SYNC: A Structurally guided Hard Negative Curricula for Generalizable Neural Code Search

Atharva Naik[1], **Soumitra Das**[2], **Jyothi Vedurada**[3], **Somak Aditya**[4]

[1] Carnegie Mellon University, [2] Synopsys, [3] Department of CSE, IIT Hyderabad
[4] Department of CSE, IIT Kharagpur

atharvanaik2018@gmail.com, soumitradas1999@gmail.com, jyothiv@cse.iith.ac.in, saditya@cse.iitkgp.ac.in

Abstract

In neural code search, a Transformers-based pre-trained language model (such as CodeBERT) is used to embed both the query (NL) and the code snippet (PL) into a joint representation space; which is used to retrieve the relevant PLs satisfying the query. These models often make mistakes such as retrieving snippets with incorrect data types, and incorrect method names or signatures. The generalization ability beyond 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 novel contrastive learning technique (SYNC) that enables efficient finetuning of code LMs with soft and hard negatives, where the hard negatives are constructed using a set of structure-aware AST-based perturbations; targeted towards possible syntactic and semantic variations. Our method achieves significant improvements in retrieval performance for three code LMs (CodeBERT, GraphCodeBERT, UniXCoder) over four Python code retrieval datasets. We also open source our code for reproducibility[1].

1 Introduction

Learning dense representations for programming languages using NLP techniques has proven effective for various downstream tasks. Akin to the evolution of NLP models, contextualized Transformers-based representations such as CodeBERT (Feng et al., 2020), and GraphCodeBERT (Guo et al., 2020), universal cross-modal models such as UniXCoder (Guo et al., 2022) and generative models such as AlphaCode (Li et al., 2022b) have achieved state-of-the-art in public benchmarks. Unlike natural language tasks, (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 the Transformers-based code embedding models consistently make mistakes such as retrieving snippets with wrong data types (sets instead of lists), wrong method names or signatures and wrong arguments; indicating a loss of structural information. For example in a python code retrieval dataset, for the query "sorting a list of lists in python", the top retrieved snippet using CodeBERT is `sorted(list_of_strings,key=lambda s: s.split(',')[1])`, which sorts a list of *strings* instead (more examples in appendix).

To preserve such structural information, we adopt 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. We utilize a mix of random negative samples along with hard negatives (Xuan et al., 2020) generated using perturbations of the Abstract Syntax Tree (AST) for the positive NL-PL pair. To balance the hardness and the learning state of the model, we use a mastering rate-based curriculum approach (Willems et al., 2020) to sample a mix of soft and hard negatives, while hard negatives are further sampled using a parameterized distribution over model scores (Robinson et al., 2021). We observe 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 contrastive learning approach (SYNC) using an AST-based curriculum 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 UniX-

[1] https://github.com/atharva-naik/SYNC

Proceedings of the 13th International Joint Conference on Natural Language Processing and the 3rd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics (Volume 1: Long Papers), pages 83–103
November 1–4, 2023.

coder, which is exposed to ASTs in the pre-training stage. Specifically, our contributions are the following. We propose 1) a novel contrastive fine-tuning approach based on (mastering rate-based) curriculum to learn from both hard and soft negatives, where hard negatives are created through structure-aware AST perturbation rules. Through ablations, we observe proposed set of rules work better than existing ones (such as in DISCO). 2) Our comprehensive evaluation shows our approach achieves best OOD performance (with comparable or better ID results) for 3 SOTA models across 4 retrieval datasets; against varying curriculum and 3) strong contrastive learning baselines (DISCO, CodeRetriever). We evaluate the representation further through analogy testing & analyzing qualitative examples. 4) We will make the code available, including the contrastive baselines.

Observations. More specifically: 1) SYNC achieves best OOD performance for 3 models, 3 datasets across 4 metrics (Tab. 2); 2) several baseline variations and strong contrastive baselines, show an ID-OOD tradeoff where these methods reach comparable or better performance (as ours) for CoNaLa test set (NL/PL: 365/490), but failing to generalize for other 3 datasets (NL/PL: 365/416, 523/803, 21k/22k; Tabs. 9,12). 3) For CoNaLa examples (ID), where SYNC performs poorly (compared to CodeRetriever), CodeBLEU scores suggest the top retrieved codes are more lexically and structurally similar to the gold code snippet (§5). 4) We show the effect of curriculum by comparing against varying curricula (Tab. 10) and plotting the effect on validation recall as soft-to-hard negative sampling ratio changes for CodeBERT (Fig. 8). We also perform hyperparameter ablations, ablations over the chosen set of rules in Appendix.

2 Related Work

Neural Code Search. For neural code search, both encoder-only and encoder-decoder pretrained architectures (Kanade et al., 2019; Feng et al., 2020) perform well, with decoder-only models (Svyatkovskiy et al., 2020; Lu et al., 2021) being more successful for generative tasks (Guo et al., 2022). CodeBERT (Feng et al., 2020) is an encoder-only model trained using replaced token detection (RTD) and masked language modeling (MLM) objectives. GraphCodeBERT (Guo et al., 2020) incorporates dataflow information and additional pre-training objectives of edge prediction and node alignment. SYNCOBERT(Wang et al., 2021), AstBERT (Liang et al., 2022), and TreeBERT (Jiang et al., 2021) utilize abstract syntax tree (AST) representation during pre-training.

Contrastive Learning for Code Representations. Several contrastive learning methods have been proposed for code representation learning. Li et al. (2022a) proposed a contrastive objective combining unimodal and bimodal text-code losses to improve code search. Ding et al. (2021) used AST and dataflow-guided perturbations to create negative and positive examples for data augmentation and achieve improvements in code clone detection and vulnerability detection. Wang et al. (2022) proposed a hierarchical contrastive learning objective to improve code clustering, classification, and clone detection. Shi et al. (2022) propose soft data augmentation by masking random tokens to create positive samples to improve code search. These approaches do not address training stability issues encountered when using hard negatives in contrastive learning.

Contrastive Learning with Hard Negatives. Mining *hard* negatives for efficient contrastive learning is popular in image processing. Xuan et al. (2020) shows that hard negatives lead to unstable training behavior but can be useful with some simple fixes. Robinson et al. (2021) proposes a parametrized distribution that uses hardness scoring to sample suitable hard negatives that can maximally benefit the learning process. Hard negatives are difficult to learn 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. We employ a mastering rate-based curriculum (Willems et al., 2020) that guides the model (dynamically) to learn from both soft (randomly sampled) and hard AST-guided negatives while considering the model learning state.

Inducing Bugs in Software. Our AST-guided perturbations for generating hard negatives draw upon literature to induce bugs in software. Pradel and Sen (2018) use simple code transformations to create artificially induced bugs with swapped arguments, incorrect binary operators, and operands for JavaScript code. Allamanis et al. (2021) propose PyBugLab, a code rewriting-based approach for Python to induce bugs such as function argument swapping, operator substitution, etc. Ding

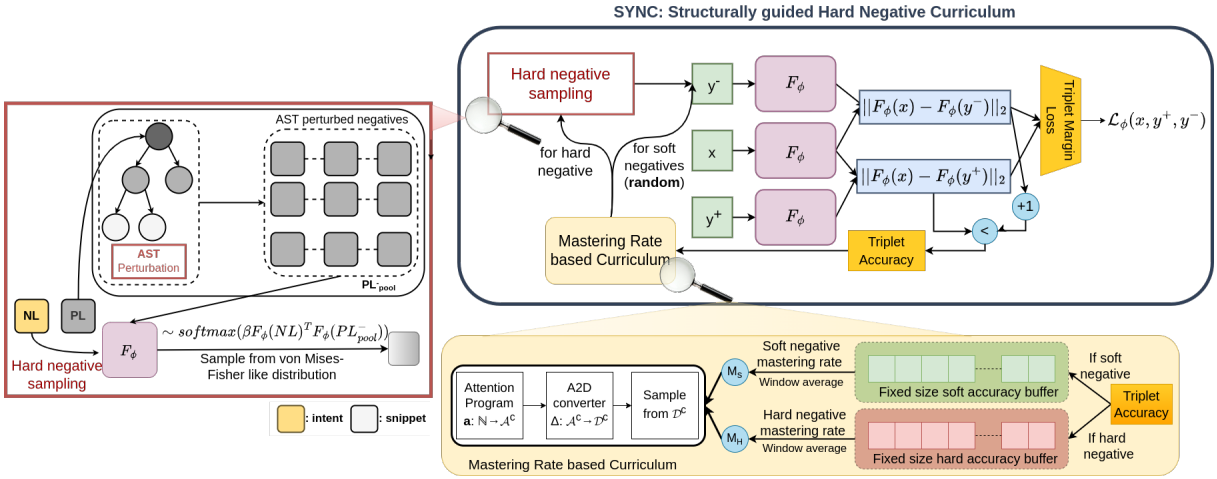

Figure 1: SYNC: Various stages of the proposed AST-guided curriculum.

et al. (2021) propose AST and dataflow-guided perturbation to create positive and negative examples. Their rules mainly target C++ and Java and are not directly applicable to Python.

3 Method

For code search, transformer-based pre-trained models are fine-tuned on annotated NL-PL pairs. Our method (Figure 1) targets improving the learned representation during this fine-tuning stage, by utilizing carefully synthesized hard negative samples. We follow the triplet network architecture (Hoffer and Ailon, 2015), where a textual query (x), a positive PL snippet (y^+), and a negative PL (y^-) snippet are sampled, and fed to a network individually. Generally, this network is a unified representation learner trained to embed text and code into a joint embedding space. We use transformer-based code LMs as encoders. After encoding the triplet, we use the contrastive loss to minimize the relative distances between the positive pair ($\langle x, y^+ \rangle$) and negative pair ($\langle x, y^- \rangle$). Negative sampling is the core of our method. We propose an improvement on the regular contrastive learning, by additionally fine-tuning the network with synthesized *hard* negatives (negative PL snippets) through well-specified AST-based perturbation rules. The set of rules is inspired by generic code constructs and the syntactic and semantic errors the SOTA models are observed to make. These perturbation rules are targeted to *infuse* the representation learners with targeted structural information. 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 Negative samples with AST perturbation

Most errors made by Transformers-based SOTA methods in code retrieval can be attributed to specific code structure violations with respect to the intent. We group such violations into three broad categories: **Type-1** (Data type mismatch): incorrect data types or data structures used (e.g. list comprehension instead of set comprehension) **Type-2** (Function call errors): Incorrect function is called or correct function is called with incorrect arguments. **Type-3** (Incorrect conditional checks): Incorrect branching, comparison ("==" instead of "!="), or logical operators (**and**, **or**). These errors motivate forcing transformer-based models to learn structural aspects, for which we use contrastive learning with synthesized hard negatives (Ding et al., 2021) and improve it using curriculum learning. 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 1. We carefully design these rules to capture potential syntactic and semantic violations, inspired by previous works (Allamanis et al., 2021; Wen et al., 2019). AST-perturbation rules replace one type of AST node with another producing negative examples with similar forms but different semantics. Next, we will explain how the rules directly address the above violations.

Rule-1 addresses Type-2 violations, 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 A.2). Rule-4 addresses violations 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 constants equal to the string lengths (e.g. `"hi"*n` to `2*n`). Finally rules 5, 6, 7, and 9 address violations 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.

Our AST perturbation Algorithm 1 (in Appendix) applies the appropriate rules to create a set of corrupted code candidates. We use Python's AST parser (Foundation, 2023a) to parse the code snippets into ASTs. Following the node-level substitutions, we unparse the AST using unparser (Foundation, 2023b) to get the corresponding code.

Sampling of Hard Negatives. After generating a set of candidate 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 $\frac{e^{\beta q^T c_i}}{\sum_i e^{\beta q^T c_i}}$ 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.

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 curriculum approach. Willems et al. (2020) defines 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 $d : N \rightarrow \mathcal{D}^{\mathcal{C}}$, 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* ($a : N \rightarrow \mathcal{A}^{\mathcal{C}}$) 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:

$$
\begin{aligned}
a_s^{(t)} &= (\delta \cdot (1 - \mathcal{M}_s^{(t)}) + (1 - \delta) \cdot \hat{\gamma}_s^{linreg}(t)) \\
&\quad \cdot (1 - \mathcal{M}_h^{(t)}) \\
a_h^{(t)} &= (\mathcal{M}_s^{(t)})^p \cdot (\delta \cdot (1 - \mathcal{M}_h^{(t)}) \\
&\quad + (1 - \delta) \cdot \hat{\gamma}_s^{linreg}(t))
\end{aligned}
$$

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. 1

	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)
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
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
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; exp_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 1: Representative AST perturbation rules and their corresponding grammars in Python's Abstract Syntax Description Language (ASDL). ASL has 4 inbuilt data types: identifier, int, string, constant. Full list in Tab. 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.

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

We compute the triplet accuracy $\mathcal{T}_a$ to estimate the mastering rates $\mathcal{M}_S^{(t)}$ and $\mathcal{M}_H^{(t)}$ using Eqn. 2:

$$\mathcal{T}_a = \frac{\sum_{i=0}^{N} \mathbf{1}_{\|x_i - y_i^+\|_2 < \|x_i - y_i^-\|_2}}{N}, \quad (2)$$

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 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 our experiments on four Python code retrieval datasets, namely CoNaLa, PyDocs, WebQuery and CodeSearchNet.

CoNaLa. CoNaLa (Yin et al., 2018) has 600k automatically mined intent-snippet pairs from StackOverflow, alongwith 2.9k manually annotated pairs (2.4k/500 train/test). Due to its size, we utilize CoNaLa as the pre-training corpus for our experiments. Based on Xu et al. (2020) and our pilot studies (see Tab. 6 in Appendix), we utilize a subset of 100k most relevant NL-PL pairs to reduce noise and achieve superior performance.

PyDocs. Xu et al. (2020) heuristically generated function calls from the specifications and queries from Python's standard library API documentation. They then resample the data to match the CoNaLa NL-PL pair distribution, using a weighted distribution (with temperature) to balance between uniform and CoNaLa-induced distribution. We use the data corresponding to the lowest temperature (2, i.e., most similar to CoNaLa) and set aside 365 queries and 416 documents as a test set. The remaining training& validation data have 9.7k NL-PL pairs.

WebQuery. The WebQuery test set is a part of the CoSQA corpus curated by Huang et al. (2021). The text queries in this dataset are "web queries" with a code search intent. The code candidates are functions that are annotated as relevant to the query

using the docstring, the function header, and the body. WebQuery test set has 523 NL queries and 803 unique code candidates.

CodeSearchNet. The CodeSearchNet corpus (Husain et al., 2019) is collected from open-source GitHub repositories for six programming languages including Python. Authors extracted function-documentation pairs from these code bases. We utilize the Python test set which has 21.5k queries and 22k documents. As the *queries* are function documentation written by developers, they have a different distribution than web search queries.

4.2 Experiments

Training. We train the models on both CoNaLa and PyDocs to compare generalization performance (Tab. 5) and proceed with CoNaLa mined pairs 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. 6). 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 §4.1, in Tab. 2. For all experiments, we use zero as the random seed.

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 evaluated on CodeSearchNet as it is part of its pre-training corpus. The summary statistics of each test set are shown in Table 4. 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 report Normalized Discounted Cumulative Gain (NDCG), recall@k (k = 5, 10), and Mean Reciprocal Rank (MRR) as the metrics.

4.2.1 Baselines

As baselines, we include a few simple modeling baselines (CNN/RNN/n-Bow) based on (Husain et al., 2019) (in appendix E). We further propose several contrastive learning-based baselines. We had to adapt and re-implement these baselines for the python code retrieval scenario.

• **CodeRetriever:** We propose a strong baseline based on the unimodal and bimodal contrastive learning objectives in Li et al. (2022a). We adapt their objective based on differences in the training corpora. We use the negative euclidean distance instead of cosine similarity as the similarity measure $s(x, y)$ in Eqn. 3). Additionally, Li et al. (2022a) use "NameMatcher" and "DocMatcher" retrieval to create code pairs (y, y^+) of functions for CodeSearchNet. CoNaLa has code snippets instead of functions and Stack Overflow post titles as the intent so we can not replicate this step. Instead, we pair code snippets sharing the same intent. Also, we drop the code-comment bimodal objective used in the original paper as most code snippets in CoNaLa don't have comments. We also use temperature $\tau = 1$.

$$\mathcal{L}_\phi(x, y) = -\Bigg[\log(\frac{e^{s(x,y)/\tau}}{\sum_{y' \in Y} e^{s(x,y')/\tau}}) + \log(\frac{e^{s(y,y^+)/\tau}}{\sum_{y' \in Y} e^{s(y,y')/\tau}})\Bigg], \quad (3)$$

• **(DIs)-Similarity of Source Code from Program Contrasts (DISCO):** Ding et al. (2021) proposed AST-guided rule-based perturbations for data augmentation to leverage contrastive learning for tasks such as code-clone and vulnerability detection. As the original rules are for C/C++ and Java, we adapt them for Python (see Table 8). Additionally, we transform the objective to a bimodal one (Eqn. 4) for intent representation x, code representation y, and AST perturbed negative sample y^-) as the original DISCO approach only deals with code representations. For this conversion, we use the intent representation x instead of the code representations obtained from semantics-preserving changes. Also similar to the CodeRetriever baseline we use the negative euclidean distance for similarity and temperature $\tau = 1$. We also simplify the objective by dropping the MLM and local AST node-type MLM terms.

$$\mathcal{L}_\phi(x, y) = -\log(\frac{e^{s(x,y)/\tau}}{\sum_{n=1}^{N} e^{s(x,y_n)/\tau} + e^{s(x,y_n^-)/\tau}}) \quad (4)$$

• **Dynamic Negative Sampling (DNS):** We propose a strong baseline loosely based on the STAR and ADORE algorithms (Zhan et al., 2021). STAR uses static hard negatives and randomly sampled soft negatives to simultaneously train the query and document encoder. ADORE freezes the document encoder and trains the query encoder while using

Model	MRR			NDCG			Recall@5			Recall@10		
	ID	OOD	Total	ID	OOD	Total	ID	OOD	Total	ID	OOD	Total
n-BOW	6.19	6.13	6.15	20.76	18.7	19.21	8	6.73	7.05	9.6	10.37	10.17
CNN	6.05	1.93	2.96	21.88	15.05	16.76	5.4	1.67	2.6	10.4	3.45	5.19
RNN	13.02	4.59	6.7	29.28	18.8	21.42	16.2	5.55	8.21	25.6	9.52	13.54
CB (zero shot)	2.77	2.79	2.78	16.77	15.35	15.7	3	3.28	3.21	5	4.96	4.97
CB	54.99	44.21	46.91	65.65	56.27	58.62	62.8	53.25	55.64	78.4	63.73	67.4
CB+DNS	54.51	48.59	50.07	65.51	59.94	61.33	65	57.55	59.41	79.2	67.26	70.25
CB+CR	58.03	48.55	50.92	68.43	59.89	62.02	**70.4**	57.78	60.79	**83.2**	67.05	71.09
CB+DISCO	**58.27**	40.48	44.93	**68.63**	52.87	56.81	67.4	49.23	53.77	80.2	59.7	64.82
CB+SYNC	56.28	**50.95**	**52.28**	66.89	**61.93**	**63.17**	67.8	**60.34**	**62.21**	81.8	**70.51**	**73.33**
GCB (zero shot)	9.89	18.64	16.45	23.72	29.48	28.04	12	22.2	19.65	17.22	24.77	22.88
GCB	57.4	48.4	50.65	67.78	59.94	61.19	69.8	58.84	61.58	83.2	69.1	72.62
GCB+DNS	59.28	50.87	52.97	69.26	62.08	63.88	67.6	60.59	62.34	80.8	71.45	73.79
GCB+CR	59.2	50.16	52.42	69.28	61.3	63.29	70.2	59.47	62.16	83.8	69.51	73.08
GCB+DISCO	**61.84**	42.75	47.52	**71.45**	54.82	58.98	74	51.8	57.35	**84.2**	61	66.8
GCB+SYNC	58.28	**55.44**	**56.15**	68.37	**65.73**	**66.39**	68.4	**63.99**	**65.09**	83.6	**73.38**	**75.93**
UX (zero shot)	20.7	56.24	44.39	34.9	67.32	56.51	24	63.88	50.59	30.4	70.61	57.21
UX	59.82	65.79	63.8	69.53	75.3	73.37	69.6	74.59	72.92	83	83.49	83.33
UX+DNS	59.13	66.74	64.2	69.02	76.01	73.68	72.4	75.79	74.66	84.2	82.99	83.39
UX+CR	64.35	66.98	**66.1**	73.21	76.23	**75.23**	75.66	74.93	**75.15**	**88.8**	82.8	**84.8**
UX+DISCO	**65.24**	65.5	65.41	**73.91**	75.11	74.71	**77**	73.08	74.39	84.6	81.51	82.54
UX+SYNC	60.21	**67.6**	65.13	70.06	**76.83**	74.58	72.4	**76.05**	74.83	84.8	**84.61**	84.68

Table 2: In-domain (ID) performance of the models when trained on CoNaLa and the out-of-domain (OOD) performance averaged over PyDocs, WebQuery, and CodeSearchNet (excluded for UniXcoder). CB: CodeBERT, GCB: GraphCodeBERT, UX: UniXcoder, CR: CodeRetriever. "Total": average performance with equal weights for all datasets. If we weigh the datasets relative to their sizes, UX+SYNC also achieves the best "Total" performance.

the document encoder to dynamically retrieve hard negatives from the whole dataset. The original paper applies the STAR approach first and further trains the query encoder with ADORE. However, since we have a shared query and code encoder, we strike a balance by performing an ADORE-like retrieval but only over the documents of each mini-batch (Fig. 3 in the Appendix). We find this approach to be stable during training and see significant improvements in performance for CodeBERT and GraphCodeBERT as outlined in section 5.

5 Results and Discussion

The code retrieval results are in Tab. 2. Based on all metrics, our proposed AST-guided curriculum achieves the **best OOD generalization** out of all the methods for all 3 transformer models, with the biggest improvements in GraphCodeBERT. We also observe that the DISCO implementation performs worse than the base variants on OOD performance. Our approach achieves better ID performance than the DNS approach, but slightly lower performance as compared to the CodeRetriever and DISCO approaches. Our approach achieves the **best "total" performance** in terms of the metrics averaged across all test sets for CodeBERT and GraphCodeBERT. The OOD corpora, i.e., CodeSearchNet has 60 times more queries than CoNaLa test set, and thus the generalization results holds more significance in terms of the number and type of queries that our models can answer.

As CodeSearchNet is excluded from test for UniXcoder, the gain of our approach in OOD performance doesn't reflect in total score in comparison to CodeRetriever and DISCO (MRR and NDCG). We analyze this in more detail later. Finally we observe the simple modeling baselines fail to reach zero-shot performance of structure-aware LMs like UniXcoder, highlighting the importance of incorporation of structure for code search.

Comparison of errors made by our approach (UX+SYNC) and CodeRetriever (UX+CR) for UniXcoder. We analyze the drop in in-domain performance of UX+SYNC against UX+CR by finding instances where the UX+CR approach retrieves the correct document within the top 5 results and

our approach doesn't. There are 23 such instances. Among these, 5 are labeling errors, 6 are incorrect operation invoke and 5 are correct function with incorrect arguments (more in Tab. 16). We further analyze the CodeBLEU (Ren et al., 2020) scores of the top-ranked code candidates for each query. UX+SYNC and UX+CR achieves a CodeBLEU score of 71.65 and 73.56 ID performance resp.ly. Interestingly, for the error instances by UX+SYNC (i.e., where the correct document is not in the top-5 candidates) we achieve a CodeBLEU score of 18.36 as compared to 16.62 by UX+CR. This implies that our model retrieves syntactically and lexically closer snippets to the gold labels compared to UX+CR. For the OOD data (WebQuery and PyDocs), UX+SYNC achieves a CodeBLEU score of 60.91 compared to 57.97 achieved by UX+CR, and a score of 10.69 as compared to 9.47 for mistakes, showing the trend of better structural similarity holds for OOD as well.

Why do CodeRetriever and DISCO have better ID performance but worse OOD performance? Both DISCO and CodeRetriever use cross-entropy loss instead of triplet margin loss (which is used for our approach) and gain ID performance at the expense of OOD generalization. This ID-OOD tradeoff can be observed even while adding more negatives to CodeBERT (Tab. 9), which matches our ID performance on CoNaLa, while lagging behind in OOD performance.

For DISCO, the OOD performance is worse than the base variants (UX/GCB/CB rows) trained with soft negatives and a triplet margin loss. We performed additional ablations over the CodeRetriever objective to tease out the influence of the loss function (triplet margin loss vs. cross-entropy loss) and the effect of the unimodal loss component (in Appendix C.1). The results suggest that the choice of triplet margin loss leads to better OOD performance at the cost of ID performance and the unimodal code similarity objective also improves the OOD performance at the cost of the ID performance, which explains why CodeRetriever outperforms DISCO on OOD data, while DISCO does better on ID data. This aligns with the intuition that margins add robustness to models and prevent overfitting, motivating our choice of triplet margin loss as the objective for the AST-guided curriculum.

Qualitative Analysis. We show some motivating examples in Appendix Tab. 13. For the first example, all the top 5 retrieved code snippets using 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 improvements for CodeBERT and GraphCodeBERT and a slight drop for some rules in UniXcoder in Tabs. 14 & 15.

6 Conclusion

For neural code search, Transformers-based pre-trained models make certain common mistakes which indicates loss of syntactic and semantic information, while learning the representation. To learn robust structure-preserving representation during fine-tuning, we propose a structure-aware hard negative sampling through AST perturbation along with a mastering-rate-based curriculum, where our AST perturbation rules are motivated by common semantic and syntactic variations of code. Our experiments show significant improvement over standard contrastive learning for three SOTA transformer models on four code retrieval datasets (in ID and OOD settings), outperforming competitive contrastive learning baselines like DISCO and CodeRetriever over OOD generalization. Interestingly, our method shows improvement even for UniXCoder which is exposed to the underlying AST structure of the code snippets during pre-training. Additionally, CodeBLEU scores show that even when models trained using our approach makes mistakes in retrieval, it still exhibits comparably better structural similarity with gold truth code snippets.

Acknowledgments

The work was partially supported by SERB SRG/2022/000648, grant (PI: Somak Aditya). For computational requirements, we thank and acknowledge the use of Paramshakti Supercomputing Facilities by IIT Kharagpur.

Limitations

- In our approach, we use a set of AST perturbation rules (manually written by us and following previ-

ous work) to create hard negatives around common grammatical (and semantic) constructs. While we do not assume "necessity" or "sufficiency" of the rules, it is still hard to define any such properties for a collection of rules.

• Our approach mitigates the problem of training instability and susceptibility to local optima encountered in hard negative mining while achieving the best OOD generalization. However it fails to achieve the best ID performance (compared to other contrastive learning approaches). Some of these limitations stem from the choice of the triplet margin loss function that improves the OOD performance at the expense of the ID performance as shown by the CodeRetriever ablations (Appendix). We will investigate this further as part of future work.

Ethics Consideration

In current work, we utilize python code retrieval datasets that are publicly available. Some of the datasets are semi-automatically curated from scraping StackOverflow and Python standard library API reference. To the best of our knowledge, we have not noted any offensive or adult content in the datasets. Secondly, current code language models are targeted towards high-resource language such as English. If our work is accepted, we also plan to work on local low-resource language and this is a reason we use models such as UniXCoder that have the potential to be extended to multiple languages. Apart from this, we do not foresee any specific ethical considerations.

References

Kian Ahrabian, Aarash Feizi, Yasmin Salehi, William L. Hamilton, and Avishek Joey Bose. 2020. Structure aware negative sampling in knowledge graphs. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6093–6101, Online. Association for Computational Linguistics.

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

Guanyi Chu, Xiao Wang, Chuan Shi, and Xunqiang Jiang. 2021. Cuco: Graph representation with curriculum contrastive learning. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, pages 2300–2306. ijcai.org.

Yangruibo Ding, Luca Buratti, Saurabh Pujar, Alessandro Morari, Baishakhi Ray, and Saikat Chakraborty. 2021. Towards learning (dis)-similarity of source code from program contrasts. In *Annual Meeting of the Association for Computational Linguistics*.

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*.

The Python Software Foundation. 2023a. Python 3.7 AST Grammar and Parsing. https://docs.python.org/3.7/library/ast.html. Accessed: 2023-5-23.

The Python Software Foundation. 2023b. Python 3.7 Unparser. https://github.com/python/cpython/blob/3.7/Tools/parser/unparse.py. Accessed: 2023-5-23.

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

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*.

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

Junjie Huang, Duyu Tang, Linjun Shou, Ming Gong, Ke Xu, Daxin Jiang, Ming Zhou, and Nan Duan. 2021. 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)*, pages 5690–5700, Online. Association for Computational Linguistics.

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

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

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

Xiaonan Li, Yeyun Gong, Yelong Shen, Xipeng Qiu, Hang Zhang, Bolun Yao, Weizhen Qi, Daxin Jiang, Weizhu Chen, and Nan Duan. 2022a. Coderetriever: Unimodal and bimodal contrastive learning for code search.

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. 2022b. Competition-level code generation with alphacode. *ArXiv*, abs/2203.07814.

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

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *ArXiv*, abs/1907.11692.

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. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *ArXiv*, abs/2102.04664.

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

Michael Pradel and Koushik Sen. 2018. Deepbugs: A learning approach to name-based bug detection. *Proc. ACM Program. Lang.*, 2(OOPSLA).

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, M. Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *ArXiv*, abs/2009.10297.

Joshua David Robinson, Ching-Yao Chuang, Suvrit Sra, and Stefanie Jegelka. 2021. Contrastive learning with hard negative samples. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

Ensheng Shi, Wenchao Gub, Yanlin Wang, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. 2022. Enhancing semantic code search with multimodal contrastive learning and soft data augmentation. *ArXiv*, abs/2204.03293.

Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. 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*.

Xiao Wang, Qiong Wu, Hongyu Zhang, Chen Lyu, Xue Jiang, Zhuoran Zheng, Lei Lyu, and Songlin Hu. 2022. Heloc: Hierarchical contrastive learning of source code representation. *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)*, pages 354–365.

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

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

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

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

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

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

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

A Method

A.1 Algorithm Outline for AST perturbation

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

	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; exp_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 3: All 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, and constant.

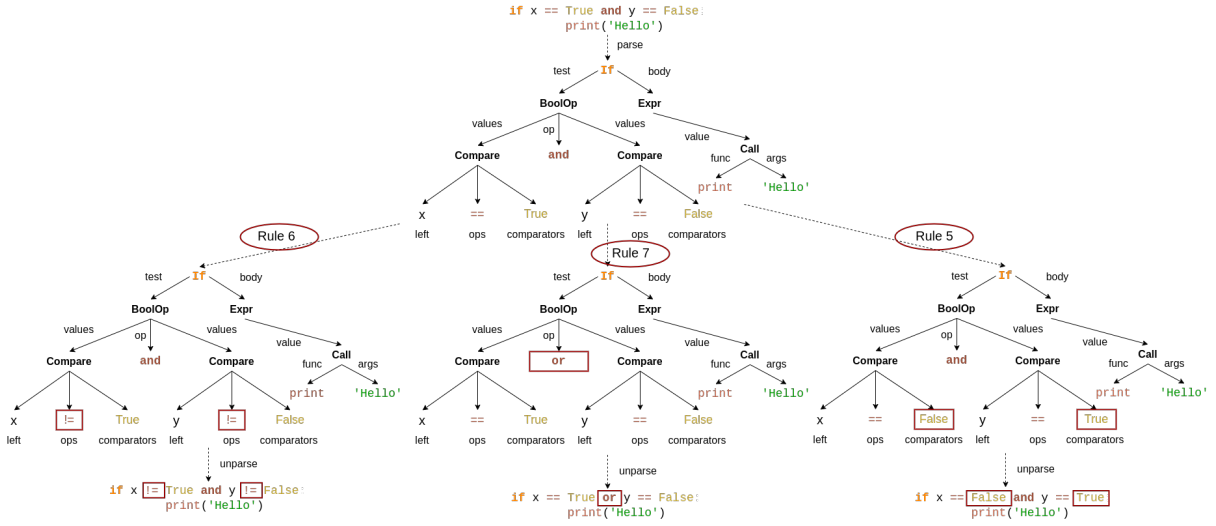

Figure 2: **AST perturbation in action:** For the given code snippet, rule 5, 6, 7 & 9 are applicable, leading to 4 AST-based hard negative candidates. Rule 5 flips the leaf nodes corresponding to the named constants "True" & "False", while rule 6 replaces boolean "and" operator leaf node with an "or" 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.

Algorithm 1: Pseudocode for AST guided code perturbation

Data: $\rho, \mathcal{R}$ *[r]ρ is program snippet, $\mathcal{R}$ is set of rules
Result: $\mathcal{P} = \{\rho'_1, \ldots \rho'_n\}$ *[r]ρ'_i is i^{th} corrupted program snippet

$\mathcal{T} \leftarrow$ parseAST(ρ);
$\mathcal{S} \leftarrow \emptyset$*[r]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$\rangle$)*[r]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($\mathcal{T}$)*[r]Create a copy of the AST to modify later into a corrupted program snippet
    $\mathcal{T}_c \leftarrow$ applyRule($\mathcal{T}_c, \eta$, r)*[r]Apply rule on valid site node and transform AST
    $\rho' \leftarrow$ unparseAST($\mathcal{T}_c$)*[r]Regain program snippet from transformed AST
    $\mathcal{P}$.push(ρ')*[r]Collect set of corrupted program snippets

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 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 1). 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[2] 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 and 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 4. Notably, PyDocs, WebQuery and CodeSearchNet all vary

[2] 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):
				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 4: Statistics of each dataset: CoNaLa, PyDocs, WebQuery, CodeSearchNet.

from CoNaLa in the way queries are expressed. WebQuery and CodeSearchNet contain larger code snippets compared to PyDocs and CoNaLa.

B.1 Filtering CoNaLa Corpus

As mentioned in the Experiments section, we use 100k most relevant pairs of CoNaLa for fine-tuning, as it achieves comparable or better performance on the CoNaLa test set. Details are shown in Table 6.

C Ablation Studies

C.1 Ablation I: Utilizing CodeRetriever objective With CodeBERT

We perform the following ablations for the CodeRetriever objective with CodeBERT:

1. **Impact of unimodal loss:** To test the impact of the unimodal loss component, we run an experiment without it, using only the bimodal loss.
2. **Impact of choice of loss function:** We test the tradeoffs involved in choosing triplet margin loss instead of cross-entropy loss. Under this setting, both the unimodal and bimodal loss are triplet margin losses instead of cross entropy losses.
3. **No. of epochs:** Impact of no. of training steps/epochs on ID and OOD performance. We test this out with 5, 10, and 15 epochs.
4. **Similarity measure:** Using cosine similarity vs using negative euclidean distance.

We summarize the results of these ablations in Table 7.

The results indicate that performance usually degrades over epochs meaning fewer training steps are better. Notably, it is the OOD performance that drops drastically while the ID performance does decrease much, which indicates overfitting to noise. Using cosine similarity leads to very poor performance as compared to the negative euclidean distance, prompting us to use the latter as the similarity measure for DISCO as well. We also see that the loss type affects the ID and OOD breakdown. Using triplet margin loss greatly degrades the ID but also achieves the best OOD performance, however the average performance is poor. Finally, we observe that removing the unimodal loss improves the ID at the expense of the OOD performance, which indicates that it potentially acts as a regularization to prevent overfitting the bimodal loss.

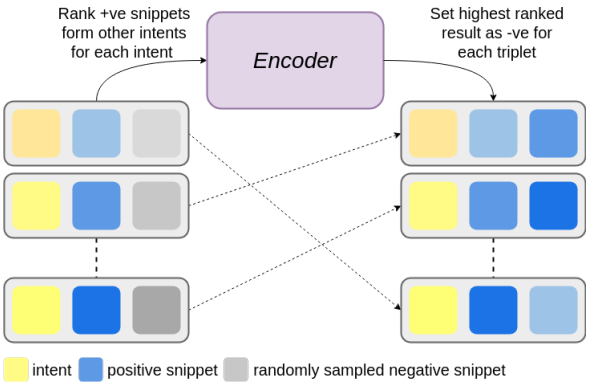

Figure 3: We create a feedback loop in the model training by using the current model weights to pair each intent-snippet pair with the closest snippet from another intent. These create the hardest negatives at a batch level, which are expected to be harder than randomly sampled negatives.

Model	Trained on	MRR	NDCG	Recall@5	Recall@10
CodeBERT	CoNaLa	46.91	58.62	55.64	67.4
GraphCodeBERT	CoNaLa	50.65	61.19	61.58	72.62
UniXcoder	CoNaLa	**63.8**	**73.37**	**72.92**	**83.33**
CodeBERT	PyDocs	44.08	55.36	48.57	56.07
GraphCodeBERT	PyDocs	52.42	62.94	58.82	67.18
UniXcoder	PyDocs	49.22	60.42	54.83	63.42

Table 5: Effect of training dataset on the generalizability of models.

Model	MRR	NDCG	Recall@5	Recall@10
CodeBERT	51.68	63.21	61.4	77.6
CodeBERT 100k	**54.99**	**65.65**	**62.8**	**78.4**
GraphCodeBERT	57.03	67.3	66.2	79.2
GraphCodeBERT 100k	**57.4**	**67.78**	**69.8**	**83.2**
UniXcoder	59.77	69.45	69.2	83
UniXcoder 100k	**59.82**	**69.53**	**69.6**	**83**

Table 6: 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.

Unimodal Loss	Loss Type	Similarity measure	Epochs	MRR			NDCG			Recall@5			Recall@10		
				Total	ID	OOD	Total	ID	OOD	Total	ID	OOD	Total	ID	OOD
True	cross entropy	-euclid_dist	5	50.92	58.03	**48.55**	62.02	68.43	59.89	**60.79**	70.4	57.78	71.09	83.2	**67.05**
True	cross entropy	-euclid_dist	10	48.68	57.28	45.82	60.13	67.85	57.55	58.46	69.8	54.67	69.35	83.6	64.59
True	cross entropy	-euclid_dist	15	45.92	58.03	41.88	57.65	68.2	54.14	54.93	67.8	50.64	65.91	81.6	60.67
True	cross entropy	cosine_sim	5	19.65	30.65	16.19	34.4	44.91	30.9	24.2	36.8	20.3	33.92	50.4	28.42
True	cross entropy	cosine_sim	10	18.76	27.54	15.83	33.5	42.67	30.44	23.08	33.4	19.64	32.37	47	27.5
True	cross entropy	cosine_sim	15	18.81	27.61	15.88	33.53	42.66	30.49	22.81	33	19.41	32.53	47.2	27.64
False	cross entropy	-euclid_dist	5	**50.97**	**60.73**	47.72	62.04	**70.6**	59.19	60.6	**73**	56.46	**71.52**	**86**	66.69
True	triplet margin	-euclid_dist	5	50.87	53.82	**49.89**	**62.06**	64.7	**61.18**	60.01	64.2	**58.61**	71.05	77.4	**68.94**

Table 7: Various ablations for the CodeRetriever objective for CodeBERT.

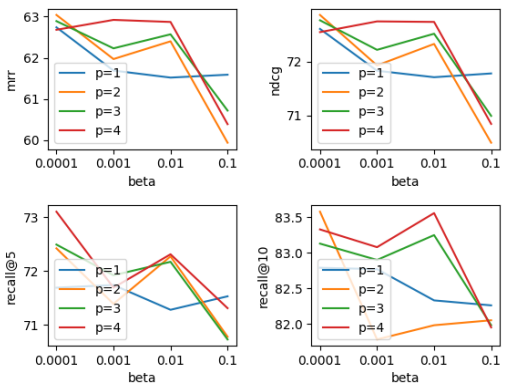

Figure 4: Hyperparameter search over the p and β for UniXcoder

C.2 Ablation II: Hyper Parameter Variations (β in Von-Mises Distribution, p in Mastering Rate Curriculum, Warmup steps)

We explore variations of the various hyper-parameters associated with the mastering-rate curriculum-learning algorithm and the von-Mises Fischer sampling. Intuitively, higher β samples harder negatives (negatives more similar to query) through von-Mises sampling, and higher p forces the curriculum to focus more on learning from soft negatives (i.e. mastering the soft negative task first); therefore creating a natural tradeoff.

Figure fig. 4 shows various variations of the β (used for the sampling in §3.1) and p (used in the

Our implementation of DISCO	Original DISCO implementation
Change ‘int‘/‘float‘ constant to ‘str‘	DataType Misuse
Flip comparators: == to !=, <to >=, >to <=, ’is’ to ’is not’, ’in’ to ’not in’ and vice versa for each case	Change of Conditional Statements
Replace If-Else statement with if’s body	Change of Conditional Statements
Swap function arguments	Change of Function Calls
Replace If-Else statement with else’s body	Change of Conditional Statements
Change ‘str‘ constant to ‘int‘	DataType Misuse
Change ‘str‘ constant to ‘float‘	DataType Misuse
VarMisuse: Replace variables with each other	Misuse of Variables
Division-by-zero error introduced	Misuse of Values

Table 8: A comparison of the perturbation rules implemented in our adaptation of the DISCO baseline for Python. We remove rules like “Misuse of Pointers” that don’t have any Python equivalents but were proposed in the original DISCO paper for C/C++

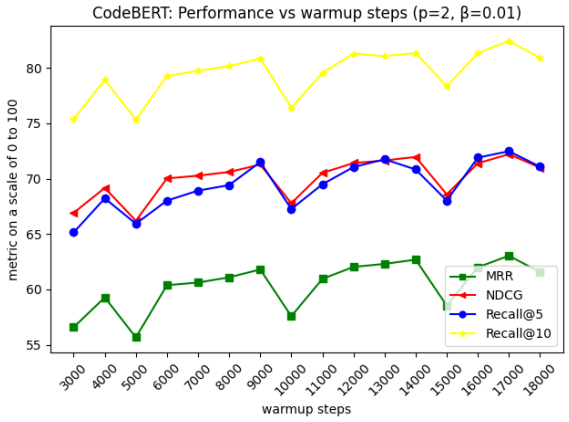

Figure 5: Effect of warmup steps on the performance of CodeBERT over all 4 metrics, with $p = 2$ & $\beta = 0.01$

hard negative attention calculation for the mastering rate curriculum) 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 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

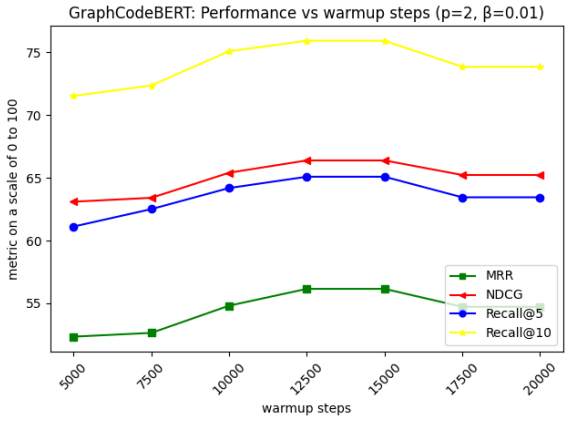

Figure 6: Effect of warmup steps on the performance of GraphCodeBERT over all 4 metrics, with $p = 2$ & $\beta = 0.01$

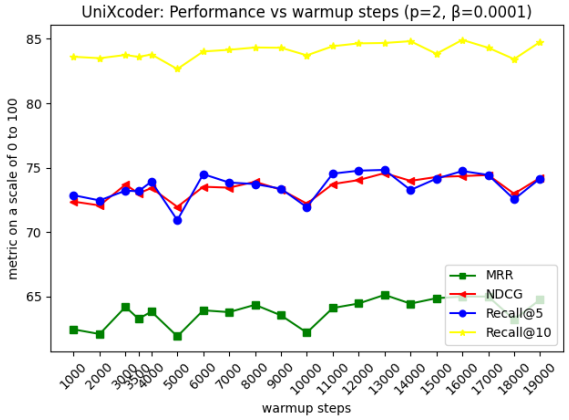

Figure 7: Effect of warmup steps on the performance of UniXcoder over all 4 metrics, with $p = 2$ & $\beta = 0.0001$

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 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.3 Ablation III: Effect of Adding More Negatives

We conduct experiments with 10 soft negatives per NL-PL pair instead of 3 as used in the previous triplet margin loss-based experiments and this seems to boost both the OD and IID performance of the model as shown in table 9, at the expense of three to four times increased training time. While this achieves slightly better in-domain performance than SYNC, it still lags behind in the out-of-domain performance which shows our method can achieve better generalization with fewer (one-third) negatives.

C.4 Ablation IV: 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 curriculum ("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 10. We see that using hard negatives only leads to the worst performance while just introducing some soft negatives through 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.5 Ablation V: Effect of different perturbation rule sets

We conducted experiments with different sets of perturbation rules to measure the impact of the rule choice as shown in table 11. Each setting is explained below:

SYNC rules: The standard setting where we use the rules outlined in table 3.

DISCO rules: For this setting we use the rules outlined in table 8 which we implemented for the DISCO baseline for Python.

Natural rules: For this setting we considered the union of SYNC and DISCO rules with "unnatural rules" being removed. We classed unnatural rules as rules that induce bugs that a Python developer is unlikely to make. These unnatural rules included the following rules from table 3: rule 4, rule 8, and from "VarMisuse: Replace variables with each other" and "Division-by-zero error introduced" table 8.

In Python string constants (like "Hello World") and floating point/integer (like `1.0` or `1`) constants are very unlikely to be confused with each other or used in similar contexts explaining the choice of excluding rule 4, for rule 8 an experienced developer is unlikely to declare a variable with the name of common python functions like "print" (often highlighted in a different color by IDEs). Finally, the VarMisuse error would swap all occurrences of two randomly picked variables while a user is more likely to make errors where they swap just one occurrence and for the division, by zero-error, we replace denominators in division operators with zero, which a developer is unlikely to do (it is more likely to occur due to a mistake in an equation or numerical instability).

Top-12 rules: for this experiment we again take the union of DISCO and SYNC rules and then used the regularly fine-tuned CodeBERT model to see the triplet separation accuracy (accuracy of projecting the correct code closer to the intent as compared to the hard negative) for each rule. We considered this to be the rule difficulty and picked the top-12 rules based on this score.

The results show that our originally chosen set of rules for SYNC works the best (except ID recall@5), especially for OOD performance, indicating some sensitivity of our approach to the choice of rules.

D Additional Experimental Results, Qualitative Analysis and Analogy Tests

D.1 Dataset-wise performance breakdown

The dataset-wise performance breakdown for all three transformer models (CodeBERT, GraphCodeBERT, and UniXcoder) and all five training variants (regular triplet training, dynamic negative sampling (DNS), DISCO, CodeRetriever and our proposed AST-guided training curriculum (AST)) is shown

Model	MRR			NDCG			Recall@5			Recall@10		
	ID	OOD	Total	ID	OOD	Total	ID	OOD	Total	ID	OOD	Total
CB 3 -ves	54.99	44.21	46.91	65.65	56.27	58.62	62.8	53.25	55.64	78.4	63.73	67.4
CB 10 -ves	**56.58**	47.99	50.14	**67.03**	59.37	61.28	**68.6**	57.37	60.18	**82.6**	66.58	70.59
CB+SYNC	56.28	**50.95**	**52.28**	66.89	**61.93**	**63.17**	67.8	**60.34**	**62.21**	81.8	**70.51**	**73.33**

Table 9: Table showing the impact of using more soft negatives per NL-PL pair with the triplet margin loss. CB refers to CodeBERT and 3 -ves implies 3 soft negatives are sampled per NL-PL pair (the default) setting, while 10 -ves represents a setting where 10 -ves are sampled per NL-PL pair. Clearly more negatives improve the ID and OOD performance but the OOD performance is still below SYNC (CB+SYNC) which uses 3 -ves per NL-PL pair.

Model	Curriculum Type	MRR	NDCG	Recall@5	Recall@10
CodeBERT	soft neg	46.91	58.62	55.64	67.4
CodeBERT	hard neg	18.4	33.28	23.37	32.56
CodeBERT	rand curr	46.59	58.36	55.46	66.2
CodeBERT	MR curr	**52.28**	**63.17**	**62.21**	**73.33**
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**

Table 10: Results of ablations with various curriculum types. Using hard negatives only leads to worse results than using soft negatives only because of training instability and getting trapped in local minima. Meanwhile, even a random curriculum that mixes soft and hard negatives with equal probabilities doesn't match the performance achieved by using soft negatives only, indicating the challenge of effective curriculum design.

Model	MRR			NDCG			Recall@5			Recall@10		
	ID	OOD	Total	ID	OOD	Total	ID	OOD	Total	ID	OOD	Total
SYNC Rules	**56.28**	**50.95**	**52.28**	**66.89**	**61.93**	**63.17**	67.8	**60.34**	**62.21**	**81.8**	**70.51**	**73.33**
DISCO Rules	53.92	48.74	50.04	64.98	60.13	61.34	**69.2**	57.4	60.35	81.4	67.97	71.32
Natural Rules	53.43	48.4	49.65	64.61	59.87	61.06	68	57.25	59.94	80.4	67.98	71.08
Top-12 Rules	54.14	48.63	50.01	65.15	60.03	61.31	69	57.02	60.01	81.0	67.76	71.07

Table 11: The in-domain (ID) performance of CodeBERT, when trained on CoNaLa and the out-of-domain performance (OOD) averaged over PyDocs, WebQuery, and CodeSearchNet for different sets of rules.

in table 12.

D.2 Qualitative Analysis

We show the qualitative effect of our approach through examples in Table 13.

D.3 How does the curriculum help with hard negatives?

We analyze the variation in validation recall@5 between CodeBERT and CodeBERT+SYNC (our curriculum learning-based approach) with the weight of the soft negatives (soft negative attention) for CodeBERT+SYNC as determined by our curriculum. The results are shown in figure 8.

For the initial steps, the soft negative attention

Model	CoNaLa				PyDocs				WebQuery				CodeSearchNet			
	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)	2.77	16.77	3.0	5.0	6.48	20.37	8.65	12.26	1.26	16.45	0.48	1.53	0.63	9.22	0.69	1.08
CB	54.99	65.65	62.8	78.4	64.33	72.23	74.52	83.89	41.87	58.34	51.34	66.06	26.44	38.24	33.9	41.24
CB+DNS	54.51	65.51	65.0	79.2	69.76	76.39	78.37	86.3	42.83	59.01	52.58	65.97	33.19	44.43	41.7	49.52
CB+CR	58.03	68.43	**70.4**	**83.2**	68.52	75.76	77.4	85.82	46.1	61.71	56.21	68.64	30.72	42.2	39.13	46.68
CB+DISCO	**58.27**	**68.63**	67.4	80.2	60.81	69.31	70.91	82.21	40.91	57.49	51.24	64.53	19.72	31.82	25.53	32.35
CB+SYNC	56.28	66.89	67.8	81.8	**71.09**	**77.47**	**79.81**	**87.98**	**46.57**	**62.03**	**57.07**	**71.51**	**35.2**	**46.28**	**44.15**	**52.04**
CB+SYNC (hard neg)	20.92	36.4	26.2	37.4	18.34	32.48	23.56	32.45	25.92	44.49	32.7	45.79	8.42	19.75	11.04	14.58
CB+SYNC (rand curr)	53.88	64.92	63.0	77.6	63.49	71.33	73.08	81.25	42.26	58.81	51.91	65.01	26.74	38.39	33.87	40.95
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+CR	59.2	69.28	70.2	83.8	71.02	77.51	79.81	88.46	44.81	60.57	54.78	68.64	34.65	45.81	43.84	51.41
GCB+DISCO	**61.84**	**71.45**	**74**	**84.2**	65.67	73.19	76.44	84.62	39.88	56.71	49.81	62.43	22.71	34.57	29.14	35.97
GCB+SYNC	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+SYNC (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+SYNC (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+CR	64.35	73.21	75.6	**88.8**	**85.44**	**88.85**	91.35	93.03	48.52	63.62	58.51	72.56	-	-	-	-
UX+DISCO	**65.24**	**73.91**	**77**	84.6	83.82	87.61	90.14	93.99	47.17	62.61	56.02	69.02	-	-	-	-
UX+SYNC	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+SYNC (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+SYNC (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 12: 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

remains 1, meaning only soft negatives are picked, which corresponds to the warmup steps. CodeBERT+SYNC remains above CodeBERT during this stage. Once the warmup steps are finished the soft negative attention drops and hard negatives are introduced leading to an initial drop in the validation of CodeBERT+SYNC, but as the training continues the validation performance climbs up above CodeBERT. The behavior in between the logging steps is slightly oscillatory which we have smoothed here through a moving average across the intervals between the logging steps.

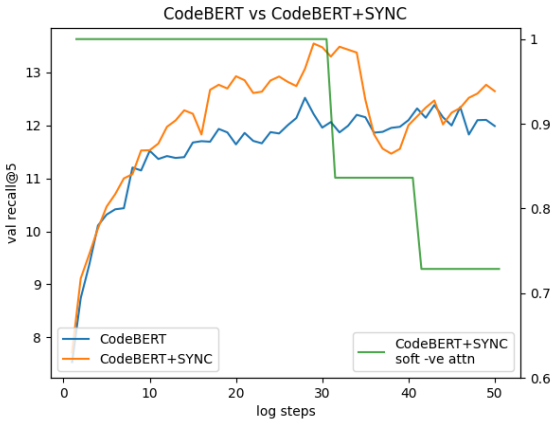

Figure 8: **How the curriculum helps:** validation recall@5 for CodeBERT and CodeBERT+SYNC and soft negative attention of CodeBERT+SYNC at uniform logging steps

D.4 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

Model, Dataset & Query	Gold Candidates	AST Hits@5	Baseline Hits@5
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)
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)
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)

Table 13: Qualitative examples illustrating how our AST-guided curriculum corrects errors

test bed of 1800 examples of the form a:b; c:d with 200 samples per rule category. Some examples are shown in table 17. During the sampling process, we also filter out code snippets smaller than 30 characters, to ensure example quality.

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.

E Details about Simple Baseline Architectures

We train a few simple modeling-based, baselines in a siamese configuration similar to Husain et al. (2019) but use the same architecture for both the text & code encoders to create a universal encoder. 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.

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

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

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+SYNC	**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+SYNC	**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+SYNC	89	**96.5**	**95.5**	**96.5**	**96.5**	**96.5**	**96**	91	95.5	94.778

Table 14: 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+SYNC	**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+SYNC	**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+SYNC	**89**	**96.5**	**95.5**	**96.5**	**96.5**	**96.5**	**96**	91	96	94.833

Table 15: Analogy test results similar to Table 14 using cosine similarity instead of euclidean distance to rank ds for a given $b + c - a$

Error	Freq	Description
correct data structure, incorrect operation	5	Correct data structure or object used but incorrect operation invoked
correct function call, incorrect arguments	6	Correct function called with incorrect arguments
extra function call	1	At least one function call which shouldn't have been invoked
incorrect data structure	2	Wrong data structure or object is operated on
incorrect function call	3	Incorrect function call invoked
labeling error	5	
missing function calls	1	Correct solution involves multiple function calls and at least one or more are missing

Table 16: Analysis of the types and frequencies of errors made by our approach over the CodeRetriever approach for in-domain performance (i.e., on CoNaLa test set.)

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).

• **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.

F Computational Budget

We used 16 GB Tesla P100 GPUs for training with roughly 2.5 hrs/epoch (roughly 3 times more for the increased hard negatives experiment) and 5 epochs for each experiment. We carried out roughly 170 experiments including the various ablations and some exploration. The CodeBERT and GraphCodeBERT models are RoBERTa (Liu et al., 2019) based encoders with roughly 123M parameters (same as RoBERTa-base) while UniXcoder has both an encoder and a decoder, but roughly the same number of parameters at 125M.

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 17: 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.