree and other static analysis outputs) during pre-training. The generalization ability beyond the training data is also limited (as the code retrieval datasets vary in the ways NL-PL pairs are collected). In this work, we propose a structure-aware hard negative sampling method combined with a mastering-rate based curriculum learning technique (SYNC) that enhances the pre-trained representation using both soft (random) and the (synthesized) hard negative samples. Our experiments show significant improvements (up to 5% in MRR) over all three state-of-the-art pre-trained language models (for PL) over four Python code retrieval datasets (under both in-distribution and out-of-distribution settings).

1 INTRODUCTION

Learning dense representations for programming languages using NLP techniques has proven effective for both downstream retrieval and generative tasks. Representations have evolved from fixed token-wise representations (such as Code2Vec, Code2Seq) to contextualized Transformers-based representations such as CodeBERT (Feng et al. (2020)), and GraphCodeBERT (Guo et al. (2020)). Parallel to the evolution of NLP models, universal cross-modal models such as UniXCoder (Guo et al. (2022)) and generative models such as AlphaCode (Li et al. (2022)) have achieved state-of-the-art in public benchmarks. Unlike natural language, (ideally) the output programming language is expected to be consumed by compilers (or interpreters) which expect the code to follow well-defined syntax and semantics. In this work, we explore whether such information about syntax and semantics expressed in natural languages is preserved while retrieving corresponding code snippets in the code retrieval task (primarily for Python). Our initial exploration shows that state-of-the-art Transformers-based code embedding models consistently make mistakes such as retrieving intents with wrong data types (sets instead of lists), wrong method names or signatures and wrong arguments (`strftime` vs `strptime` from python library) as shown in Tab. 1.

As a remedy, we look towards contrastive learning using dynamic structure-aware negative sampling. Recently, researchers (Robinson et al. (2021); Ahrabian et al. (2020)) have shown how synthesized hard negative sampling can be used along with a contrastive loss objective to learn efficient representations. To stabilize training (Xuan et al. (2020)), we utilize a mix of random negative samples, alongwith hard negatives generated using perturbations of the Abstract Syntax Tree for the positive NL-PL pair. To balance the hardness and the learning state of the model, we use the mastering rate-based curriculum approach to sample a mix of soft and hard negatives, while hard negatives are sampled using a parametrized distribution over model scores Robinson et al. (2021). We ob-

Query	Expected	Retrieved	Error Type
Sorting a list of lists in Python	`c2.sort(key=lambda row: (row[2], row[1]))`	`sorted(list_of_strings, key=lambda s: s.split(',')[1])`	Sorts list of strings instead of list of lists
Using %f with strftime() in Python to get microseconds?	`datetime.datetime.now() .strftime('%H:%M:%S.%f')`	`time.strptime('30/03/09 16:31: 32.123','%d/%m/%y %H:%M:%S.%f')`	Wrong function call
How to remove symbols from a string with Python?	`re.sub('[\\^w]', ' ', s)`	`re.sub('\', '', 'aas30dsa20')`	Correct function call, wrong arguments

Table 1: We show the search query (from CoNaLa test set), the corresponding expected PL snippet, the top ranked snippet by GraphCodeBERT. We mention the type of retrieval error in the last column.

serve that our approach helps boost learning efficiency for three code embedding models across four code retrieval datasets (CoNaLa, PyDocs, CodeSearchNet, and WebQuery). Our experiments show that the proposed AST-based curriculum approach (SYNC) with a contrastive loss can be used to effectively integrate structure information of programming languages during the fine-tuning stage for SOTA code embedding models, including ones such as UniXcoder, which is exposed to ASTs in the pre-training stage. Specifically, our contributions are the following. We propose 1) a structure aware, AST perturbation based hard negative sampling approach; and 2) a mastering rate-based curriculum for balancing hard and soft negatives in contrastive learning. We perform 3) comprehensive evaluation of our approach on three SOTA models across four retrieval datasets.

2 Related Work

Neural Code Search. Transformer-based pre-trained language models (such as BERT, RoBERTa, and GPT) have proven to be quite successful across various language understanding tasks. Similar models when trained on code corpora with programming language (PL) oriented pre-training objectives Kanade et al. (2019); Feng et al. (2020) perform well on various PL understanding tasks such as code search, clone detection, code translation, and code refinement. For neural code search, both encoder-only and encoder-decoder architectures exist, with decoder-only models (Svyatkovskiy et al. (2020); Lu et al. (2021)) being more successful for generative tasks (Guo et al. (2022)). Encoder-only models use a bidirectional transformer with a "masked language modeling" (MLM) objective along with PL specific objectives. CodeBERT Feng et al. (2020) is trained on github public repositories using replaced token detection (RTD) objective apart from MLM, which allows it to use both bimodal and unimodal data. GraphCodeBERT Guo et al. (2020) incorporates dataflow information as input as well as two additional pre-training objectives of edge prediction and node alignment. SYNCOBERT Wang et al. (2021a) is a syntax guided pre-training approach which uses two additional objectives of Identifier Prediction and AST Edge Prediction and optimizes mutual information between multiple modalities. AstBERT (Liang et al. (2022)) uses pruned AST of source code as input for training. TreeBERT Jiang et al. (2021) changes the AST of code into a collection of composition paths, adds a node position embedding, and uses two new objectives - Tree Masked Language Modeling (TMLM) and Node Order Prediction (NOP).
Among *encoder-decoder* models, BART-based PLBART Ahmad et al. (2021) is pre-trained using denoising strategies for better understanding of program syntax and style. CodeT5 Wang et al. (2021b) supports multitask learning and uses a pre-training objective that depends on code token type information such as identifiers, and keywords. UniXcoder Guo et al. (2022) is a unified cross-modal model which incorporates the AST of the code, a denoising objective shown to be effective for encoder-decoder models & code fragment representation learning tasks in addition to MLM and causal language modeling (CLM). In this work, we propose a structure-aware AST perturbation-based approach to generate hard negatives and combine it with a mastering rate-based curriculum Willems et al. (2020) to improve the underlying code representation. To this end, we test the effectiveness of our approach on CodeBERT, GraphCodeBERT, and UniXcoder as they span various input modalities such as text, dataflow, and AST as well as encoder-decoder and encoder-only architecture configurations.
Contrastive Learning with Hard Negatives. Mining *hard* negatives for efficient contrastive learning has found several applications in the field of image processing. Recent work (Xuan et al. (2020)) shows why hard negatives lead to unstable training behavior, but can be useful with some simple fixes. Robinson et al. (2021) proposes a learnable distribution that models the hardness using the inner product between the examples and underlying model embedding. Using this hardness as a re-weighting mechanism, authors show how to sample suitable hard negatives that can most benefit

the learning process. Hard negatives are difficult to learn from in the early stages of training, which prompted researchers to explore curriculum learning strategies Chu et al. (2021). However Chu et al. (2021) proposes a static curriculum where negative samples are scored, sorted from easy-to-hard, and batched. Willems et al. (2020) propose a mastering-rate-based framework that generates a probability distribution over interdependent tasks based on the dependency relationship between them and the degree to which they have been learned (or *mastered*) by the model. We employ a mastering rate curriculum which guides the model to learn from both soft (randomly sampled) and hard AST-guided negatives while taking the model learning state into account.

Inducing & Detecting Bugs in Software. Our AST guided perturbations for generating hard negatives draw upon literature to induce and detect bugs in software Allamanis et al. (2021); Patra & Pradel (2021); Pradel & Sen (2018), detect API misuse Wen et al. (2019), and name value pair consistencies Patra & Pradel (2022). Pradel & Sen (2018) use simple code transformations to create artificially seeded bugs to learn a name-based semantic bug detector. They build bug detectors for detecting accidentally swapped arguments, incorrect binary operators and operands for JavaScript code. Patra & Pradel (2021) propose a technique to semantically adapt bug patterns to local context by learning token embeddings that capture semantic similarities between variables and literals. Patra & Pradel (2022) dynamically analyze the assignment of values to a variable and learn a model to predict if a variable name is appropriate for the value it holds. Wen et al. (2019) propose eight mutation operators (altering API call sequences, conditional branching, API call arguments, etc.) to detect API misuse using mutation testing. We draw upon some of these patterns as sources of inspiration for our AST perturbation rules. Allamanis et al. (2021) propose PyBugLab, a code rewriting-based approach for Python to induce typographical bugs that might arise from copy-paste errors and are statistically common Karampatsis & Sutton (2020); Just et al. (2014). They learn a pointer network Merity et al. (2017) to localize the bug and a set of rules (variable, function argument swapping, operator substitution & operand corruption) and associated score functions to determine the most likely bug. Some of the rules proposed in this work are similar in nature (rule 6 and "wrong operator", rule 4, 5 and "wrong literal"), but we generate all possible corrupted candidates applying one rule at a time, and use the current model weights to rank the corrupted code snippets against the intent to find code snippets the model is likely to be confused by.

3 METHOD

For neural code search, Transformers based pretrained encoders are finetuned on annotated NL-PL pairs. Our method is targeted towards improving the learnt representation during this finetuning stage, by utilizing carefully synthesized hard negative samples. In this stage, we follow the triplet network architecture (proposed by Hoffer & Ailon (2015)), where the NL query (x), a positive PL snippet (y^+) and a negative PL (y^-) snippet are sampled and fed to a network individually. In general, this network is a unified representation learner, that is trained to embed both text and code into a joint embedding space. Here, we use transformer-based pretrained encoders. After the triplet is encoded, we use the contrastive loss to minimize the relative distances between the positive pair ($\langle x, y^+ \rangle$) with respect to the negative pair ($\langle x, y^- \rangle$). Negative sampling lies at the core of this training paradigm. In this work, we propose an improvement on the regular contrastive learning, by additionally finetuning the network with synthesized *hard* negatives (negative PL snippets) through well-specified AST-based perturbation rules. The set of rules are inspired by generic code constructs and the abstract type of errors the SOTA models are observed to make. In a way, these perturbation rules are targeted to *fix* such known types of errors, while making representation learner more robust. As these synthesized hard negatives are harder to distinguish (i.e., not easy to learn from) in the initial phases of learning, we further adopt the mastering rate based curriculum learning Willems et al. (2020) approach to learn from both soft and hard negatives.

3.1 GENERATING NEGATIVE SAMPLES THROUGH AST PERTURBATION

For code retrieval, most errors made by Transformers-based SOTA methods can be mapped to specific syntactic and semantic constructs. We group the errors into three broad categories: **Type-1** (Data type mismatch): incorrect data types or data structures used (e.g. list comprehension instead of set comprehension, the addition of a string to a variable instead of an integer, etc.) **Type-2** (Function call errors): Incorrect function is called or correct function is called with incorrect arguments. **Type-3** (Incorrect conditional checks): Errors in branching, incorrect comparison ("==" instead of

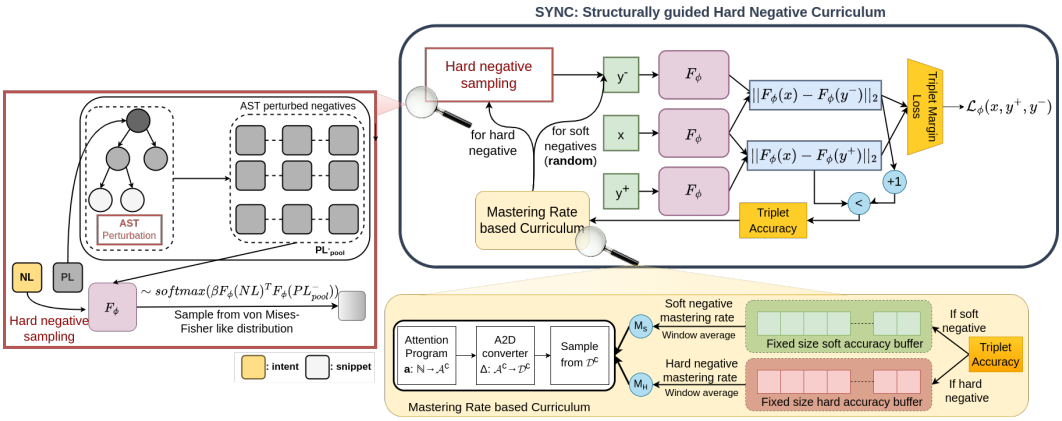

Figure 1: SYNC: Various stages of the proposed AST-guided curriculum.

"!=") or logical operators (**and**, **or**). The first example in Table 1 represents an error of Type-1 while the other two represent errors of Type-2. To make models robust against making such errors, we use contrastive learning with synthesized hard negatives based on *AST-perturbation* rules. We first generate a set of candidate hard negatives and sample the final hard negatives using the underlying code search model dynamically. We explain the two steps below.

Generation of Candidate Hard Negatives. We use AST-perturbation rules to generate hard negatives, as outlined in Table 2. We carefully design these rules to capture the common patterns in mistakes made by SOTA models with inspiration from previous works Allamanis et al. (2021); Wen et al. (2019). An AST-perturbation rule substitutes a node of a specific type in the AST with a different node. Most[1] of the rules ensure that the perturbed code snippets do not satisfy the intent of the original code, i.e. for a code snippet C, if a rule produces a code snippet C', then there exists a test case t such that the output of C, $C(t)$ is not equal to $C'(t)$ (i.e., $\exists t \ni C(t) \neq C'(t)$).

We will now briefly explain how the rules directly address the error types outlined above. Rule-1 addresses Type-2 errors, by replacing standard library functions with the closest library function based on function name and signature. To find the closest library functions, we retrieve k(=10) functions from a list of 5.9k function specifications gathered from standard python modules and some well-known data science libraries like NumPy, pandas, etc. present in the dataset by scoring their similarity based on lexical and function signature overlap (detailed in Appendix section A.2). Notably, we just change the name of the function in the invocation and not the arguments. Rule-4 addresses errors of Type-1 by replacing integer or floating point constants with quoted string versions of them (e.g. `x+3` to `x+"3"`) and replacing string constants with integer or float valued constants equal to the length of the string (e.g. `"hi"*n` to `2*n`). Finally rules such as rules 5, 6, 7, and 9 address errors of Type-3 by removing branching (rule 9), flipping conditional expressions (rule 5, 6), or altering composite conditions (rule 7). We do not chain the application of multiple rules, as it can lead to a code snippet that ends up satisfying the original intent. For example, `if x == True:print("Hello")` would be semantically equivalent to `if x != False:print("Hello")`, which can be obtained from the original code snippet by chained application of rules 5 and 6.

As shown in Algorithm 1 (in Appendix), we generate a set of corrupted code candidates by applying the rules discussed above. To implement our AST perturbation algorithm, we parse the code snippets into abstract syntax trees (ASTs) using Python's AST parser[2] module (for Python 3.7). Then we use node level substitutions to perturb the AST and unparse it using the AST unparser tool[3] to recover the corresponding code.

[1]For some rules such as rules 2 & 3, which convert list comprehensions into set comprehensions and vice-versa, it is possible to design code snippets that have the same output for all test cases irrespective of whether a set or list is used. Additionally, for rule 1 the substituted function call might end up being identical both in behavior and input specifications to the original function call in rare cases.

[2]https://docs.python.org/3.7/library/ast.html

[3]https://github.com/python/cpython/blob/3.7/Tools/parser/unparse.py

Sampling of Hard Negatives. Once we generate a candidate set of hard negatives c_i through AST perturbation, we use the current model weights to score them against the NL intent or query q and sample hard negatives by using the probability distribution given by the *softmax* over the scores, as $\frac{e^{q^T c_i}}{\sum_i e^{q^T c_i}}$ similar to Robinson et al. (2021) (equivalent to von Mises-Fisher distribution with uniform prior over candidates). The concentration parameter β controls the hardness of the sample hard negatives. A high beta leads to a distribution that picks candidates that the model thinks are most similar to the intent, leading to harder negatives, while a low beta is close to a uniform distribution, making each hard negative candidate equally likely, leading to softer negatives.

	Rule	Input Pattern	Perturbed Output	Description
1	Library function substitution	$f(exp_1, exp_2, \ldots exp_n)$	$f'(exp_1, exp_2, \ldots exp_n)$	Replace standard library functions with closest library function based on function name and signature
2	List comprehension to set comprehension	$[exp_1, exp_2, ...exp_n]$ $[x$ **for** x **in** exp_1 **if** $exp_2]$	$\{exp_1, exp_2, \ldots exp_n\}$ $\{x$ **for** x **in** exp_1 **if** $exp_2\}$	Replace list comprehension with set comprehension (Box brackets to curly brackets)
3	Set comprehension to list comprehension	$\{exp_1, exp_2, \ldots exp_n\}$ $\{x$ **for** x **in** exp_1 **if** $exp_2\}$	$[exp_1, exp_2, \ldots exp_n]$ $[x$ **for** x **in** exp_1 **if** $exp_2]$	Replace set comprehension with list comprehension (Curly brackets to box brackets)
4	Convert integer/float constants to strings and vice versa	$dig_1 \ldots dig_n$ $dig_1 \ldots dig_n.dig'_1 \ldots dig'_m$ "$char_1 \ldots char_n$" "$char_1 \ldots char_n$"	"$dig_1 \ldots dig_n$" "$dig_1 \ldots dig_n.dig'_1 \ldots dig'_m$" n $n.0$	Substitute integer constant with string (enclose in quotation marks) and replace string with integer or floating value equal to the length of the string
5	Flip boolean constants	exp == **True** exp != **False**	exp == **False** exp != **True**	Replace 'True' with 'False' and vice-versa
6	Flip comparators	$exp_1 == exp_2$ $exp_1 != exp_2$ $exp_1 < exp_2$ $exp_1 > exp_2$ $exp_1 \geq exp_2$ $exp_1 \leq exp_2$	$exp_1 != exp_2$ $exp_1 == exp_2$ $exp_1 \geq exp_2$ $exp_1 \leq exp_2$ $exp_1 < exp_2$ $exp_1 > exp_2$	Flip comparators <to >=, >to <=, == to !=, "is" to "is not", "in" to "not in" and vice-versa
7	Flip boolean operators	exp_1 **or** exp_2 exp_1 **and** exp_2	exp_1 **and** exp_2 exp_1 **or** exp_2	Replace "and" with "or" and vice-versa in composite boolean expressions
8	Replace function calls with identifier name	$f(exp_1, \ldots exp_n)$ v *op*= $f(exp_1, \ldots exp_n)$ v = exp' *op* $f(exp_1, \ldots exp_n)$	f v *op*= f v = exp' *op* f	Replace function call with identifier of the same name
9	Replace If-Else statement or expression with its body	**if** exp: s_1; **else** s_2; **if** exp_1: s_1; **elif** exp_2: s_2; ... **else**: s_n; $expr_1$ **if** exp_2 **else** exp_3	s_1 s_1 exp_1	Remove branching in the form of if-else statements, if-else if ladders or inline if-else expressions with the body of the if statement

Table 2: AST perturbation rules and their corresponding grammars in Python's Abstract Syntax Description Language (or ASDL) format. ASL has 4 inbuilt data types: identifier, int, string, constant

3.2 TRAINING CURRICULUM

To carefully learn from both soft and hard negatives (Zhan et al. (2021)), we use a mastering-rate (Willems et al. (2020)) based training curriculum approach. Willems et al. (2020) define curriculum learning by 1) a *curriculum* i.e. a set of tasks $\mathcal{C} = \{c_1, \ldots, c_n\}$, where a task is set of examples of similar type with a sampling distribution, and 2) a *program* which for each training step defines the tasks to train the learner given its learning state and the curriculum. Formally, the program $d : \mathbb{N} \rightarrow \mathcal{D}^{\mathcal{C}}$, is a sequence of distributions over $\mathcal{C}$. To learn tasks that are *learnable but not learnt yet*, the mastering-rate based algorithm requires as input a directed graph over tasks in $\mathcal{C}$. An edge from A to B indicates that learning task A before B is preferable. The learnability for each task depends on mastering rate ($\mathcal{M}_c(t)$) estimated from the normalized mean accuracy for that task at time-step t. To estimate the distribution over examples, at each time-step, the algorithm computes *attention* ($a : \mathbb{N} \rightarrow \mathcal{A}^{\mathcal{C}}$) over the tasks ($a_c(t)$) from mastering rates of its ancestors and successors (in the DAG). Finally, it uses an *attention-to-distribution converter* ($\Delta : \mathcal{A}^{\mathcal{C}} \rightarrow \mathcal{D}^{\mathcal{C}}$) which converts the attention to a distribution over $\mathcal{C}$, which is used to sample minibatches during training.

For our curriculum, we consider two sub-tasks, i.e., hard negative and soft negative learning, where learning from soft negatives is preferable before hard negatives. We generate a distribution over these two tasks as a function of the current mastering rate (windowed triplet accuracy) for each learning task. We compute the mastering rate for a task L at the t^{th} step using $\mathcal{M}_L^{(t)} = \frac{\sum_{i=0}^{k} (\mathcal{T}_a)_L^{(t-i)}}{k}$,

where $(\mathcal{T}_a)_L^{(t-i)}$ is the triplet accuracy at the $(t-i)^{th}$ step for L, and k is the window size. The mastering rates are used to determine the *attention* over the hard ($a_h^{(t)}$) and soft negative ($a_s^{(t)}$) learning tasks at the t^{th} step as follows:

$$a_s^{(t)} = (\delta \cdot (1 - \mathcal{M}_s^{(t)}) + (1 - \delta) \cdot \hat{\gamma}_s^{linreg}(t)) \cdot (1 - \mathcal{M}_h^{(t)}) \tag{1}$$

$$a_h^{(t)} = (\mathcal{M}_s^{(t)})^p \cdot (\delta \cdot (1 - \mathcal{M}_h^{(t)}) + (1 - \delta) \cdot \hat{\gamma}_s^{linreg}(t)). \tag{2}$$

Here $\hat{\gamma}_s^{linreg}$ is the slope of the linear regression over the values of the triplet accuracy for the last k steps (window size), while δ is a coefficient that weighs the contribution of the mastering rates $\mathcal{M}_s^{(t)}$ and $\mathcal{M}_h^{(t)}$, and $\hat{\gamma}_s^{linreg}$. Finally, we compute $\Delta(a^{(t)})$, the probability distribution over the two learning tasks at the t^{th} step, as shown in Eqn. 3 as the weighted combination between the softmax over the attention weights and a bias distribution Δ_{bias} with epsilon being the weight of the bias distribution. Willems et al. (2020) assume a uniform distribution as the bias distribution, but we find a weight of 0.8 for soft negatives and 0.2 for hard negatives to be more suitable for our setting.

$$\Delta(a^{(t)}) := (1 - \epsilon) \cdot \frac{e^{a_c^{(t)}}}{\sum_{c'} e^{a_{c'}^{(t)}}} + \epsilon \cdot \Delta_{Bias} \tag{3}$$

We compute the triplet accuracy $\mathcal{T}_a$ to estimate the mastering rates $\mathcal{M}_S^{(t)}$ and $\mathcal{M}_H^{(t)}$ using Eqn. 4:

$$\mathcal{T}_a = \frac{\sum_{i=0}^{N} \mathbf{1}_{\|x_i - y_i^+\|_2 < \|x_i - y_i^-\|_2}}{N}, \tag{4}$$

where x_i, y_i^+, and y_i^+ represent the anchor text, positive code snippet and negative code snippet representations respectively, while $\mathbf{1}_i$ is an indicator variable which is 1 if $i > 0$ and 0 otherwise).

Loss function We use the following triplet loss function: $\mathcal{L}_\phi(x_i, y_i^+, y_i^-) = \max\{\|x_i - y_i^+\|_2 - \|x_i - y_i^-\|_2 + 1, 0\}$, where x_i, y_i^+ and y_i^- represent the intent, positive code sample and negative code sample respectively. We use the default margin of 1. Ablations with different margins for hard and soft negatives don't lead to better performance.

4 Experimental Setup

4.1 Datasets

We conduct several experiments on four popular Python code retrieval datasets, namely CoNaLa, PyDocs, WebQuery and CodeSearchNet.

CoNaLa. The CoNaLa dataset (Yin et al. (2018)) has 2.4k training and 500 test examples with intents and corresponding code curated by human annotators. They also automatically mine 600k intent-snippet pairs from StackOverflow. Due to its size, we utilize this set of mined pairs as the main pre-training corpus for our experiments. Based on Xu et al. (2020) and our pilot studies (see table 7 in Appendix), we filter out the 100k most relevant NL-PL pairs (detailed in Appendix) to reduce noise in the dataset and achieve better performance.

PyDocs. The PyDocs dataset is curated from function signatures and corresponding documentation from Python's standard library API reference by Xu et al. (2020). The authors heuristically generate syntactically correct function calls from their specifications and queries from their documentation. Next, they resample the data to match the distribution of CoNaLa by retrieving k most relevant API NL-PL pairs based on CoNaLa NL and PL separately. They further use the frequencies of the retrieved NL-PL pairs to build the sampling distribution with a temperature parameter to balance between uniform sampling and matching the distribution of the CoNaLa data. In this work we use the data corresponding to both NL and PL based retrieval with the lowest temperature(=2) (most similar to CoNaLa) and set aside 365 queries and 416 corresponding documents as a test set while the remaining training and validation data has 9.7k NL-PL pairs.

WebQuery. The WebQuery test set is associated with the CoSQA corpus: a dataset of 20.6K human annotated query-code pairs curated by Huang et al. (2021). The NL queries in this dataset are "web queries" with a code searching intent as judged by human annotators while the code candidates are functions which are annotated for relevance to the query using the docstring, function header and body. WebQuery test set has 523 NL queries and 803 unique code candidates in it.

CodeSearchNet. The CodeSearchNet corpus Husain et al. (2019) is collected from the most popular open-source Github repositories for six programming languages including Python. The authors extract functions from the codebases automatically along with their respective documents using heuristic regular expressions; discarding functions with very short (or no) documentations. They truncate documentations to the first paragraph to make their length comparable to search queries, filter out functions shorter than 3 lines, functions containing the substring "test" and standard extension methods (e.g. `__str__` dunder method in Python) and near duplicates (using Allamanis (2019) and Lopes et al. (2017)). We utilize the Python subset of the test set which has 21.5k queries and 22k documents. Since the "queries" are function documentation written by the authors of the codes, they have a very different distribution than regular search queries.

4.2 EXPERIMENTS

Training. We train the transformer-based models on both CoNaLa and PyDocs to compare the generalizability based on the training dataset (Tab. 3) and proceed with CoNaLa mined pairs as the primary dataset based on the results. Additionally, we also compare the effect of using the top 100k most relevant NL-PL pairs instead of the whole dataset (Tab. 7). We train the transformer models on the CoNaLa 100k data with regular fine-tuning, dynamic negative sampling-based fine-tuning (DNS), and our AST-guided curriculum and show the results over, all 4 test sets described in the previous section, in Tab. 4 along with other modeling baselines, explained in the following section.
Model Selection. We use a retrieval style validation with 14k queries and 18.3k code candidates and the recall@5 metric to pick the best model.
Testing. We test all the models, except UniXcoder on all 4 datasets. UniXcoder is not tested on CodeSearchNet as it is part of its pre-training corpus. The summary statistics of each test set are shown in Table 6. We average the metrics over all test sets to report the generalization performance and use the average performance barring CoNaLa to report out-of-domain (OOD) generalization.
Metrics. We use Normalized Discounted Cumulative Gain (NDCG), recall@k (for k = 5, 10), and Mean Reciprocal Rank (MRR) as the metrics for evaluation.

Model	Trained on	MRR	NDCG	Recall@5	Recall@10
CodeBERT	CoNaLa	58	68.13	66.62	77.52
GraphCodeBERT	CoNaLa	50.65	61.19	61.58	72.62
UniXcoder	CoNaLa	**63.8**	**73.37**	**72.92**	**83.33**
CodeBERT	PyDocs	49.64	60.15	54.17	61.07
GraphCodeBERT	PyDocs	52.42	62.94	58.82	67.18
UniXcoder	PyDocs	49.22	60.42	54.83	63.42

Table 3: Effect of training dataset on the generalizability of models.

Baselines. We train our baselines in a siamese configuration similar to Husain et al. (2019) but use the same architecture for both the text & code encoders. We train them using a binary cross entropy loss function objective where x and y are the code snippet and intent representation, whereas $\mathbf{1}_n$ is an indicator variable which is 1 if intent and snippet are related to each other and 0 otherwise.

$$\mathcal{L}_\phi(x, y) = -[\mathbf{1}_n \cdot \log(\frac{1}{1 + e^{-x^T y}}) + (1 - \mathbf{1}_n) \cdot \log(\frac{e^{-x^T y}}{1 + e^{-x^T y}})], \tag{5}$$

We obtain binary classification data of roughly 950k NL-PL pairs for training and 237.6k for validation with a roughly even class distribution from the CoNaLa data by random sampling of negatives.
• **Neural Bag of Words (n-BOW)**: Here, we treat the intent and code snippet as a bag of words and computes their representation via a 1-D mean pool over all the tokens. We use CodeBERT tokenizer to obtain the tokens, and the token-level embeddings are initialized from the 768 dimensional embedding layer of CodeBERT.
• **CNN Baseline:** For the CNN baseline, we use three successive 1-D convolutions with a kernel of width 16. We use padding, residual connections and dropout of 0.2 at each layer. Finally, we pool across the sequence by using an attention-like weighted sum. The architecture closely follows the CNN baseline proposed in Husain et al. (2019).

Model	MRR (Δ)	NDCG (Δ)	Recall@5 (Δ)	Recall@10 (Δ)
n-BOW	6.15	19.21	7.05	10.17
CNN	2.96	16.76	2.6	5.19
RNN	6.7	21.42	8.21	13.54
CodeBERT (zero shot)	6.24	19.37	7.41	9.94
CodeBERT	58	68.13	66.62	77.52
CodeBERT + DNS	60.95 (+2.95)	70.51 (+2.38)	69.88 (+3.26)	79.43 (+1.91)
CodeBERT + AST	63.04 (+5.04)	72.21 (+4.08)	72.48 (+5.86)	82.44 (+4.92)
CodeBERT + AST (hard neg)	30.02	43.97	36.08	45.6
GraphCodeBERT (zero shot)	16.45	28.04	19.65	22.88
GraphCodeBERT	50.65	61.19	61.58	72.62
GraphCodeBERT + DNS	52.97 (+2.32)	63.88 (+2.69)	62.34 (+0.76)	73.79 (+1.17)
GraphCodeBERT + AST	56.15 (+5.50)	66.39 (+5.20)	65.09 (+3.51)	75.93 (+3.31)
GraphCodeBERT + AST (hard neg)	33.79	47.09	41.43	52.59
UniXcoder (zero shot)	44.39	56.51	50.59	57.21
UniXcoder	63.8	73.37	72.92	83.33
UniXcoder + DNS	64.2 (+0.40)	73.68 (+0.31)	74.66 (+1.74)	83.39 (+0.06)
UniXcoder + AST	**65.13** (+1.33)	**74.58** (+1.21)	**74.83** (+1.91)	**84.68** (+1.35)
UniXcoder + AST (hard neg)	50.9	62.94	61.44	72.78

Table 4: The averaged metrics over all 4 test sets: CoNaLa, PyDocs, CodeSearchNet, WebQuery (CodeSearchNet is excluded for UniXcoder) for all models and baselines when trained on CoNaLa. The Δ represents improvements in each model when using DNS & AST compared to the base model

	MRR		NDCG		Recall@5		Recall@10	
Model	IID	OOD	IID	OOD	IID	OOD	IID	OOD
CodeBERT	54.7	59.1	65.33	69.06	62.2	68.09	77.4	77.56
CodeBERT+DNS	54.96	62.95	65.8	72.08	66.2	71.1	79.8	79.3
CodeBERT+AST	**56.62**	**65.17**	**67.08**	**73.92**	**68.0**	**73.97**	**82.4**	**82.46**
GraphCodeBERT	57.4	48.4	67.78	59.94	69.8	58.84	83.2	69.1
GraphCodeBERT+DNS	**59.28**	50.87	**69.26**	62.08	67.6	60.59	80.8	71.45
GraphCodeBERT+AST	58.28	**55.44**	68.37	**65.73**	**68.4**	**63.99**	**83.6**	**73.38**
UniXcoder	59.82	65.79	69.53	75.3	69.6	74.59	83.0	83.49
UniXcoder+DNS	59.13	66.74	69.02	76.01	**72.4**	75.79	84.2	82.99
UniXcoder+AST	**60.21**	**67.6**	**70.06**	**76.83**	**72.4**	**76.05**	**84.8**	**84.61**

Table 5: Breakdown of in-distribution performance on CoNaLa and out-of-distribution performance on other datasets for CodeBERT, GraphCodeBERT, and UniXcoder for all 3 training variants.

• **RNN Baseline:** We use a 2-layered Bi-LSTM architecture with a dropout of 0.2. Similar to Husain et al. (2019), we use a final attention-like weighted sum layer across all hidden states to calculate the final representation.

• **Dynamic Negative Sampling (DNS):** We propose a strong baseline loosely based on the STAR and ADORE algorithms Zhan et al. (2021). STAR uses a mixture of static hard negatives and randomly sampled soft negatives to train both the query and document encoder. ADORE runs a retrieval over all document embeddings for each mini-batch to dynamically find hard negatives, while freezing the document encoder and only updating the query encoder. Zhan et al. (2021) sequentially train the query and document encoder with STAR and then further train the query encoder only with ADORE. This procedure doesn't translate directly to our setting since we use the transformer models as universal encoders for both queries (NL) and documents (code/PL) simultaneously. We strike a balance by performing an ADORE-like retrieval for each mini-batch of queries and documents but on the limited set of documents present in a mini-batch instead of the whole corpus and update the combined document and query encoder at each step (Fig. 3 in the Appendix). We find this approach

to be stable during training and see significant improvements in performance for CodeBERT and GraphCodeBERT and modest improvement for UniXcoder as outlined in section 5.

5 RESULTS AND DISCUSSION

Effect of Training Data on Generalization. Before measuring the effect of our approach on the generalization capability of models, we measure the effect of the training data on generalization by comparing the performance of the models when trained on PyDocs and CoNaLa. The results in Tab. 3 show that training on CoNaLa leads to significantly better generalization for CodeBERT and UniXcoder; and better performance for recall for GraphCodeBERT but slightly worse performance for MRR and NDCG. Therefore we use CoNaLa as the training data for all subsequent experiments.
Effect of AST-guided Curriculum on Generalization. We compare the performance of the baselines, and the three transformer models in zero-shot, regular contrastive learning, dynamic negative sampling (DNS), and AST-guided curriculum learning settings (AST) (results in Tab. 4). We observe significant improvement in the performance over the regular training for DNS and AST for CodeBERT (+2.625 & +4.975 over all metrics) and GraphCodeBERT (+1.735 & +4.38). For UniXcoder we observe comparably less improvement (+0.6275 and +1.45 for DNS and AST), potentially because it already incorporates some AST information during pre-training. Interestingly, GraphCodeBERT performs worse than CodeBERT over generalization, due to poor performance over the CodeSearchNet corpus as shown in the dataset-wise breakdown in Table 8 in the Appendix. Additionally, we observe that the transformer models dominate the simpler models even in zero-shot settings, especially UniXcoder which has a +39.708 gain over the RNN baseline & +30.42 over zero-shot GraphCodeBERT. We also investigate the importance of curriculum design, with experiments that use only the hard negatives ("hard neg" in Table 4) and a naive curricula (Table C.3 in Appendix). The results showcase the effectiveness of the mastering-rate based curriculum in making the most of the hard negatives and stabilizing the training.
Analysis of In-Distribution & Out-of-Distribution Performance. Table 5 shows the in-domain and out-of-domain performance of each model (CodeBERT, GraphCodeBERT, and UniXcoder) for regular training, DNS and AST-guided curriculum-based training. The metrics indicate that the use of AST-guided curricula leads to significant performance gains in out-of-domain generalization over the base training as well as DNS while achieving similar performance on in-domain data for UniXcoder and GraphCodeBERT and significant improvement for CodeBERT.
Qualitative Analysis. Our structure-aware training curriculum leads to improvements over the error classes identified in section 3.1. We show some motivating examples in Tab. 9 (in Appendix). For the first example, all the top 5 retrieved code snippets for the AST model invoke the correct function call `extend` compared to the base model. In the second example, three of the top five retrieved code snippets for the AST model have a tuple of tuples or a list of tuples data structure, while the baseline model retrieval results feature 1D lists instead. Finally, for the third example, the highest ranked candidate gets all 3 function arguments correct.

We also perform analogy testing (of the form a:b::c:? for each rule) over the code representations to quantify their sensitivity to the perturbation patterns introduced by our AST-guided hard negatives. We observe that our approach leads to better performance overall for CodeBERT and GraphCodeBERT, with equivalent or slightly better performance over each rule pattern. For UniXcoder we observe a slight drop in performance over certain rule types (details in Appendix Section C.6).

6 CONCLUSION

For neural code search, state-of-the-art Transformers-based pretrained encoders make certain common mistakes that indicate limited understanding in code syntax and semantics. We notice three broad error categories such as retrieving code with wrong data-type, method with incorrect signature or incorrect arguments or incorrect branching for CodeBERT, GraphCodeBERT and UniXCoder. To learn more efficient representation during fine-tuning, we propose a structure-aware hard negative sampling through AST perturbation alongwith a mastering-rate based curriculum, where our AST perturbation rules are motivated by above error categories and generic code constructs. Our experiments show significant improvement on above three models on four code retrieval datasets (in IID and OOD settings). Interestingly, our method shows improvement even for UniXCoder which is exposed to underlying AST structure of the code snippets during pre-training.

REFERENCES

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

Kian Ahrabian, Aarash Feizi, Yasmin Salehi, William L. Hamilton, and Avishek Joey Bose. Structure aware negative sampling in knowledge graphs. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 6093–6101, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.492. URL https://www.aclweb.org/anthology/2020.emnlp-main.492.

Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code. *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2019.

Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. Self-supervised bug detection and repair. In *NeurIPS*, 2021.

Guanyi Chu, Xiao Wang, Chuan Shi, and Xunqiang Jiang. Cuco: Graph representation with curriculum contrastive learning. In Zhi-Hua Zhou (ed.), *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, pp. 2300–2306. ijcai.org, 2021. doi: 10.24963/ijcai.2021/317. URL https://doi.org/10.24963/ijcai.2021/317.

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

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

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*, 2022.

Elad Hoffer and Nir Ailon. Deep metric learning using triplet network. In *SIMBAD*, 2015.

Junjie Huang, Duyu Tang, Linjun Shou, Ming Gong, Ke Xu, Daxin Jiang, Ming Zhou, and Nan Duan. CoSQA: 20,000+ web queries for code search and question answering. 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)*, pp. 5690–5700, Online, August 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.442. URL https://aclanthology.org/2021.acl-long.442.

Hamel Husain, Hongqi Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *ArXiv*, abs/1909.09436, 2019.

Xue Jiang, Zhuoran Zheng, Chen Lyu, Liang Li, and Lei Lyu. Treebert: A tree-based pre-trained model for programming language. In *Uncertainty in Artificial Intelligence*, pp. 54–63. PMLR, 2021.

René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pp. 654–665, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330565. doi: 10.1145/2635868.2635929. URL https://doi.org/10.1145/2635868.2635929.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Pre-trained contextual embedding of source code. 2019.

Rafael-Michael Karampatsis and Charles Sutton. How often do single-statement bugs occur?: The manysstubs4j dataset. *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020.

Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom, Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de, Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey, Cherepanov, James Molloy, Daniel Jaymin Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de, Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *ArXiv*, abs/2203.07814, 2022.

Rong Liang, Yujie Lu, Zhen Huang, Tiehua Zhang, and Yuze Liu. Astbert: Enabling language model for code understanding with abstract syntax tree. *arXiv preprint arXiv:2201.07984*, 2022.

Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. Déjàvu: A map of code duplicates on github. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017. doi: 10.1145/3133908. URL https://doi.org/10.1145/3133908.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation. *ArXiv*, abs/2102.04664, 2021.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *ArXiv*, abs/1609.07843, 2017.

Jibesh Patra and Michael Pradel. Semantic bug seeding: A learning-based approach for creating realistic bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, pp. 906–918, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385626. doi: 10.1145/3468264.3468623. URL https://doi.org/10.1145/3468264.3468623.

Jibesh Patra and Michael Pradel. Nalin: learning from runtime behavior to find name-value inconsistencies in jupyter notebooks. *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pp. 1469–1481, 2022.

Michael Pradel and Koushik Sen. Deepbugs: A learning approach to name-based bug detection. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018. doi: 10.1145/3276517. URL https://doi.org/10.1145/3276517.

Joshua David Robinson, Ching-Yao Chuang, Suvrit Sra, and Stefanie Jegelka. Contrastive learning with hard negative samples. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL https://openreview.net/forum?id=CR1XOQ0UTh-.

Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. Intellicode compose: code generation using transformer. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020.

Xin Wang, Yasheng Wang, Fei Mi, Pingyi Zhou, Yao Wan, Xiao Liu, Li Li, Hao Wu, Jin Liu, and Xin Jiang. Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation. *arXiv preprint arXiv:2108.04556*, 2021a.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021b.

Ming Wen, Yepang Liu, Rongxin Wu, Xuan Xie, Shing-Chi Cheung, and Zhendong Su. Exposing library api misuses via mutation analysis. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 866–877, 2019. doi: 10.1109/ICSE.2019.00093.

Lucas Willems, Salem Lahlou, and Yoshua Bengio. Mastering rate based curriculum learning. *ArXiv*, abs/2008.06456, 2020.

Frank F. Xu, Zhengbao Jiang, Pengcheng Yin, Bogdan Vasilescu, and Graham Neubig. Incorporating external knowledge through pre-training for natural language to code generation. In *ACL*, 2020.

Hong Xuan, Abby Stylianou, Xiaotong Liu, and Robert Pless. Hard negative examples are hard, but useful. In *ECCV*, 2020.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. Learning to mine aligned code and natural language pairs from stack overflow. In *International Conference on Mining Software Repositories*, MSR, pp. 476–486. ACM, 2018. doi: https://doi.org/10.1145/3196398.3196408.

Jingtao Zhan, Jiaxin Mao, Yiqun Liu, Jiafeng Guo, M. Zhang, and Shaoping Ma. Optimizing dense retrieval model training with hard negatives. *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2021.

A METHOD

A.1 ALGORITHM OUTLINE FOR AST PERTURBATION

Algorithm 1: Pseudocode for AST guided code perturbation

Data: $\rho, \mathcal{R}$ /* ρ is program snippet, $\mathcal{R}$ is set of rules */
Result: $\mathcal{P} = \{\rho'_1, \ldots \rho'_n\}$ /* ρ'_i is i^{th} corrupted program snippet */
$\mathcal{T} \leftarrow$ `parseAST`(ρ);
$\mathcal{S} \leftarrow \emptyset$; /* Traverse AST & collect applicable rule sites */
$\eta \leftarrow$ `getRoot`$(\mathcal{T})$;
$\mathcal{W} \leftarrow \{\eta\}$;
while $\mathcal{W} \neq \emptyset$ **do**
  $\eta \leftarrow \mathcal{W}$.`pop()`;
  for $r \in \mathcal{R}$ **do**
    if *isValidSite*(η, r) **then**
      $\mathcal{S}$.`push`$(\langle \eta, r \rangle)$; /* Collect valid candidate rule sites without modifying AST */
  for $n \in$ *succ*(η) **do**
    $\mathcal{W}$.`push(n)`;
$\mathcal{P} \leftarrow \emptyset$;
for $\langle \eta, r \rangle \in \mathcal{S}$ **do**
  $\mathcal{T}_c \leftarrow$ `copy`$(\mathcal{T})$; /* Create a copy of the AST to modify later into a corrupted program snippet */
  $\mathcal{T}_c \leftarrow$ `applyRule`$(\mathcal{T}_c, \eta, r)$; /* Apply rule on valid site node and transform AST */
  $\rho' \leftarrow$ `unparseAST`$(\mathcal{T}_c)$; /* Regain program snippet from transformed AST */
  $\mathcal{P}$.`push`(ρ'); /* Collect set of corrupted program snippets */

We discuss the pseudo-code of our AST perturbation in algorithm 1. To detect valid sites for each rule application we use checks on the type of the node (each node type has a dedicated Python class representation). In fact Python's `ast` module provides `visit` functions for each type of node (for e.g. `visit_List` for nodes of `List` type). The function `visit_Type` is called whenever a node of type `Type` is visited, and we override these functions to keep a track of certain nodes which are sites for valid rule applications. An additional detail that might not be apparent from the pseudo-code is that we successively apply a rule on all of its valid sites at a time, but we apply at most one rule at a time. For e.g. while applying rule 5 on `if x == True and y == False`, we substitute all occurrences of `True` with `False` and `False` with `True`, to obtain `if x == False and y == True`. Our procedure is guaranteed to give syntactically correct corrupted

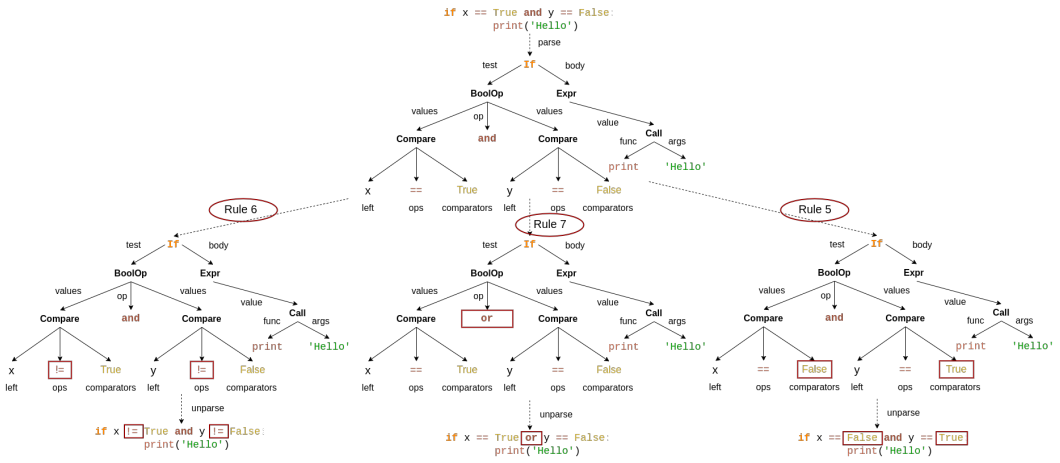

Figure 2: **AST perturbation in action:** For the given code snippet, rule 5, 6, 7 & 9 are applicable, leading to 4 AST-based hard negative candidates. Rule 5 flips the leaf nodes corresponding to the named constants "True" & "False", while rule 6 replaces boolean "and" operator leaf node with an "or" leaf node, rule 7 flips the "==" leaf nodes to "!=" leaf nodes and finally rule 9 replaces the code snippet with the body of the if statement "print('Hello')". Rule 9 is not shown due to a lack of space.

codes as output, as the unparse module fails to recover the code string if the transformed AST is invalid. Now we will briefly cover the approach we use to score and rank candidate function calls for the function call substitution rule (rule 1 in 2). Fig. 2 shows our algorithm in action for `if x != True and y != False: print("Hello").`

A.2 Function similarity scoring for function call substitution (rule 1)

For a target, function call $\mathcal{F}_i$ to be substituted by a target function call $\mathcal{F}_j$ we compute the score s_{ij} as the sum of the function name match score s^n_{ij} and the function signature match score s^s_{ij} ($s_{ij} = s^n_{ij} + s^s_{ij}$). We compute s^n_{ij} using the `token_sort_ratio` measure, implemented by the fuzzwuzzy[4] python package, between the function strings after replacing underscores with spaces and normalizing it to be between 0 to 1, instead of 0 to 100. s^s_{ij} also has two components: a return type match score s^{ret}_{ij} and a parameter match score s^p_{ij} and is compute as $s^p_{ij} = s^{ret}_{ij} + s^p_{ij}$. s^{ret}_{ij} is 1 if both function calls have the same return type and 0 otherwise, while s^p_{ij} attempts to match the parameter kinds (positional argument vs keyword argument) and default values, from left to right and normalizes it by the maximum possible score. We do not use the data type information for function arguments, as it is not available for several of the function calls (Python doesn't require explicit data types in function specifications and data type information can only be given as optional hints or annotations). Each component score in s_{ij}, varies between 0 to 1, leading to s_{ij} itself ranging from 0 to 3 (as $s_{ij} = s^n_{ij} + s^{ret}_i j + s^p_{ij}$). We recognize that prior work like Patra & Pradel (2022) has applied learned semantic embeddings to match code entities, but we avoid doing it for function names here to enable faster matching over larger candidate sets ($\approx$ 5.9k candidates). Efficient ways to incorporate learned embeddings or even the current model weights to match candidate functions could be a promising extension of our work.

B Dataset Statistics

We show number of unique queries and documents (code snippets) alongwith representative examples for each of the four test sets in Table 6. Notably, PyDocs, WebQuery and CodeSearchNet all vary from CoNaLa in the way queries are expressed. WebQuery and CodeSearchNet contain larger code snippets compared to PyDocs and CoNaLa.

[4]https://pypi.org/project/fuzzywuzzy/

Dataset	#Queries	#Docs	Intent	Code Snippet
CoNaLa	365	490	How can I send a signal from a python program?	os.kill(os.getpid(), signal.SIGUSR1)
PyDocs	365	416	Return the current collection counts as a tuple of (count0, count1, count2).	gc.get_count()
WebQuery	523	803	python git get latest commit	def latest_commit(self) ->git.Commit: """":return: latest commit :rtype: git.Commit object""" latest_commit: git.Commit = self.repo.head.commit LOGGER.debug('latest commit: %s', latest_commit) return latest_commit

CodeSearchNet	21504	22176	str->list Convert XML to URL List. From Biligrab	def sina_xml_to_url_list(xml_data):
				"""str->list Convert XML to URL List. From Biligrab."""
				rawurl = []
				dom = parseString(xml_data)
				for node in dom.getElementsByTagName('durl'):
				url = node.getElementsByTagName('url')[0]
				rawurl.append(url.childNodes[0].data)
				return rawurl

Table 6: Statistics of each dataset: CoNaLa, PyDocs, WebQuery, CodeSearchNet.

C Experimental Details

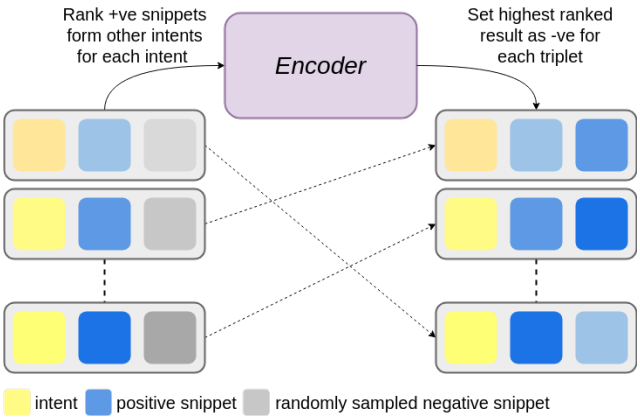

Figure 3: We create a feedback loop in the model training by using the current model weights to pair each intent-snippet pair with the closest snippet from another intent. These create the hardest negatives at a batch level, which are expected to be harder than randomly sampled negatives.

C.1 Hyper Parameter Variations

We explore some variations of the various hyper-parameters associated with the mastering-rate curriculum-learning algorithm. Figure 4 shows various variations of the β (used for the sampling in 3.1) and p (used in equation 2 for curriculum learning) for UniXcoder. We observe that increasing p generally leads to better performance, which indicates that hard negative attention needs to be sensitive to a drop in the mastering rate/accuracy of the soft negative learning task. Additionally, we see that for very low βs (almost uniform distribution over hard negatives) lead to a better performance with lower values of p, peaking a $p = 2$. This intuitively makes sense, as low βs correspond to softer hard negatives, reducing the gap between hard and soft negative learning tasks. However similar variations with CodeBERT & GraphCodeBERT indicated $p = 2$ and $\beta = 0.01$ to be overall better. We perform a large parameter sweep over various values of warmup steps (number of steps for which only soft negatives are used). For UniXcoder and CodeBERT we use a batch size of 48, which leads to 5000 steps per epoch for CoNaLa-100k (80:20 train-validation split and 3 negative samples per NL-PL pair), while for GraphCodeBERT a batch size of 32 was used, leading to 7500 steps per epoch, so we investigate variations in warmup steps in increments of 1000 for UniXcoder (Fig. 7) and CodeBERT (Fig. 5) and for increments of 2500 for GraphCodeBERT (Fig. 6). These

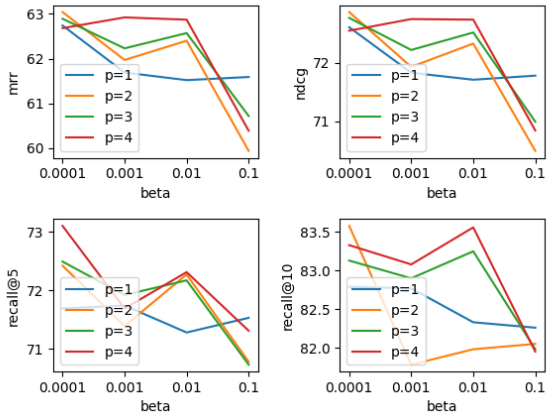

Figure 4: Hyperparameter search over the p and β for UniXcoder

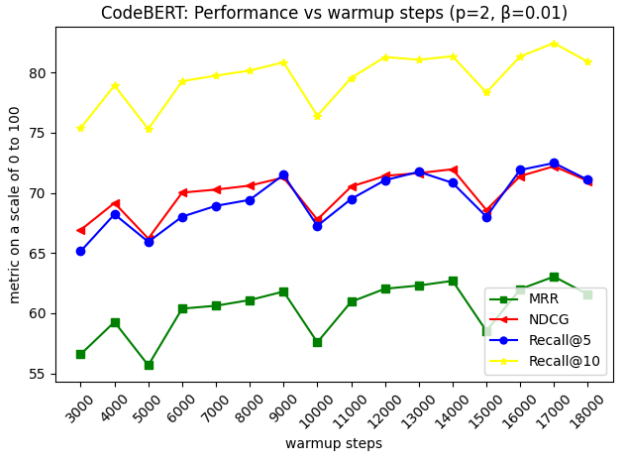

Figure 5: Effect of warmup steps on performance of CodeBERT over all 4 metrics, with $p = 2$ & $\beta = 0.01$

experiments indicate a general upward trend in performance, with a lot of fluctuations between consecutive points. For UniXcoder the upward trend is a lot weaker, which leads to the fluctuations being more significant overall. We also observe that varying the warmup steps seems to have more impact on the performance than p and β. We find that $p = 2, \beta = 0.01$ & 17k warmup steps work best for CodeBERT, while $p = 2, \beta = 0.0001$ & 13k warmup steps work best for UniXcoder and $p = 2, \beta = 0.01$ & 12.5k warmup steps work best for GraphCodeBERT. For all experiments we used $\epsilon = 0.8$ and $\delta = 0.5$. Future work would also examine the effect of these parameters on the overall performance.

C.2 Filtering CoNaLa Corpus

As mentioned in the Experiments section, we use 100k most relevant pairs of CoNaLa for finetuning, as it achieves comparable or better performance on CoNaLa test set. Details are shown in Table 7.

C.3 Effect of curriculum

We examine the impact of the curriculum design by trying some simple variations, like using only soft negatives ("soft neg"), using only hard negatives ("hard neg"), using a naive or random curricu-

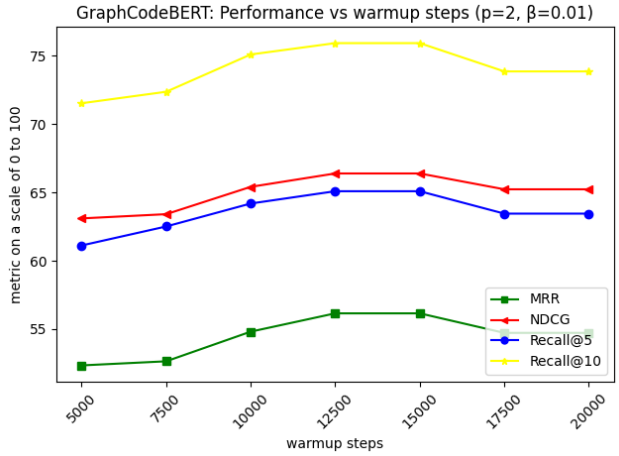

Figure 6: Effect of warmup steps on performance of GraphCodeBERT over all 4 metrics, with $p = 2$ & $\beta = 0.01$

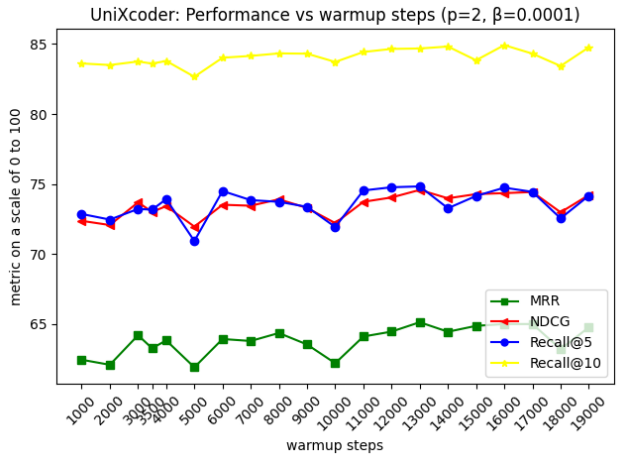

Figure 7: Effect of warmup steps on performance of UniXcoder over all 4 metrics, with $p = 2$ & $\beta = 0.0001$

Model	MRR	NDCG	Recall@5	Recall@10
CodeBERT	62.2	**78**	51.9	63.4
CodeBERT 100k	**62.2**	77.4	**54.7**	**65.3**
GraphCodeBERT	66.2	79.2	57	67.3
GraphCodeBERT 100k	**69.8**	**83.2**	**57.4**	**67.8**
UniXcoder	69.2	83	59.8	69.5
UniXcoder 100k	**69.6**	**83**	**59.8**	**69.5**

Table 7: Pilot study comparing the effect of training on the entire CoNaLa mined pairs dataset (roughly 600k NL-PL pairs) vs training on the 100k most "relevant" pairs based on the "prob" score. Using these 100k NL-PL pairs leads to similar or better performance in $\frac{1}{6}^{th}$ the training time.

lum ("rand curr") where we sample soft or hard negative instances with equal probability and our mastering rate curriculum ("MR curr"). The results are outlined in Table C.3. We see that using hard negatives only leads to the worst performance while just introducing some soft negatives through

Model	Curriculum Type	MRR	NDCG	Recall@5	Recall@10
CodeBERT	soft neg	58	68.13	66.62	77.52
CodeBERT	hard neg	30.02	43.97	36.08	45.6
CodeBERT	rand curr	57.18	67.5	66.25	76.24
CodeBERT	MR curr	**63.04**	**72.21**	**72.48**	**82.44**
GraphCodeBERT	soft neg	50.65	61.19	61.58	72.62
GraphCodeBERT	hard neg	33.79	47.09	41.43	52.59
GraphCodeBERT	rand curr	50.35	61.58	59.84	70.6
GraphCodeBERT	MR curr	**56.15**	**66.39**	**65.09**	**75.93**
UniXcoder	soft neg	63.8	73.37	72.92	83.33
UniXcoder	hard neg	50.9	62.94	61.44	72.78
UniXcoder	rand curr	61.12	71.29	72.07	82.26
UniXcoder	MR curr	**65.13**	**74.58**	**74.83**	**84.68**

the random curriculum greatly improves the performance, but still doesn't do as well as just using soft negatives. This shows how challenging it is to design a curriculum like the mastering rate based curriculum used here to make the most of the hard negatives, and achieve better performance than just using soft negatives.

C.4 Dataset-wise performance breakdown

	CoNaLa				PyDocs				WebQuery				CodeSearchNet			
Model	MRR	NDCG	R@5	R@10	MRR	NDCG	R@5	R@10	MRR	NDCG	R@5	R@10	MRR	NDCG	R@5	R@10
n-BOW	6.19	20.76	8	9.6	**16.21**	**30.55**	**18.51**	**27.88**	1.51	16.77	0.96	2.29	**0.67**	8.78	**0.72**	**0.92**
CNN	6.05	21.88	5.4	10.4	3.23	17.46	3.12	6.49	2.43	19.01	1.82	3.73	0.12	8.67	0.06	0.13
RNN	**13.02**	**29.28**	**16.2**	**25.6**	6.73	21.91	8.41	14.42	**6.49**	**24.42**	**7.74**	**13.29**	0.55	**10.06**	0.49	0.86
CB (zero shot)	2.76	16.75	3	5	6.48	20.37	8.65	12.26	1.62	16.97	1.24	3.35	14.1	23.41	16.76	19.14
CB	54.7	65.33	62.2	77.4	64.33	72.23	74.52	83.89	42.65	58.91	52.01	66.63	70.32	76.04	77.74	82.16
CB+DNS	54.96	65.8	66.2	79.8	69.76	76.39	78.37	86.3	43.18	59.23	52.77	66.06	75.91	80.61	82.16	85.55
CB+AST	56.62	**67.08**	**68**	**82.4**	**71.09**	**77.47**	**79.81**	**87.98**	**46.66**	**62.05**	**57.65**	**71.61**	**77.78**	**82.25**	**84.46**	**87.79**
CB+AST (hard neg)	21.2	36.64	27	37.8	18.34	32.48	23.56	32.45	26.15	44.71	32.41	45.7	54.38	62.06	61.34	66.46
CB+AST (rand curr)	51.43	63.04	61.6	76.6	60.8	69.11	71.63	79.09	41.97	58.38	51.43	65.77	70.12	75.8	77.31	81.58
GCB (zero shot)	9.89	23.72	12	17.2	53.93	62.86	64.66	70.91	1.88	16.98	1.91	3.25	0.12	8.6	0.03	0.14
GCB	57.4	67.78	69.8	83.2	67.41	74.69	78.85	86.78	43.84	59.92	54.49	69.5	33.96	45.2	43.18	51.01
GCB+DNS	**59.28**	**69.26**	67.6	80.8	70.44	77.08	79.09	88.7	44.45	60.55	55.35	70.08	37.72	48.62	47.32	55.57
GCB+AST	58.28	68.37	**68.4**	**83.6**	**78.47**	**83.32**	**85.58**	**91.11**	**47.04**	**62.49**	**55.64**	**70.46**	**40.82**	**51.37**	**50.75**	**58.57**
GCB+AST (hard neg)	30.77	45.17	37.8	49.8	51.1	61.22	62.26	75.72	32.79	50.5	39.87	53.63	20.49	31.47	25.81	31.2
GCB+AST (rand curr)	56.53	67.14	66	80.4	68.98	75.67	79.33	85.82	43	59.36	52.39	66.92	32.9	44.15	41.66	49.25
UX (zero shot)	20.7	34.9	24	30.4	83.02	86.94	90.87	93.99	29.46	47.7	36.9	47.23	-	-	-	-
UX	59.82	69.53	69.6	83	83.42	87.27	89.9	**94.71**	48.17	63.32	59.27	72.28	-	-	-	-
UX+DNS	59.13	69.02	72.4	84.2	84.77	**88.31**	91.35	94.47	48.71	63.71	60.23	71.51	-	-	-	-
UX+AST	**60.21**	**70.06**	**72.4**	**84.8**	**84.2**	87.87	**91.59**	93.99	**50.99**	**65.8**	**60.52**	**75.24**	-	-	-	-
UX+AST (hard neg)	39.42	52.92	48	63	70.86	77.3	82.21	88.22	42.43	58.61	54.11	67.11	-	-	-	-
UX+AST (rand curr)	57.4	67.88	68.4	82.2	78.22	83.02	87.02	90.87	47.75	62.96	60.8	73.71	-	-	-	-

Table 8: A breakdown of the performance of all models and baselines when trained on CoNaLa-100k over each test set (CoNaLa, PyDocs, WebQuery, CodeSearchNet). CB: CodeBERT, GCB: GraphCodeBERT, and UX: UniXcoder

The dataset-wise performance breakdown for all the 3 transformer models (CodeBERT, GraphCodeBERT, and UniXcoder) and all 3 training variants (regular triplet training, dynamic negative sampling (DNS), and our proposed AST-guided training curriculum (AST)) is shown in table 8.

C.5 Qualitative Analysis

We show the qualitative effect of our approach through examples in Table 9.

Model, Dataset & Query	Gold Candidates	AST Hits@5	Baseline Hits@5	Error Type
Model: CodeBERT **Dataset:** CoNaLa **Query:** Append elements of a set to a list in Python	a.extend(b) a.extend(list(b))	**a.extend(list(b))** c.extend(a) **a.extend(b)** list2.extend(list1) list1.extend(mylog)	list(set(source_list)) list(set(t)) my_list.append(l2) dict(((x, l.count(x)) for x in set(l))) list2.extend(list1)	Type-2
Model: CodeBERT **Dataset:** CoNaLa **Query:** How do I convert tuple of tuples to list in one line (pythonic)?	from functools import reduce reduce(lambda a, b: a + b, ((‘aa’,),(‘bb’,),(‘cc’,))) map(lambda a: a[0], ((‘aa’,),(‘bb’,),(‘cc’,)))	tuple(l) **map(lambda a: a[0],** ((‘aa’,), (‘bb’,), (‘cc’,)) zip(*[(‘a’, 1), (‘b’, 2), (‘c’, 3), (‘d’, 4)]) zip(*[(‘a’, 1), (‘b’, 2), (‘c’, 3), (‘d’, 4), (‘e’,)] [val for pair in zip(l1, l2) for val in pair]	tuple(l) “““, ”””.join(‘(’ + ‘, ’ .join(i) + ‘)’ for i in L) print([item for item in [1, 2, 3]]) list(t) “““,”””.join(l)	Type-1
Model: GraphCodeBERT **Dataset:** PyDocs **Query:** Asynchronous version of socket.getaddrinfo ( ) . With arguments “host”, “port”, “family”.	loop.getaddrinfo(host, port, family=0)	**loop.getaddrinfo(host, port, family=0)** dispatcher.create socket(family=socket.AF INET) socket.gethostbyname(hostname) socket.getfqdn() socket.getservbyname(servicename)	asyncio.open connection(port=None) dispatcher.create socket(family=socket.AF INET) asyncio.BaseProtocol asyncore.dispatcher_with_send async exit stack.push async exit(exit)	Type-2

Table 9: Qualitative Examples, that show how Type-1, Type-2 errors get corrected through our AST-guided curriculum.

C.6 Analogy tests for code representations

We perform analogy testing of the form a:b; c:? for each rule category, to gauge the effectiveness of the AST guided curriculum on the sensitivity of the code embeddings towards transformations based on the rules. To create the dataset we first apply the AST rules over the CoNaLa mined pairs train set and then sample 100 pairs of original code and transformed code for each of the 9 rule categories. Then we sample 200 examples from all possible 2 element combinations of the pairs to get an analogy test bed of 1800 examples of the form a:b; c:d with 200 samples per rule category. Some examples are shown in table C.6. During the sampling process, we also filter out code snippets smaller than 30 characters, to ensure example quality.

Model	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	Rule 8	Rule 9	Total Score
CodeBERT	82.5	96	93	94	96.5	96.5	96	86.5	92.5	92.611
CodeBERT+AST	**85**	**96.5**	**93.5**	**95.5**	**96.5**	**96.5**	**96**	**89**	**93**	**93.5**
GraphCodeBERT	86.5	94	90.5	94.5	96.5	96.5	96	85	92	92.389
GraphCodeBERT+AST	**87**	**96.5**	**90.5**	**94.5**	**96.5**	**96.5**	**96**	**90**	**95**	**93.611**
UniXcoder	**89.5**	96.5	94.5	96.5	96.5	96.5	96	**95**	**96.5**	**95.278**
UniXcoder+AST	89	**96.5**	**95.5**	**96.5**	**96.5**	**96.5**	**96**	91	95.5	94.778

Table 10: Analogy test results (recall@5) for the retrieval task of fetching d given $b + c - a$ from 1800 candidates from the CoNaLa dataset. Rule-wise and overall performance are shown, with 200 samples from each rule (transformation corresponding to the rule generates b from a and d from c). Euclidean distance is used here to score similarity between $b + c - a$ and candidate ds.

Model	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	Rule 8	Rule 9	Total Score
CodeBERT	82.5	96	93.5	94	96.5	96.5	96	86	93	92.667
CodeBERT+AST	**83.5**	**96.5**	**93.5**	**95**	**96.5**	**96.5**	**96**	**89.5**	**93.5**	**93.389**
GraphCodeBERT	86	94	90.5	94.5	96.5	96.5	96	85	93	92.444
GraphCodeBERT+AST	**87**	**96.5**	**90.5**	**94.5**	**96.5**	**96.5**	**96**	**90**	**95**	**93.611**
UniXcoder	89	96.5	94.5	96.5	96.5	96.5	96	**94**	**96.5**	**95.111**
UniXcoder+AST	**89**	**96.5**	**95.5**	**96.5**	**96.5**	**96.5**	**96**	91	96	94.833

Table 11: Analogy test results similar to Table 10 using cosine similarity instead of euclidean distance to rank ds for a given $b + c - a$

Rule	a	b	c	d
1	print('elements are not unique')	pprint('elements are not unique')	print('y = {0}'.format(y.value))	spring('y = {0}'..normalize(y.value))
2	[x for x in something_iterable if x != 'item']	{x for x in something_iterable if (x != 'item')}	[len(list(group)) for value, group in itertools.groupby(b_List) if value]	{len(list(group)) for (value, group) in itertools.groupby(b_List) if value}
5	date_ceased_to_act = models.DateField(blank=True, null=True)	date_ceased_to_act = models.DateField(blank=False, null=False)	print(df.to_csv(sep='\t', index=False))	print(df.to_csv(sep='\t', index=True))
6	def isPrime(n): if n < 2: pass	def isPrime(n): if (n >= 2): pass	import dill import pickle s = pickle.dumps(lambda x, y: x + y) f = pickle.loads(s) assert f(3, 4) == 7	import dill import pickle s = pickle.dumps((lambda x, y: (x + y))) f = pickle.loads(s) assert (f(3, 4) != 7)

Table 12: Some examples from the analogy test data. The columns "a" and "b" are the examples shown to indicate the pattern being, applied, while column "c" is the input and column "d" is the target snippet to be retrieved (a:b::c:d). We chose (a, b) & (c, d) such that the same rule is applied to get b from a and c from d, which is shown in the "Rule" column.

To evaluate the performance we measure all pairs' euclidean distance between the embeddings $c + b - a$ and d and rank all possible candidates in the 1800 sample test set for each (a, b, c) triple. We assign an analogy score of 1 to a sample if the correct d is among the top 5 retrieved candidates out of 1800 (similar to recall@5). The overall performance is just the mean over each sample and rule-wise performance is the mean over the samples involving transformations of a particular rule class.

We observe an improvement or similar performance in each rule category for all the models except UniXcoder where we see slightly worse performance for rules 1, 8, and 9. The highest performance drop is on rule 8 which substitutes a function call with the function's name as an identifier. This is a somewhat strange kind of error for a developer to make and we theorize that since UniXcoder has been pre-trained on CodeSearchNet data having function code and corresponding comments, it might not be sensitive to unnatural perturbations like this.