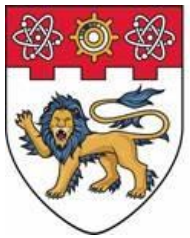


NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

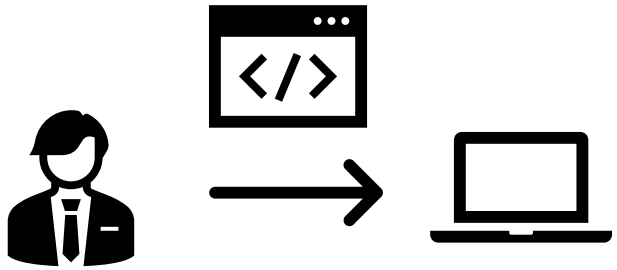
Semantic Alignment Between Natural Language and Programming Language

Haochen Li

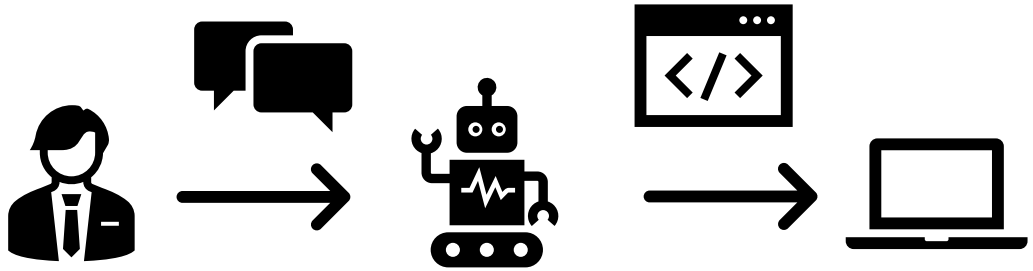
HAOCHEN003@ntu.edu.sg

AI-assisted Coding Changes How We Write Code

Before



After



A coding assistant needs to understand requirements expressed in natural language and translate them into programming language.

Overview of My Research

- One-to-One Mapping
  - How can we align representations of natural language and programming language better, given a dataset that consists of NL-code pairs?
- One-to-Many Mapping
  - In fact, an NL description often corresponds to multiple code implementations. Considering this, how can we further enhance semantic alignment?
- Many-to-Many Mapping
  - Similarly, a code snippet corresponds to multiple NL descriptions. Can we create high-quality synthetic data by leveraging this relationship?

Representation Alignment

Objective:

Similarity of a positive pair

(Sort a given matrix in ascending order according to the sum of its rows. ,)

```
def sort_matrix(M):
    result = sorted(M, key=sum)
    return result
```

> (Sort a given matrix in ascending order according to the sum of its rows. ,)

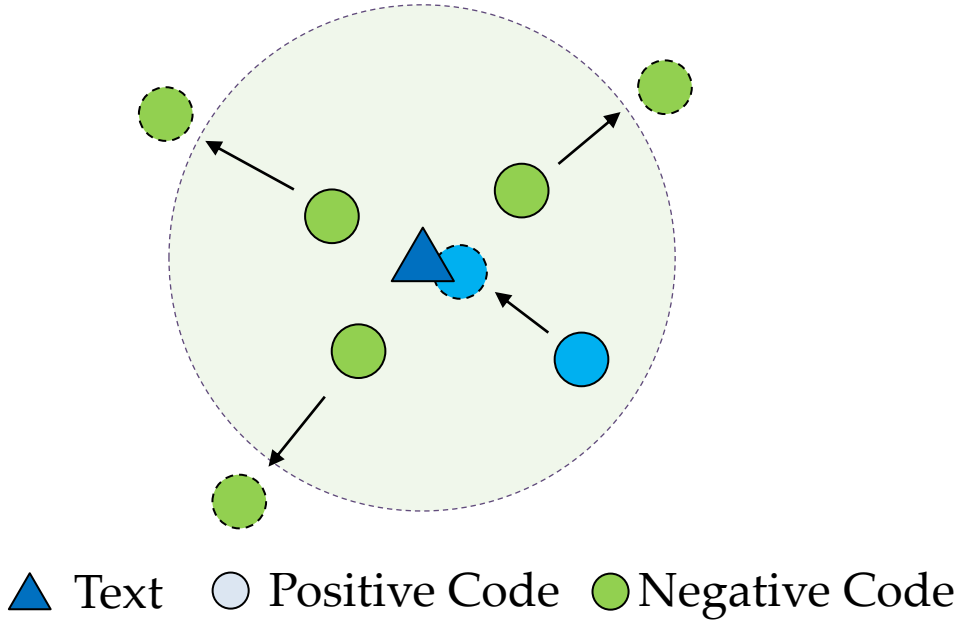
```
def square_perimeter(a):
    perimeter=4*a
    return perimeter
```

Similarity of any negative pairs

Optimization with Contrastive Learning

Dominant choice: InfoNCE

$$\mathcal{L} = -\mathbb{E}_i \left[\log \frac{\exp(text_i \cdot code_i)}{\exp(text_i \cdot code_i) + \sum_{j \neq i} \exp(text_i \cdot code_j)} \right]$$



Numerator:
Pull together representations

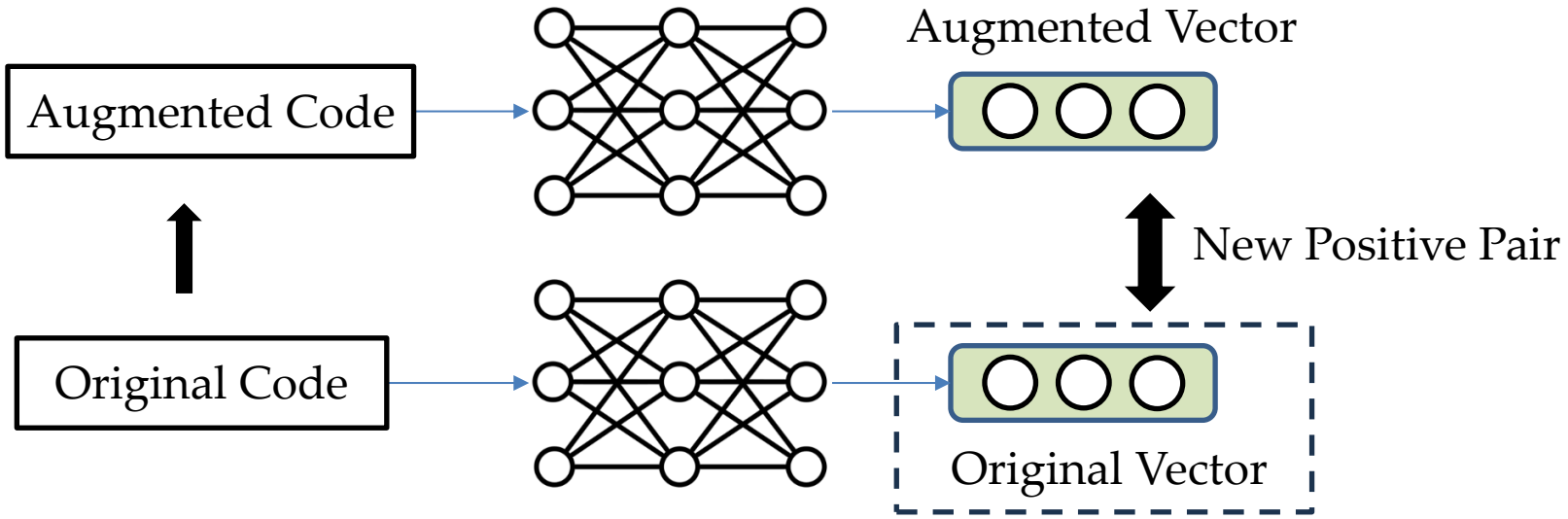
Denominator:
Push away representations

Augment Positive Pairs

Raw-data level augmentation methods are resource-consuming and limited in variety.

```
def func():
    x = [1, 2, 3]
    y = 0
    while count < len(x):
        y += x[count]
        count += 1
    return y
```

```
def func():
    x = [1, 2, 3]
    y = 0
    for i in x:
        y += i
    return y
```



Can we directly augment this vector?

Representation-level Augmentation

Existing Work:

Linear Interpolation

e.g. $0.8 \times h_1 + 0.2 \times h_2$

Stochastic Perturbation

e.g. $[h_{10}, h_{11}, h_{12}, h_{13}]$--> $[0, h_{11}, h_{12}, 0]$

New Method:

Linear Extrapolation

e.g. $1.2 \times h - 0.2 \times h'$

Binary Interpolation

e.g. $[h_{10}, h_{11}, h_{12}, h_{13}]$--> $[h_{10}, h'_{11}, h_{12}, h'_{13}]$

Gaussian Scaling

e.g. $h + \beta \odot h' \ \forall \beta \sim N(0, \sigma)$

General Format:

$$h^+ = \alpha \odot h + \beta \odot h'$$

Li, et al. "Exploring Representation-level Augmentation for Code Search." EMNLP 2022.

Theoretical Analysis

- Optimizing InfoNCE loss $\mathcal{L}$ improves the lower bound of mutual information $\boldsymbol{I}$ for a positive pair:

$$I \geq log(B) - \mathcal{L}$$

- With augmentation, the lower bound is:

$$I \geq \frac{1}{\alpha^2}(\log(NB) - \mathcal{L} - (\beta^2 + 2\alpha\beta) \cdot I^-)$$

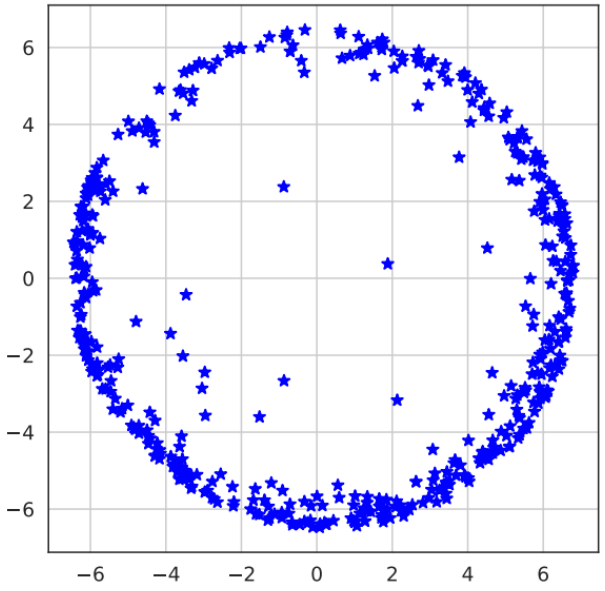
Moderate augmentation (β close to 0) leads to a **tighter** lower bound.

Mutual information indicates the similarity between two samples (the higher, the better)

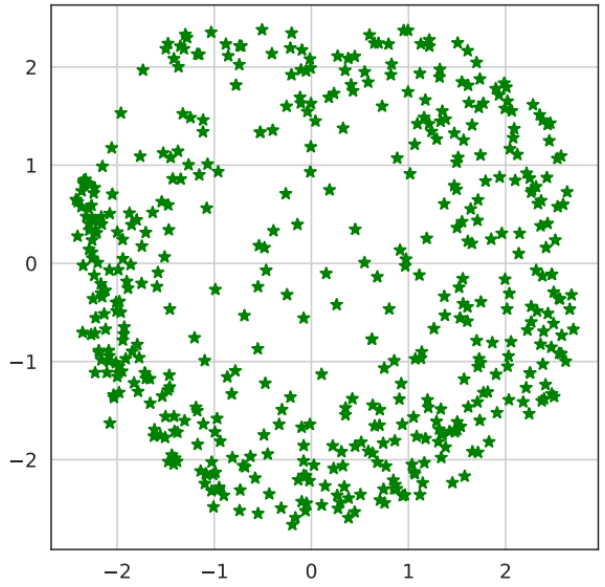
Experiment Results

Model	Ruby		JavaScript		Go		Python		Java		PHP	
	Original	w/ RA	Original	w/ RA	Original	w/ RA	Original	w/ RA	Original	w/ RA	Original	w/ RA
RoBERTa (code)	64.1	**66.5**	58.3	**61.2**	86.7	**89.2**	61.0	**66.3**	63.4	**67.4**	58.4	**61.7**
CodeBERT	64.8	**66.4**	59.4	**60.8**	87.8	**89.0**	63.6	**65.4**	66.3	**67.4**	61.5	**61.9**
GraphCodeBERT	70.5	**72.1**	64.7	**67.1**	89.6	**90.3**	69.0	**70.8**	69.1	**70.8**	64.8	**65.6**

w/o augmentation



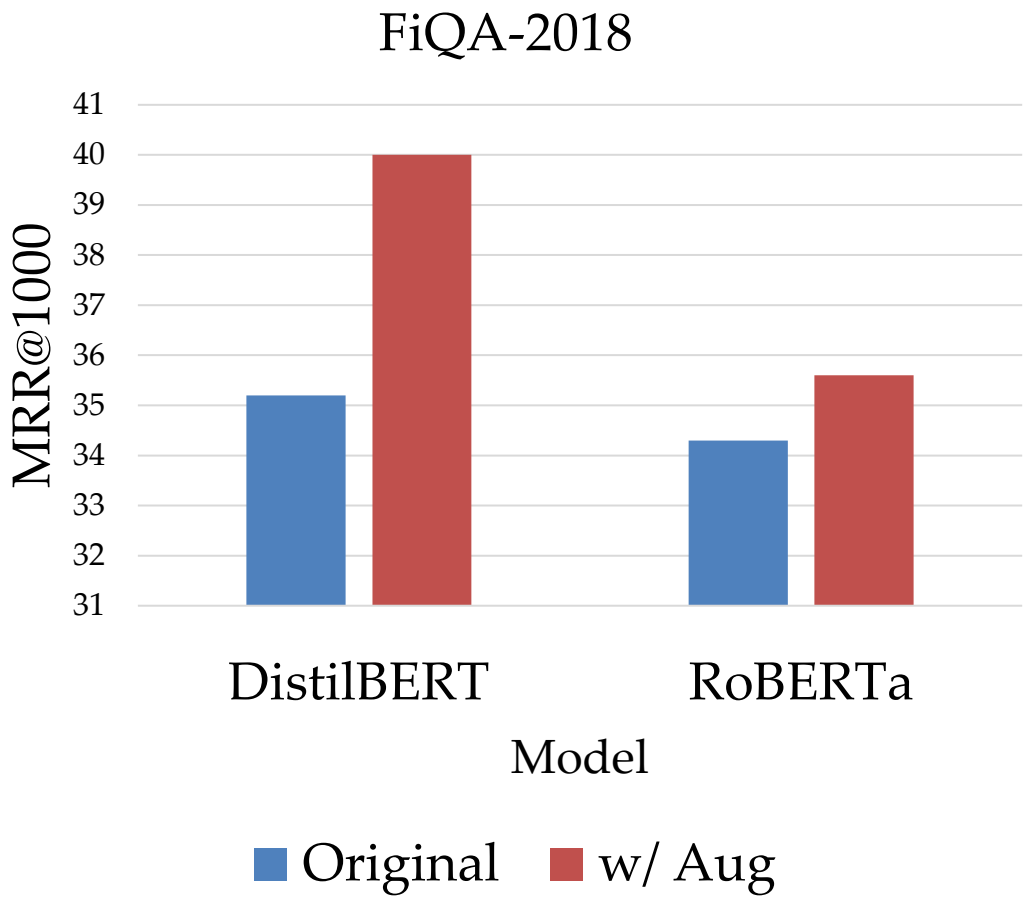
w/ augmentation

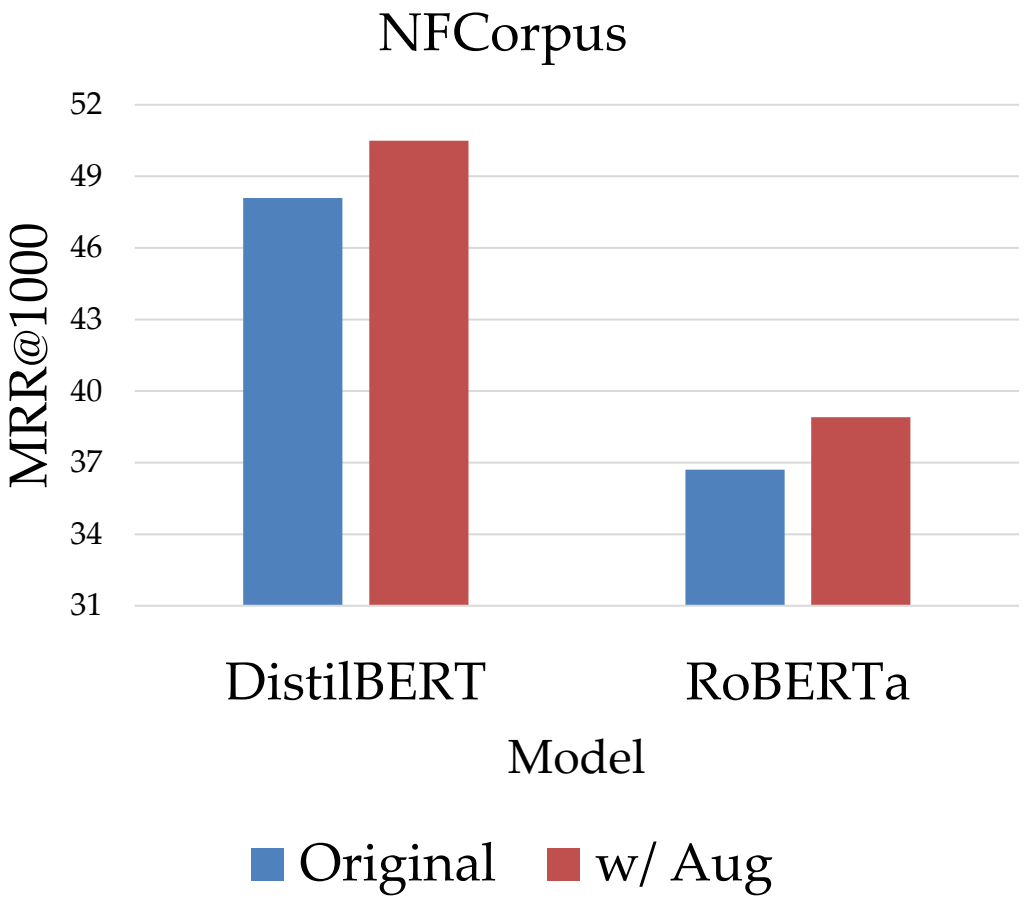


Augmentation makes code representations more evenly distributed across the entire space.

Easier to distinguish representations.

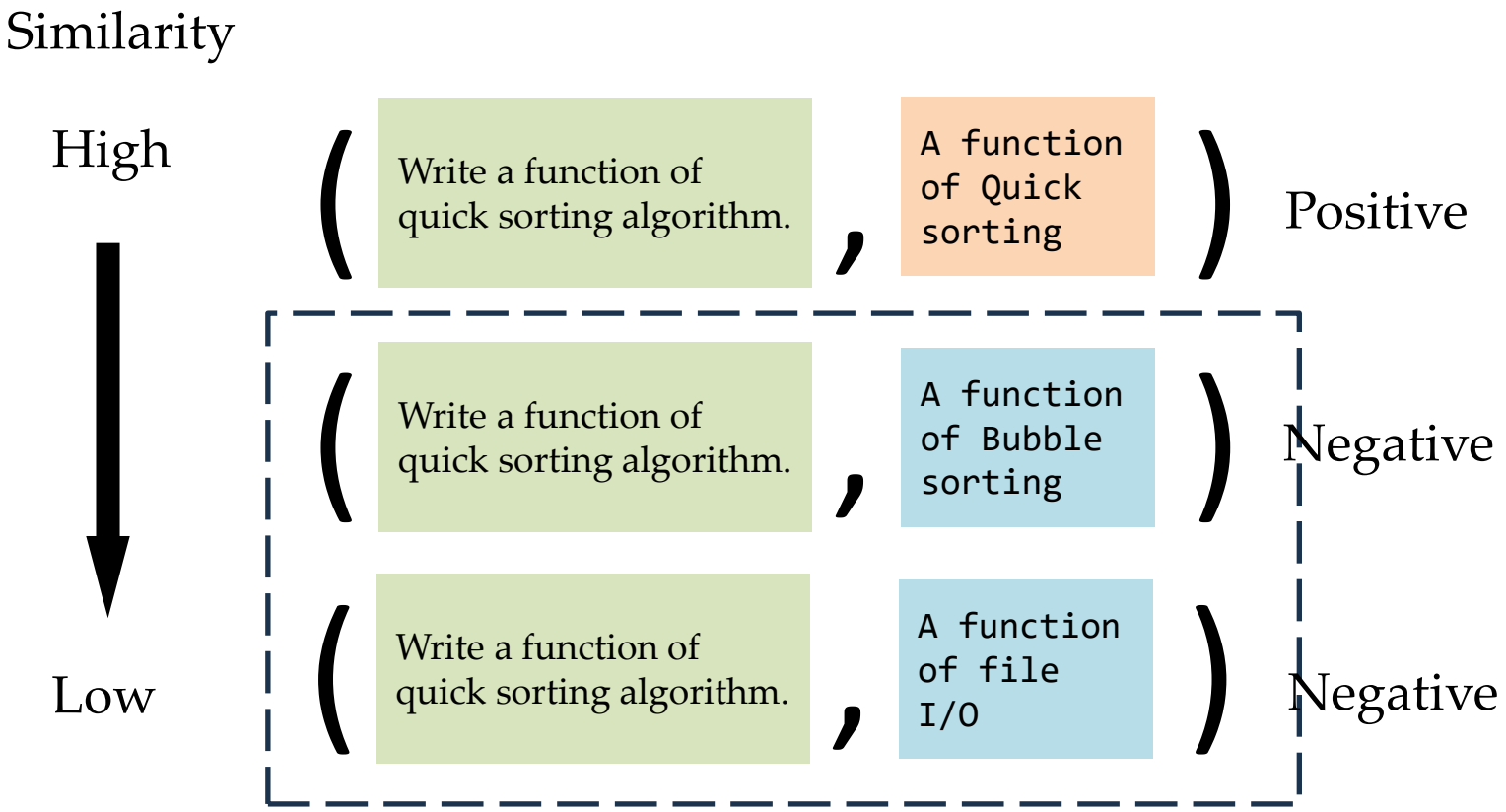
Representation-level Augmentation also benefits Passage Retrieval





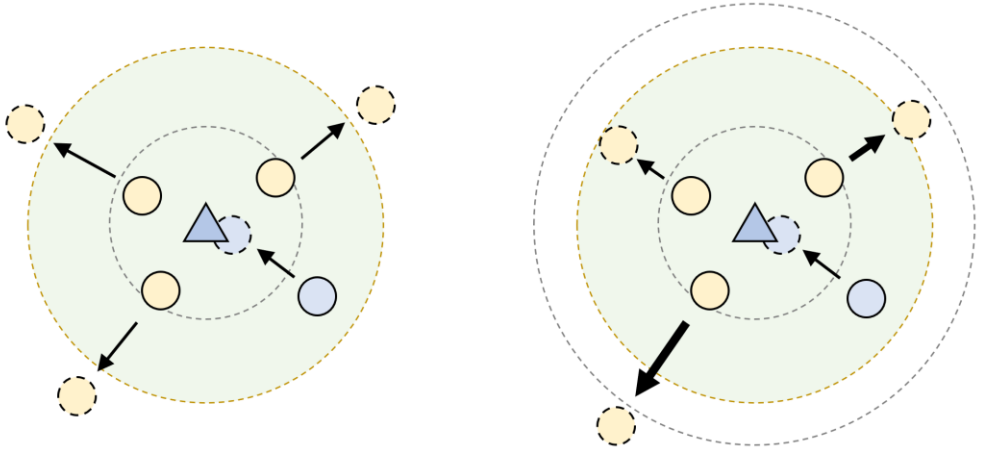
Let's turn our sight to the negative pair side

We expect…



But this relative relationship is not modelled by InfoNCE because it treats all negative codes equally.

Negative pairs should not be treated equally



Soft-InfoNCE

$$\mathcal{L} = -\mathbb{E}_i \left[\log \frac{\exp(text_i \cdot code_i)}{\sum_j \exp(text_i \cdot code_j)} \right] \quad \Rightarrow \quad \mathcal{L} = -\mathbb{E}_i \left[\log \frac{\exp(text_i \cdot code_i)}{\sum_j \lambda_{ij} \exp(text_i \cdot code_j)} \right]$$

How could we estimate λ_{ij}? BM25, SimCSE, Fine-tuned Code Search Models…

Li, et al. “Rethinking Negative Pairs in Code Search." EMNLP 2023.

Soft-InfoNCE controls the distribution of negative samples

- **Theorem 1** For a batch of texts $\{t_i\}_{i=1}^N$, codes $\{c_i\}_{i=1}^N$, and similarity scores $\{S_{ij}\}$ ($S_{ij} \propto \lambda_{ij}$), we have:

$$\mathcal{L}_{unif} \geq \frac{1}{N(N-2)} \sum_{i=1}^{N} \left[\sum_{j \neq i}^{N} \log P_\theta(c_j|t_i) + KL(S_{ij}|P_\theta(c_j|t_i)) \right] + const.$$

Where N is the batch size, $P_\theta(c_j|t_i)$ is predicted similarity between text t_i and code c_j by model θ.

While $L_{unif} \searrow$, it encourages $KL(S_{ij}|P_\theta(c_j|t_i)) \searrow$
the model predicted similarity $P_\theta(c_j|t_i)$ gets closer to the given similarity.

Soft-InfoNCE reduces bias in mutual information estimation

$$\mathcal{L}_{\text{InfoNCE}} = -\mathbb{E}\log\left[\frac{\frac{p(c_i|t_i)}{p(c_i)}}{\frac{p(c_i|t_i)}{p(c_i)} + \sum_{j\neq i}^{N}\frac{p(c_j|t_i)}{p(c_j)}}\right]$$

$$= \mathbb{E}\log\left[1 + \frac{p(c_i)}{p(c_i|t_i)}\sum_{j\neq i}^{N}\frac{p(c_j|t_i)}{p(c_j)}\right]$$

$$\approx \mathbb{E}\log\left[1 + \frac{p(c_i)}{p(c_i|t_i)}(N-1)\mathbb{E}_{c_j\in C_{neg}}\frac{p(c_j|t_i)}{p(c_j)}\right]$$

$$\geq \mathbb{E}\log\left[\frac{p(c_i)}{p(c_i|t_i)}N\right]$$

$$= -I(t_i, c_i) + \log N$$

Monte Carlo estimation for mutual information of all negative pairs C_{neg} based on the negative pairs in a batch

$$\mathbb{E}\log\left[1 + \frac{p(c_i)}{p(c_i|t_i)}\sum_{j\neq i}^{N}\lambda_{ij}\frac{p(c_j|t_i)}{p(c_j)}\right]$$

By inserting λ_{ij}, it could be considered as using importance sampling to reduce estimation bias.

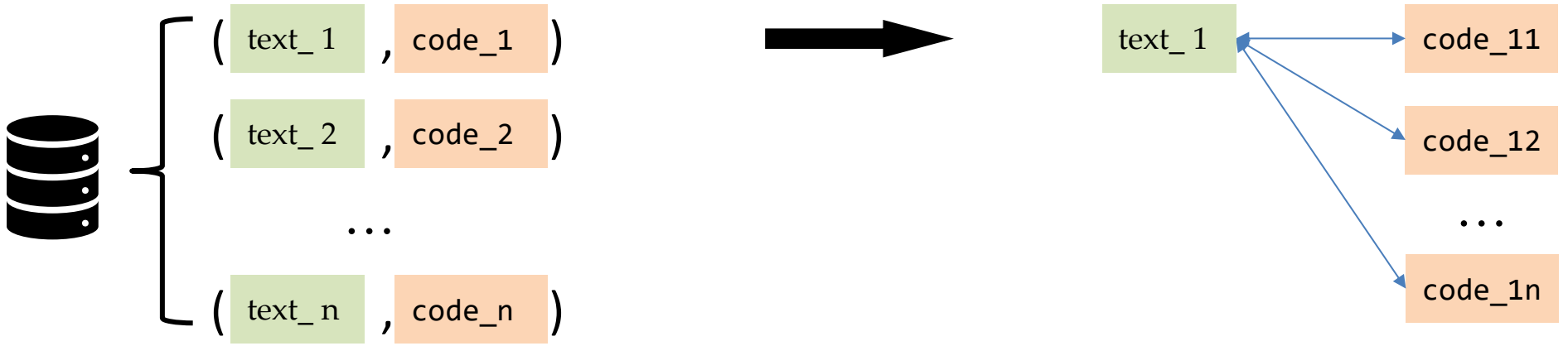
Soft-InfoNCE outperforms InfoNCE

Model	Loss	Estimator	Ruby	Python	Java	JavaScript	PHP	Go
CodeBERT	InfoNCE	-	64.8	63.6	66.3	59.4	61.5	87.8
	Soft-InfoNCE	BM25	66.0	66.4	**68.2**	60.0	62.1	88.2
		Trained Model	**68.2**	**67.5**	**68.2**	**61.5**	**63.1**	**89.8**
		SimCSE	66.6	66.8	67.0	60.7	62.3	88.7
GraphCodeBERT	InfoNCE	-	70.5	69.0	69.1	64.7	64.8	**89.6**
	Soft-InfoNCE	BM25	**73.0**	69.7	69.8	65.2	65.5	89.2
		Trained Model	71.9	69.2	69.2	65.3	64.8	88.9
		SimCSE	72.1	**70.0**	**70.2**	**65.6**	**65.7**	89.4
UniXCoder	InfoNCE	-	74.0	72.0	72.6	68.4	67.6	91.5
	Soft-InfoNCE	BM25	**75.3**	**72.8**	**73.3**	69.3	**68.4**	**91.6**
		Trained Model	**75.3**	**72.8**	73.1	69.4	68.2	91.5
		SimCSE	**75.3**	72.6	73.1	**69.9**	**68.4**	91.3

Overview of My Research

- One-to-One Mapping
  - How can we align representations of natural language and programming language better, given a dataset that consists of NL-code pairs?
- One-to-Many Mapping
  - In fact, an NL description often corresponds to multiple code implementations. Considering this, how can we further enhance semantic alignment?
- Many-to-Many Mapping
  - Similarly, a code snippet corresponds to multiple NL descriptions. Can we create high-quality synthetic data by leveraging this relationship?

Moving beyond One-to-One Mapping



In the first part, we discuss how to align representations better given NL-code pairs that are crawled from the Web (e.g. GitHub).

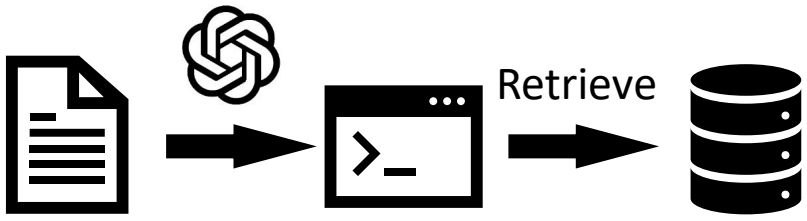
There are many semantic-equivalent code variations. It is hard to collect all variations.

Ignoring this one-to-many relationship may not be optimal.

In the first part, we try to directly align natural language and programming language.

Some works translate natural language to programming language before calculating representation similarity.

Vanilla Code Search



Generation-Augmented Retrieval (GAR)

GAR sometimes fails for code search

Text:
Write a function to get the frequency of the elements in a list.

*Count the frequency manually

Exemplar Code:

```
def count_frequency(my_list):
    frequency = {}
    for element in my_list:
        if element not in frequency:
            frequency[element] = 0
        frequency[element] += 1
    return frequency
```

*Count the frequency automatically

True Code:

```
import collections
def freq_count(list1):
    freq_count = \
        collections.Counter(list1)
    return freq_count
```

LMs cannot recognize the similarity between the exemplar code and the true code due to their implementation style.

Li, et al. "Rewriting the Code: A Simple Method for Large Language Model Augmented Code Search." ACL 2024.

Our Solution: rewrite the code (ReCo)

Text:
Write a function to get the frequency of the elements in a list.

*Count the frequency manually

Exemplar Code:
```
def count_frequency(my_list):
    frequency = {}
    for element in my_list:
        if element not in frequency:
            frequency[element] = 0
        frequency[element] += 1
    return frequency
```

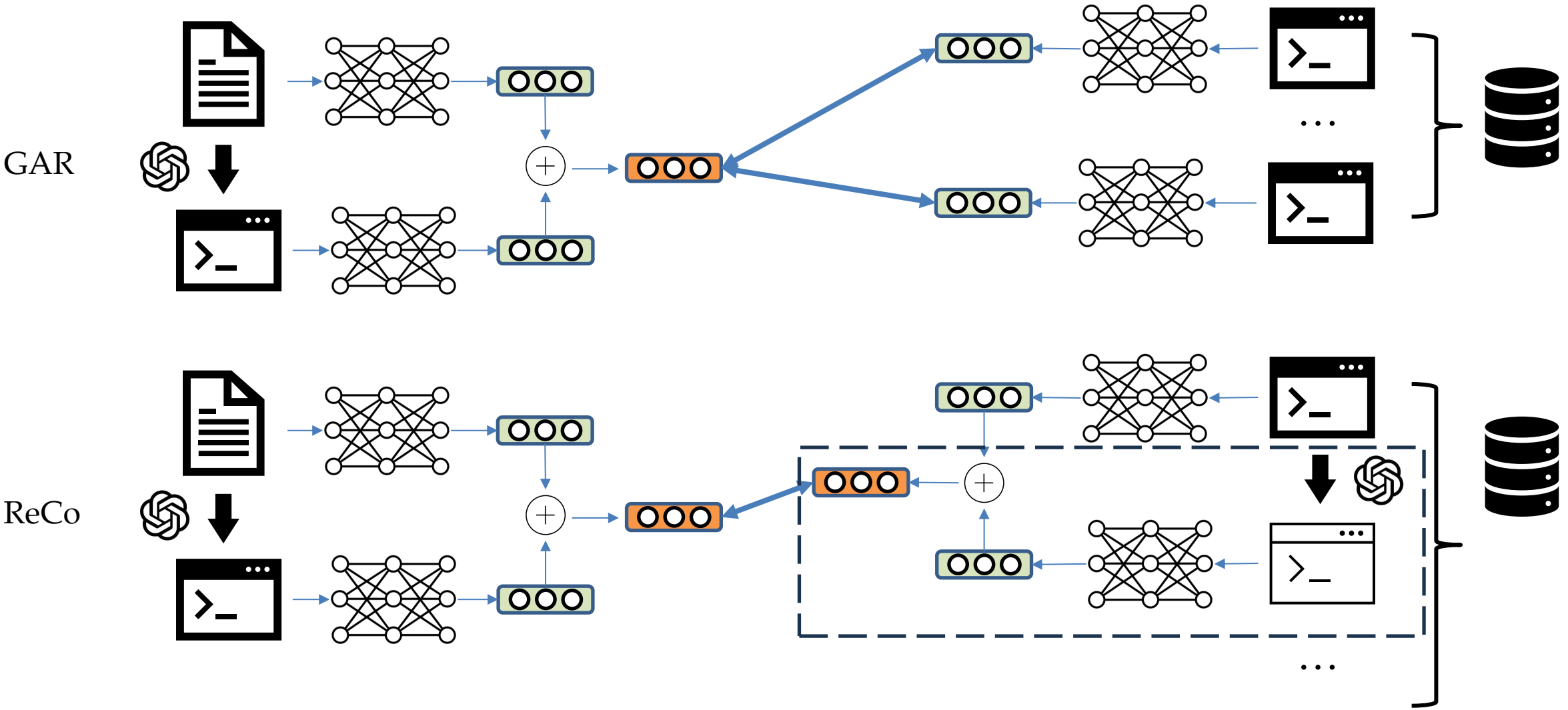
*Count the frequency automatically

True Code:
```
import collections
def freq_count(list1):
    freq_count = \
        collections.Counter(list1)
    return freq_count
```

Must use the same LLM!

Rewritten Code:
```
def frequency(my_list):
    freq = {}
    for i in my_list:
        if i in freq:
            freq[i] += 1
        else:
            freq[i] = 1
    return freq
```

Framework Overview



Is the style mismatch problem in GAR a coincidence?

- Original code in the dataset is an implementation of the text in the dataset.

$$\text{code} \sim P_{real}(\cdot \mid \text{text})$$

P_{real} refers to the real-world distribution.

- Consider the exemplar code generation process as a sampling process (LLM the sampler).

$$\text{exemplar code} \sim P_{LLM}(\cdot \mid \text{text})$$

LLM also defines a probability distribution P_{LLM}.

Once there is a gap between P_{real} and P_{LLM}, the style mismatch is inevitable.

Why ReCo works?

- Consider the exemplar code generation process as a sampling process (LLM the sampler).

$$\text{exemplar code} \sim P_{LLM}(\cdot \mid \text{text})$$

  LLM also defines a probability distribution P_{LLM}.

- We rewrite the code in the codebase through a code->summary->rewritten code process.

$$\text{rewritten code} \sim P_{LLM}(\cdot \mid \text{summary})$$

  If the summary is perfect (summary≈text):

$$\text{rewritten code} \sim P_{LLM}(\cdot \mid \text{text})$$

Experiment results

	CoNaLa	MBPP	APPS	MBJP
Unsupervised				
BM25	52.6	12.6	11.6	11.3
+ GAR	71.7	35.1	17.6	33.5
+ ReCo	**75.8**$_{+4.1}$	**70.8**$_{+35.7}$	**22.6**$_{+5.0}$	**65.3**$_{+31.8}$
UniXcoder	77.2	69.3	8.3	73.2
+ GAR	83.9	85.0	13.2	80.0
+ ReCo	**85.1**$_{+1.2}$	**92.4**$_{+7.4}$	**28.8**$_{+15.6}$	**87.6**$_{+7.6}$
Contriever	55.7	55.3	9.6	37.0
+ GAR	75.0	71.3	14.0	62.3
+ ReCo	**77.9**$_{+2.9}$	**87.4**$_{+16.1}$	**41.6**$_{+27.6}$	**76.6**$_{+14.3}$
CodeT5+	73.7	59.4	7.6	67.7
+ GAR	80.3	77.7	10.2	79.2
+ ReCo	**80.8**$_{+0.5}$	**89.4**$_{+11.7}$	**29.9**$_{+19.7}$	**84.0**$_{+4.8}$

	CoNaLa	MBPP	APPS	MBJP
Supervised				
CodeBERT	83.6	79.6	25.1	79.6
+ GAR	**88.6**	87.7	29.3	84.1
+ ReCo	85.0$_{-3.6}$	**92.3**$_{+4.6}$	**51.2**$_{+21.9}$	**89.1**$_{+5.0}$
UniXcoder	84.8	81.2	24.3	81.6
+ GAR	85.9	89.0	34.5	85.6
+ ReCo	**87.1**$_{+1.2}$	**94.2**$_{+5.2}$	**58.1**$_{+23.6}$	**90.5**$_{+4.9}$

GAR indeed improves code search, but ReCo improves more.

(non-neural model, zero-shot, fine-tune)

Experiment results

	CoNaLa	MBPP	APPS	MBJP
Unsupervised				
BM25	52.6	12.6	11.6	11.3
+ GAR	71.7	35.1	17.6	33.5
+ ReCo	**75.8**$_{+4.1}$	**70.8**$_{+35.7}$	**22.6**$_{+5.0}$	**65.3**$_{+31.8}$
UniXcoder	77.2	69.3	8.3	73.2
+ GAR	83.9	85.0	13.2	80.0
+ ReCo	**85.1**$_{+1.2}$	**92.4**$_{+7.4}$	**28.8**$_{+15.6}$	**87.6**$_{+7.6}$
Contriever	55.7	55.3	9.6	37.0
+ GAR	75.0	71.3	14.0	62.3
+ ReCo	**77.9**$_{+2.9}$	**87.4**$_{+16.1}$	**41.6**$_{+27.6}$	**76.6**$_{+14.3}$
CodeT5+	73.7	59.4	7.6	67.7
+ GAR	80.3	77.7	10.2	79.2
+ ReCo	**80.8**$_{+0.5}$	**89.4**$_{+11.7}$	**29.9**$_{+19.7}$	**84.0**$_{+4.8}$

	CoNaLa	MBPP	APPS	MBJP
Supervised				
CodeBERT	83.6	79.6	25.1	79.6
+ GAR	**88.6**	87.7	29.3	84.1
+ ReCo	85.0$_{-3.6}$	**92.3**$_{+4.6}$	**51.2**$_{+21.9}$	**89.1**$_{+5.0}$
UniXcoder	84.8	81.2	24.3	81.6
+ GAR	85.9	89.0	34.5	85.6
+ ReCo	**87.1**$_{+1.2}$	**94.2**$_{+5.2}$	**58.1**$_{+23.6}$	**90.5**$_{+4.9}$

With ReCo, non-neural model BM25 is comparable to neural models under the zero-shot setting.

Experiment results

	CoNaLa	MBPP	APPS	MBJP
Unsupervised				
BM25	52.6	12.6	11.6	11.3
+ GAR	71.7	35.1	17.6	33.5
+ ReCo	**75.8**$_{+4.1}$	**70.8**$_{+35.7}$	**22.6**$_{+5.0}$	**65.3**$_{+31.8}$
UniXcoder	77.2	69.3	8.3	73.2
+ GAR	83.9	85.0	13.2	80.0
+ ReCo	**85.1**$_{+1.2}$	**92.4**$_{+7.4}$	**28.8**$_{+15.6}$	**87.6**$_{+7.6}$
Contriever	55.7	55.3	9.6	37.0
+ GAR	75.0	71.3	14.0	62.3
+ ReCo	**77.9**$_{+2.9}$	**87.4**$_{+16.1}$	**41.6**$_{+27.6}$	**76.6**$_{+14.3}$
CodeT5+	73.7	59.4	7.6	67.7
+ GAR	80.3	77.7	10.2	79.2
+ ReCo	**80.8**$_{+0.5}$	**89.4**$_{+11.7}$	**29.9**$_{+19.7}$	**84.0**$_{+4.8}$

	CoNaLa	MBPP	APPS	MBJP
Supervised				
CodeBERT	83.6	79.6	25.1	79.6
+ GAR	**88.6**	87.7	29.3	84.1
+ ReCo	85.0$_{-3.6}$	**92.3**$_{+4.6}$	**51.2**$_{+21.9}$	**89.1**$_{+5.0}$
UniXcoder	84.8	81.2	24.3	81.6
+ GAR	85.9	89.0	34.5	85.6
+ ReCo	**87.1**$_{+1.2}$	**94.2**$_{+5.2}$	**58.1**$_{+23.6}$	**90.5**$_{+4.9}$

With ReCo, neural models under the zero-shot setting is comparable to them after fine-tuning.

A new metric to measure code style similarity

- Code Style Similarity considers similarity of style from three perspectives:
  - Variable Naming (based on Edit Distance)
  - API Invocation (based on Edit Distance)
  - Code Structure (based on Tree Edit Distance of Abstract Syntax Trees)

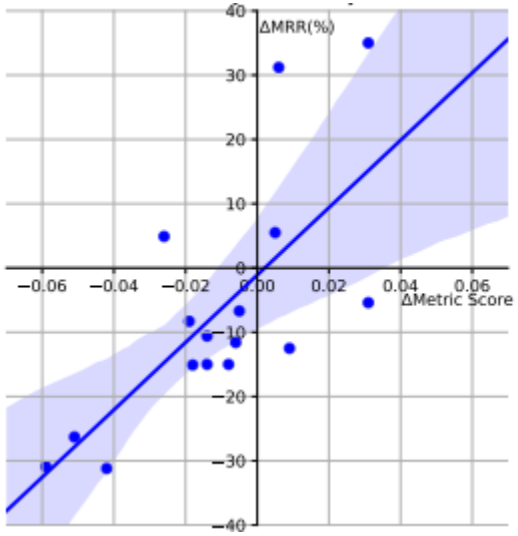
$$\mathrm{CSSim}(c_1, c_2) = 1 - \frac{1}{3}(\mathrm{Dis}_{\mathrm{Var}} + \mathrm{Dis}_{\mathrm{API}} + \mathrm{TED})$$

$$\mathrm{Dis} = \frac{1}{2}\left(\frac{1}{||\lambda||_1}\sum_{v_i \in \mathbf{S}_1} \lambda_i \min_{v_j \in \mathbf{S}_2} \mathrm{ED}(v_i, v_j) + \frac{1}{||\lambda||_1}\sum_{v_i \in \mathbf{S}_2} \lambda_i \min_{v_j \in \mathbf{S}_1} \mathrm{ED}(v_i, v_j)\right)$$

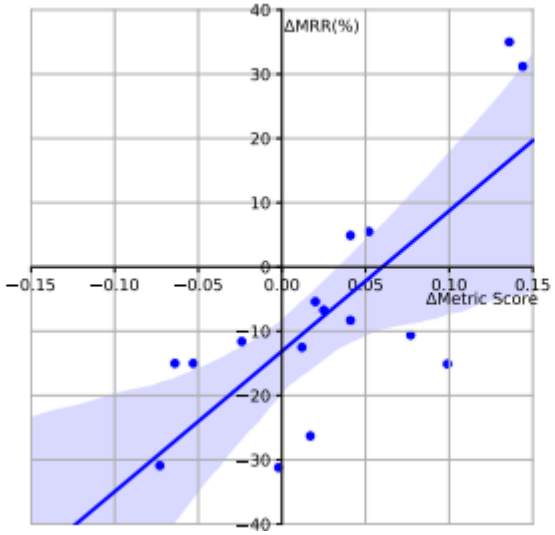
ED: Edit Distance
TED: Tree Edit Distance of Abstract Syntax Tree
λ_i: Normalized Inverse Document Frequency (IDF)
S_1, S_2: Set of extracted variables or APIs

Our metric is better than existing metrics

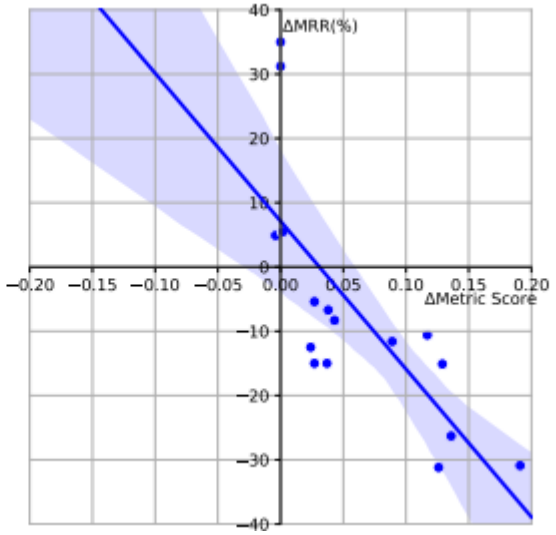
Code Style Similarity



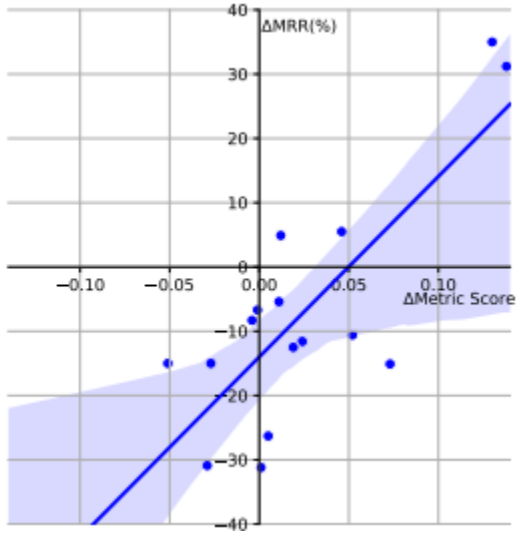
CodeBLEU



ROUGE-L



BLEU



The improvement of Code Style Similarity has stronger correlation with the performance improvement of ReCo over GAR.

Our metric is better than existing metrics

Dataset	Random	w/ the best exemplar code			
		CSSim	CodeBLEU	ROUGE-L	BLEU
CoNaLa	41.3	**43.6**	39.6	41.5	42.3
MBPP	24.0	**26.8**	25.1	23.7	24.0
APPS	14.1	**15.2**	14.8	14.2	14.2
MBJP	26.6	**29.9**	29.1	27.2	28.8

We select the best exemplar code (the most similar to true code) under different metrics for BM25+ GAR.

The Code Style Similarity outperforms other settings.

True Code

```
def find_first_duplicate(nums):
    num_set = set()
    no_duplicate = -1
    for i in range(len(nums)):
        if nums[i] in num_set:
            return nums[i]
        else:
            num_set.add(nums[i])
    return no_duplicate
```

Exemplar Code 1

```
def find_duplicate(nums):
    for i in range(len(nums)):
        if nums[i] in nums[i+1:]:
            return nums[i]
    return None
```

Preferred by other metrics

Exemplar Code 2

```
def find_duplicate(my_list):
    seen = set()
    for num in my_list:
        if num in seen:
            return num
        seen.add(num)
    return None
```

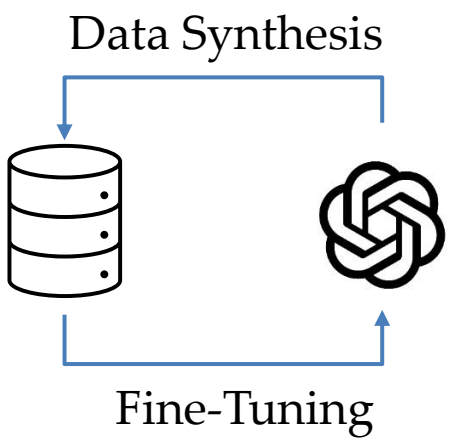
Preferred by our metric

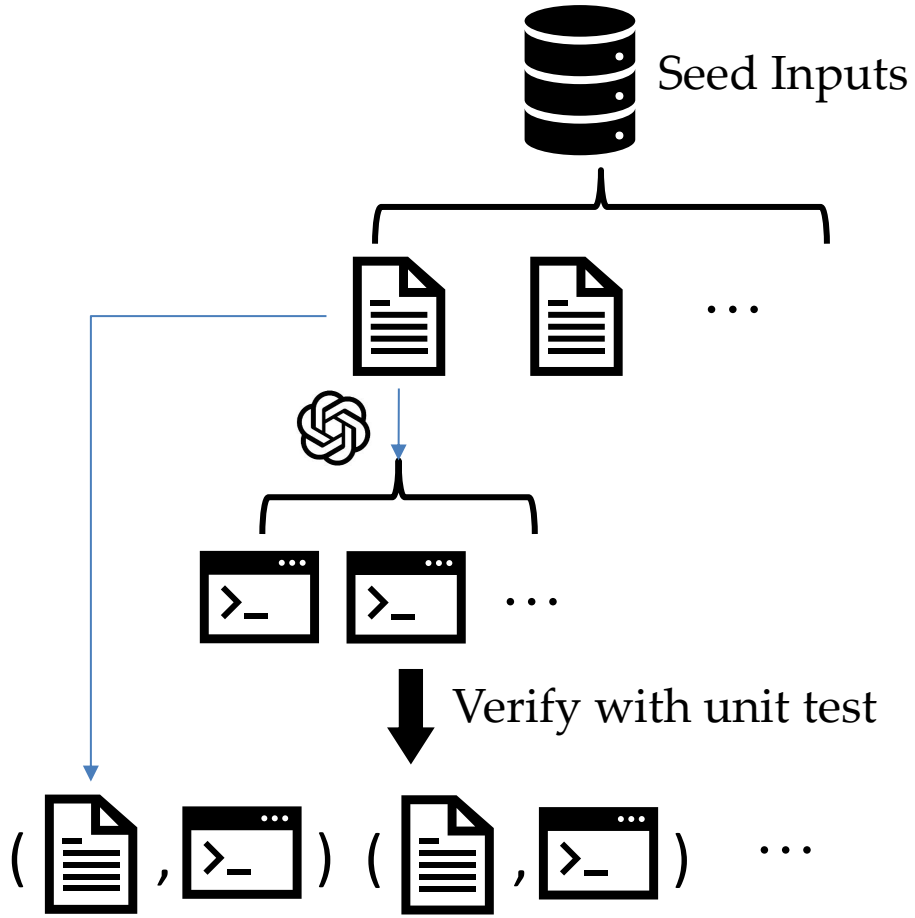
Overview of My Research

- One-to-One Mapping
  - How can we align representations of natural language and programming language better, given a dataset that consists of NL-code pairs?
- One-to-Many Mapping
  - In fact, an NL description often corresponds to multiple code implementations. Considering this, how can we further enhance semantic alignment?
- Many-to-Many Mapping
  - Similarly, a code snippet corresponds to multiple NL descriptions. Can we create high-quality synthetic data by leveraging this relationship?

Iterative Rejection Fine-Tuning

Overview:





There is bias in data synthesis process

For each seed input, it is a specific expression of the intent behind the input.

Intent:

- I want a function for substring matching.

Possible inputs 1:

- Write a function to search a string for a regex pattern.

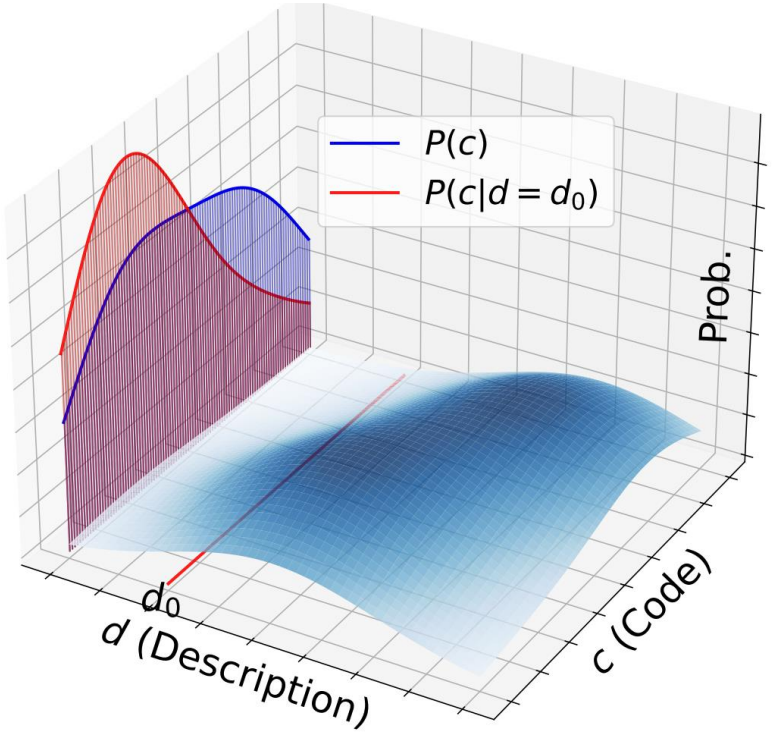
Possible input 2:

- Find a substring within a larger string that matches a given regular expression pattern.

There is bias in data synthesis process

For each seed input, it is a specific expression of the intent behind the input.

For each intent, suppose all possible inputs and valid codes form a joint space.



If we synthesize codes solely based on each seed input d_0, it is drawing codes from a **conditional distribution** (Red).

It is better to sample codes from the **marginal distribution** (Blue).

Li, et al. “GiFT: Gibbs Fine-Tuning for Code Generation." ACL 2025.

Why is marginal distribution better?

- The loss $\mathcal{L}_{marg}$ is implicitly estimated over all possible $\mathcal{L}_{cond}$.

$$\mathcal{L}_{marg} = -\mathbb{E}_{d\sim P(input)}\,\mathcal{L}_{cond}(d)$$

- The variance (diversity) of code is higher.

$$Var(c) = \mathbb{E}_d[Var(c|d)] + Var(\mathbb{E}_d[c|d])$$

Expected variance (diversity) of codes from conditional distribution

Variance of codes from different inputs

How can we sample from marginal distribution?

- One possible solution is to ask LLMs to rewrite the seed input.
  - The quality of rewritten inputs is not satisfiable.
  - Rewritten inputs are still similar.

Datasets	RD		GiFT
	Seed	Rewritten	
APPS+ (Intro.)	17.79	4.8	>10.31
APPS+ (Inter.)	3.22	0.38	>2.28
MBPP+	53.71	24.6	>24.07
CodeInsight	43.87	9.84	>24.18

Pass rate (%) of self-generated codes from the seed description, rewritten description, and GiFT.

- We got inspiration from Gibbs Sampling, commonly used to approximate joint distributions based on conditional distributions.

Introduction of Gibbs Sampling

Take a two-variable joint distribution as an example.

We want to sample from $P(x, y)$, but we only have access to $P(x|y)$ and $P(y|x)$.

Starting from x_0

For t =1 to N do:

$$y_t \sim P(y|x_{t-1})$$

$$x_t \sim P(x|y_t)$$

Collected pairs $\{(x_1, y_1)(x_2, y_2) \ldots (x_N, y_N)\}$ can be considered as being sampled from the joint distribution.

Introduction of Gibbs Sampling

Gibbs sampling

We want to sample from $P(x, y)$, but we only have access to $P(x|y)$ and $P(y|x)$.

Starting from x_0

For t =1 to N do:

$y_t \sim P(y|x_{t-1})$

$x_t \sim P(x|y_t)$

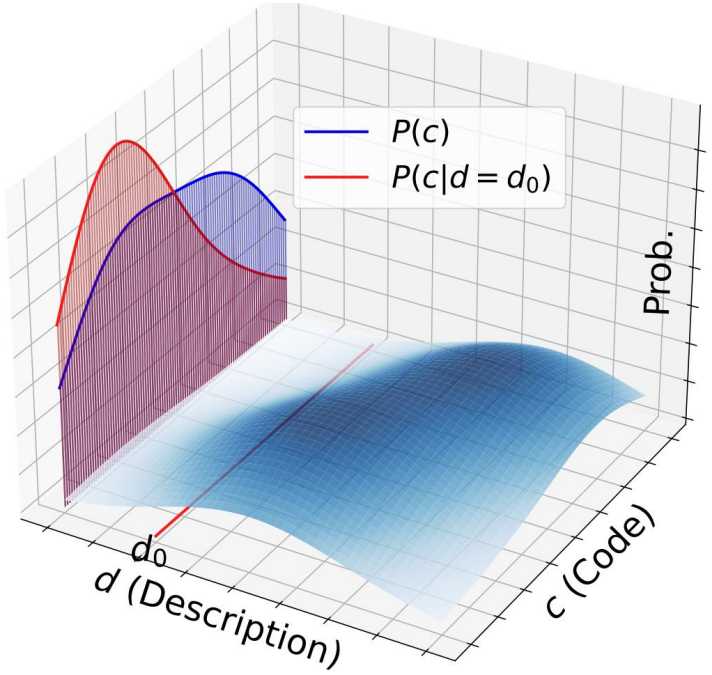
In our context

We want to sample from $P(\text{text}, \text{code})$.
We have access to:
$P(\text{text}|\text{code})$ ---- Code Summarization
$P(\text{code}|\text{text})$ ---- Code Generation

Starting from the seed input text_0

Collected pairs
$\{(\text{text}_1, \text{code}_1) \ldots (\text{text}_N, \text{code}_N)\}$

Curating codes for fine-tuning



The marginal distribution (Blue) often does not follow a uniform distribution.

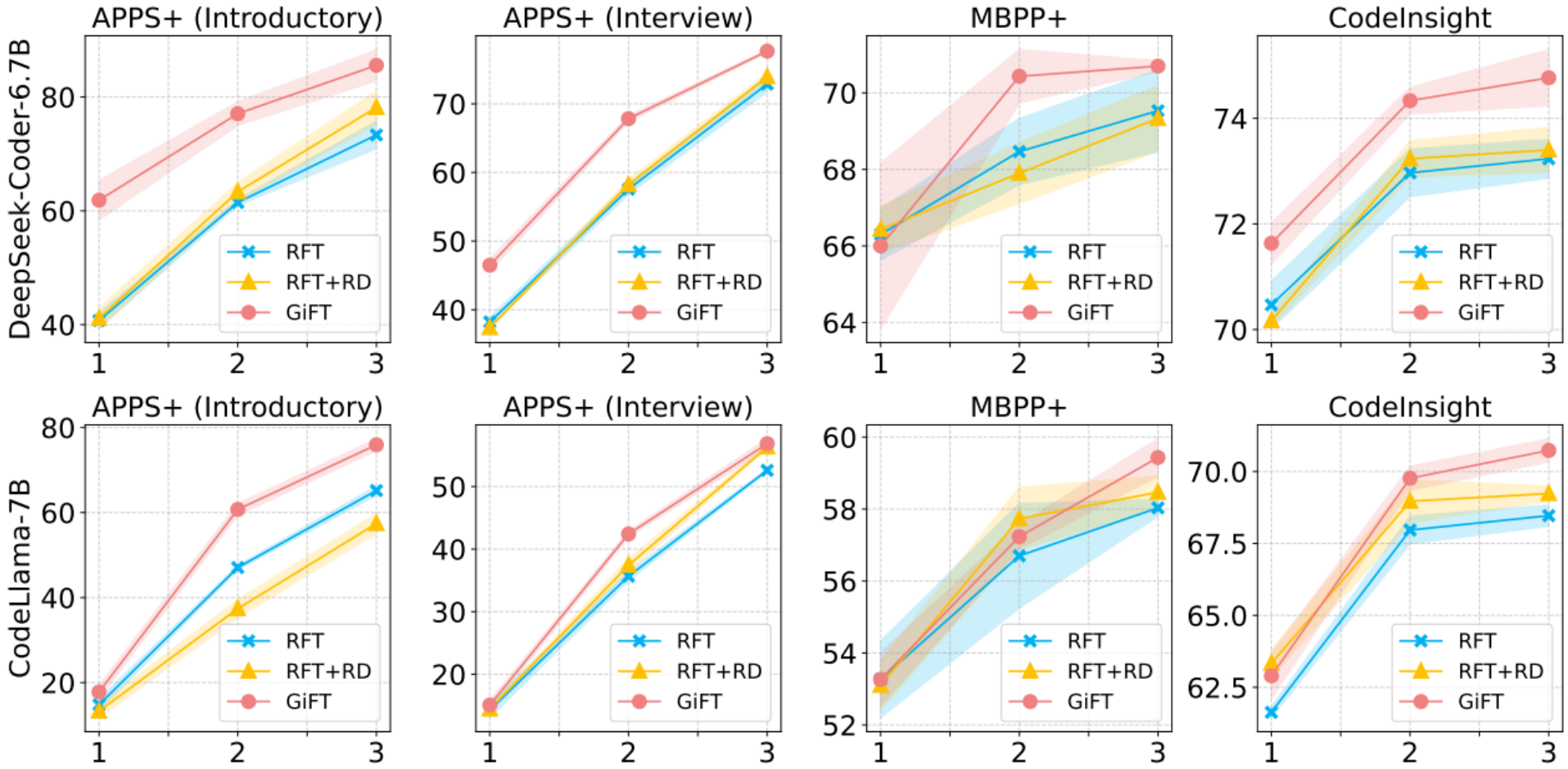
Using all correct codes for fine-tuning:

1. Bias towards easier inputs
2. Bias towards head in long-tail distribution

Solutions:

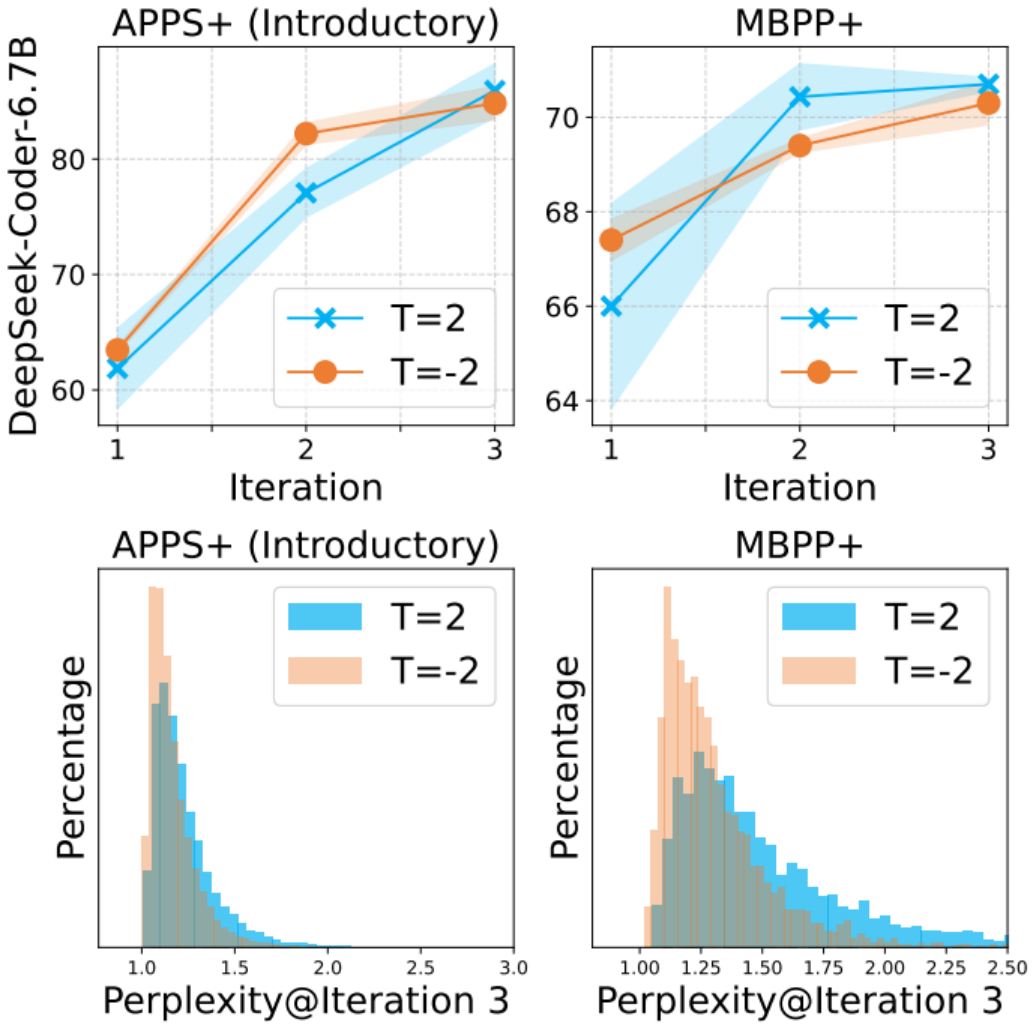
1. At most K codes per inputs
2. Weighted random sampling based on perplexity to encourage selecting tail codes

Experiment results



RFT refers to Rejection-Fine-tuning, RFT+RD refers to RFT with rewritten texts

Impact of Perplexity-guided Data Selection



T > 0: Encourage Tail

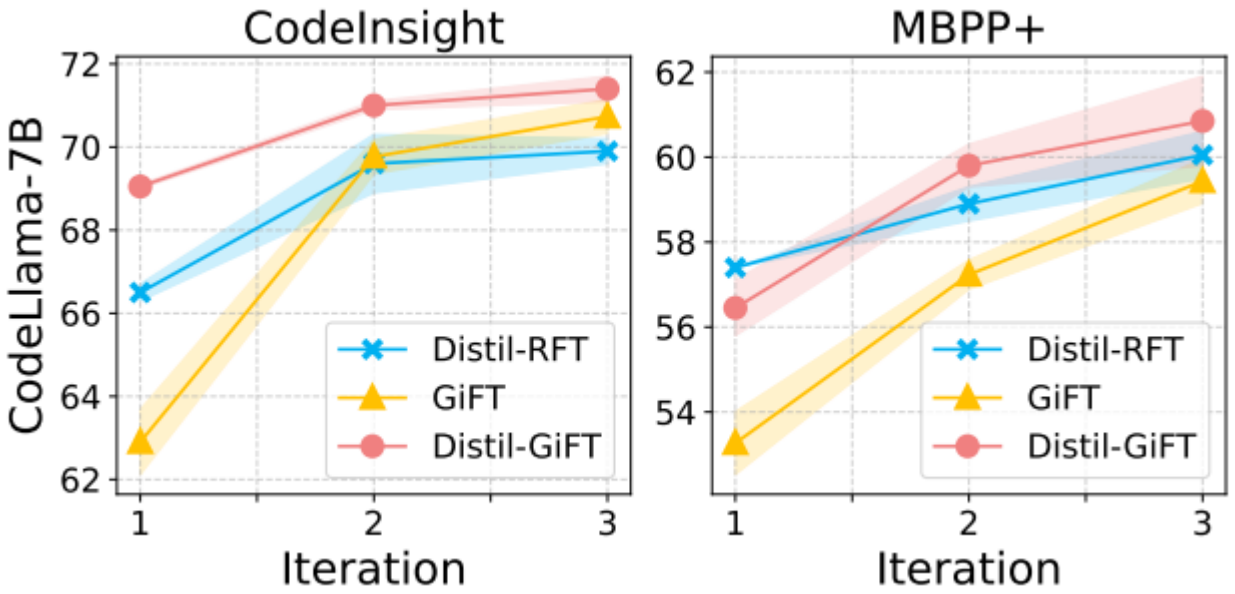
T < 0: Encourage Head

Encouraging head speeds up convergence, yet its potential is lower than encouraging tail.

Perplexity distribution of self-generated codes.

Encouraging head makes the distribution sharper.

GiFT benefits not only self-training, but also distillation

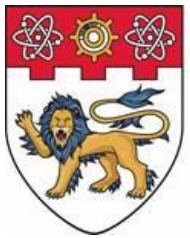


Distill DeepSeek-Coder-6.7B to CodeLlama-7B

Prerequisites of applying GiFT to other tasks?

- LLMs could seamlessly translate between inputs and outputs.
  - Counter example: Math Reasoning
  - Translate solutions back to math problems?

- There is indeed a joint input-output space.
  - Counter example: Factual Question Answering



NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

Thank you!
Any Questions?