# テキスト変換によるリポジトリレベルのコード検索の改善

近藤瑞希 [1]　河原大輔 [1]　倉林利行 [2]
[1] 早稲田大学　[2] 日本電信電話株式会社
{kondmiznotfound@toki., dkw@}waseda.jp　toshiyuk.kurabayashi@ntt.com

## 概要

大規模言語モデルによるコード生成能力は年々上昇しているが、リポジトリレベルでのコード生成の研究はあまり活発ではない。リポジトリレベルのコード生成では複数ファイルの中から関連するのコードの参照が必要となる。コード間の類似度を取ることで関連するファイルを検索し言語モデルに入力し、生成を行う。本研究では関連するファイルの検索 (コード検索) を、コード間の類似度ではなく、コードを言語モデルで変換したテキストで類似度を取る手法を提案する。テキストに変換することでコード検索の精度が向上することを確認した。

## 1 はじめに

日本でのIT人材は不足しており、2030年には最大で79万人不足するという試算もある[1)]。その解決法の一つとして、大規模言語モデル (LLM) を用いたソフトウェアの開発の支援が挙げられる。現在、LLMによるコード生成能力は飛躍的に上昇しており、単一のコードの理解や生成の精度は高い。しかし、実際のソフトウェア開発に近いリポジトリレベルでの研究は少なく、LLMによるリポジトリレベルのコード生成の能力は非常に低い。リポジトリレベルでのバグ修正ベンチマーク SWE-bench [1] のバグ解決率はわずか1.96%にとどまる。

リポジトリレベルでのコード関連タスクでは多くのファイルの参照が必要となる。しかし、LLMのほとんどは Transformer [2] ベースのモデルであるため、入力長に制限があり、多くのファイルを入力できない。そのため、類似度などを基に関連性の高いコードを検索し、それらのみを入力する手法が提案されている。コード検索の精度は、SWE-bench やリポジトリレベルのコード補完・検索ベンチマークである RepoBench [3] などでも高くない。

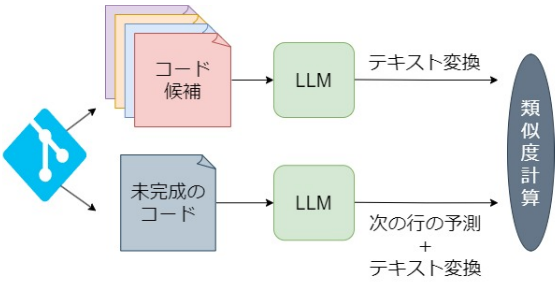


図 1: 提案手法のテキスト変換

本研究では、コード補完タスクにおけるコード検索手法の改善に取り組む。コード補完タスクは、未完成のコードの次の行を予測するタスクである。次の行の予測のために、未完成のコードを構文解析することで得られるファイル間の依存関係をもとに、複数の関連ファイルが与えられる。本論文では未完成のコードを対象コードと呼び、複数の関連ファイルをコード候補と呼ぶ。対象コードの情報を基に、コード候補から正しいコードを選択するタスクをコード検索と呼ぶ。

既存研究ではコード検索のために、対象コードおよびコード候補のそれぞれの特徴量 (言語モデルの埋め込み表現など) を取得し類似度を計算している [3]。RepoCoder [4] では、このような手法でコード検索し、一度コードの生成を行い、生成されたコードを含めて再検索し生成する手法をとる。本研究では、直接コード間の類似度を取らずに、LLM によってコードを変換した後に類似度を計算する。候補コードはテキストに変換し、対象コードはテキストへの変換や、RepoCoder の手法とテキスト変換を組み合わせた、次の行の予測とその説明などに変換を行う。図 1 にテキスト変換の流れを示す。

本手法によって、コード補完タスクにおけるコード検索の精度が向上することを確認した。また、LLM でテキスト変換を行うプロンプトについても検証を行った。

---

[1)] https://www.meti.go.jp/policy/it_policy/jinzai/houkokusyo.pdf

# 2 関連研究

## 2.1 コードで学習された LLM

近年、コードで学習された Transformer [2] ベースの LLM が増えている。Encoder モデルでは CodeBERT [5] や UniXcoder [6], Decoder モデルでは Codex [7], StarCoder [8], Code Llama [9], Encode-Decoder モデルでは, CodeT5+ [10] などが挙げられる。特に Decoder モデルの発展は著しく、コード生成能力は飛躍的に上昇した。これらのモデルは単一のコード生成精度が高く、HumanEval [7] や MBPP [11] などのベンチマークでの精度が高い。

## 2.2 リポジトリレベルの研究

ソフトウェア開発では単一ファイルではなく複数のファイルを用いて開発を行う。現実のソフトウェア開発のタスクに対応するために、GitHub などのリポジトリを対象として研究が行われてきた [12]。

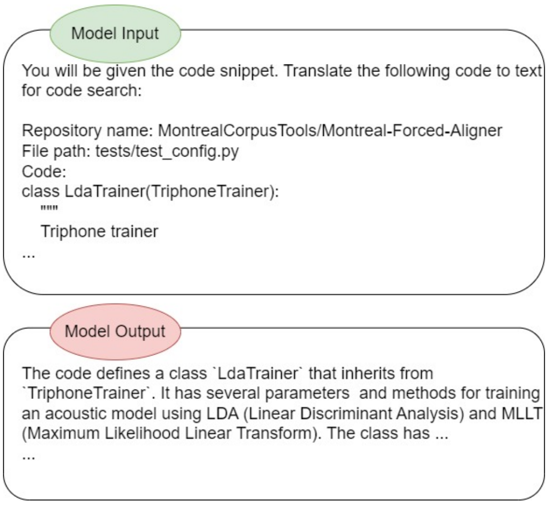
LLM によるコード生成能力の向上により、LLM によるリポジトリレベルでの研究やベンチマークが増加している [1, 3, 4, 13, 14]。RepoCoder [4] は、コード補完タスクにおいて検索と生成を 2 回繰り返すことで精度の向上させた。RepoBench [3] は、コード補完に関するベンチマークで、コード検索タスク、コード補完タスク、2 つのパイプラインタスクの 3 つで構成されている。SWE-bench [1] は GitHub issues と対応するプルリクエストを Python のリポジトリから集めたもので、現実の問題を LLM がどれくらい解けるかを競うベンチマークである。

HumanEval [7] や MBPP [11] と比べると、RepoBench [3] や SWE-bench [1] の精度は著しく低い。単一コードによる研究と比較して、リポジトリレベルでの研究は手法やデータセットの数か少なく発展途上である。

# 3 提案手法

## 3.1 概要

コード検索の従来手法では、対象コードとコード候補をそのまま言語モデル[2]に入力し、埋め込み表現からコード間の類似度を計算している [3]。提案手法では、LLM によってコードをテキストに変換し、言語モデルを用いてその埋め込み表現を得る。類似度の計算は、埋め込み表現の平均の cosine 類似度と、BERTScore [15] の 2 種類を試す。コード候補のテキスト変換例を図 2 に示す。

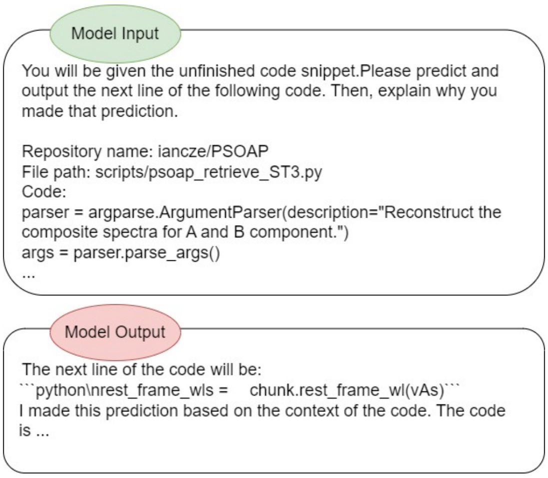


```
You will be given the code snippet. Translate the following code to text
for code search:

Repository name: MontrealCorpusTools/Montreal-Forced-Aligner
File path: tests/test_config.py
Code:
class LdaTrainer(TriphoneTrainer):
    """
    Triphone trainer
...
```

```
The code defines a class `LdaTrainer` that inherits from
`TriphoneTrainer`. It has several parameters  and methods for training
an acoustic model using LDA (Linear Discriminant Analysis) and MLLT
(Maximum Likelihood Linear Transform). The class has ...
...
```

図 2: コード候補のテキスト変換例



```
You will be given the unfinished code snippet.Please predict and
output the next line of the following code. Then, explain why you
made that prediction.

Repository name: iancze/PSOAP
File path: scripts/psoap_retrieve_ST3.py
Code:
parser = argparse.ArgumentParser(description="Reconstruct the
composite spectra for A and B component.")
args = parser.parse_args()
...
```

~~~
The next line of the code will be:
```python\nrest_frame_wls =    chunk.rest_frame_wl(vAs)```
I made this prediction based on the context of the code. The code
is ...
~~~

図 3: 対象コードのテキスト変換例

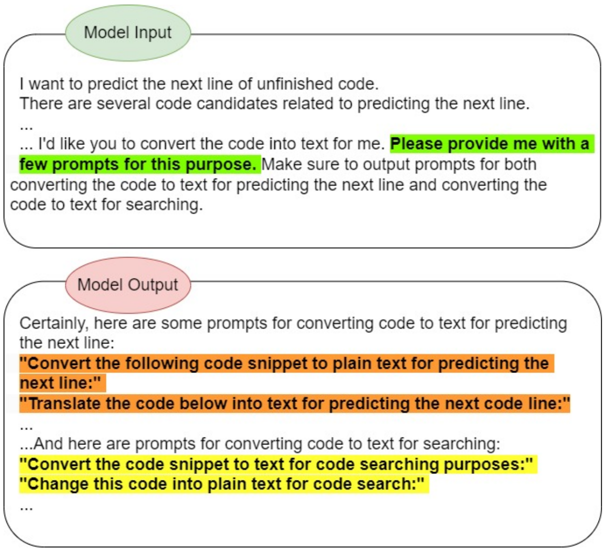
さらに、対象コードは LLM によるテキスト変換に加えて、RepoCoder [4] で提案された手法と提案手法を組み合わせた手法も提案する。RepoCoder では一度 LLM に次の行を予測させ、予測された行を使用して再検索を行う。それに対して、提案手法では次の行の予測に説明を加える方法を提案する。対象コードの予測に加えて説明を出力させる変換の例を図 3 に示す。

## 3.2 プロンプトの設計

プロンプトは LLM の出力に大きな影響を与えることが知られている [16]。そのため、プロンプトを何種類か作成し、検証を行う。本研究では人手で作

---

[2] 埋め込み表現に変換するモデルはモデルサイズが小さいため、LLM ではなく「言語モデル」と呼ぶ。



I want to predict the next line of unfinished code.
There are several code candidates related to predicting the next line.
...
... I'd like you to convert the code into text for me. **Please provide me with a few prompts for this purpose.** Make sure to output prompts for both converting the code to text for predicting the next line and converting the code to text for searching.

Certainly, here are some prompts for converting code to text for predicting the next line:
**"Convert the following code snippet to plain text for predicting the next line:"**
**"Translate the code below into text for predicting the next code line:"**
...
...And here are prompts for converting code to text for searching:
**"Convert the code snippet to text for code searching purposes:"**
**"Change this code into plain text for code search:"**
...

図 4: 自動プロンプトの例

成するプロンプトに加えて、LLM 自身にプロンプトを作成させる自動プロンプトを提案する。自動プロンプトの例を図 4 に示す。図 4 の例は、どのような状況か説明を行い、プロンプトを出力するように指示を行っている。これによりプロンプトの候補が複数個出力される。

# 4 実験

## 4.1 実験設定

**コード候補と対象コードの表現**　コード補完・検索ベンチマークにおけるコード検索タスクにおいて、LLM によるテキスト変換をいくつかのパターンで検証する。コード候補は人間が考えたプロンプトと自動プロンプトで変換したテキストに加えて変換しない元のコードの 3 種類を評価する。対象コードはこの 3 種類に加えて、3.1 節で説明した RepoCoder [4] の手法と提案手法の 5 種類を評価する。実際に使用したプロンプトは付録 A に示す。

**使用したモデル・データセット**　テキスト変換を行う LLM は gpt-3.5-turbo[3] を使用した。その際、一貫性のある生成を行うために温度パラメータを 0 に設定して生成を行った。また、埋め込み表現への変換に用いたモデルは RoBERTa[4], UniXcoder[5], CodeBERT[6], text-embedding-ada-002[7] (以下 ada-002) の 4 種類である。また変換したテキスト間の類似度の計算方法として、埋め込み表現を cosine 類似度で評価する方法と BERTScore [15] を比較した。

評価には RepoBench [3] の検索タスクである repobench-r[8] の Java データセットと Python データセットの両方を用いた。その中で対象コードとコード候補のセットが 8,000 件ある XFF, easy の設定で評価を行った。XFF は、対象コードにおいて予測すべき次の行が、そのコードの中で一番最初に外部のコードを参照する設定である。easy はコード候補が平均で Java は 6.6 ファイル、Python は 6.7 ファイル存在するタスクである。

**評価指標**　コード検索タスクの評価は RepoBench の評価方法に倣い、類似度が最も高いコード候補が正解となる割合 (acc@1) と、類似度が高い 3 つの内に正解が含まれている割合 (acc@3) で評価を行う。

## 4.2 結果

Python の評価結果を表 1 に、Java の評価結果を付録の表 4 に示す。本論文では表 1 について議論する。表 1 は左からそれぞれ類似度の計算方法、類似度計算のための埋め込み表現を作成に使用したモデル、コード候補のプロンプト、対象コードのプロンプトを表す。プロンプトはそれぞれ Human が人手によるプロンプト、Auto が自動プロンプト、Original が元のコード、Pred が次の行の予測、Pred+Explain が次の行の予測と説明を加えたものを表す。

**提案手法がベースラインより高精度**　最も精度が高かったのは acc@1, acc@3 ともに Embedding(埋め込み表現の cosine 類似度) の ada-002、コード候補が Original、対象コードが Pred+Explain の場合である。公開モデルでは BERTScore の UnixCoder、コード候補が Human、対象コードが Pred+Explain の場合である。ベースライン (コード候補が Original、対象コードが Original) と比較すると、精度を大きく上回っており、提案手法の有効性を確認できる。

**Pred+Explain が高精度**　全体的に精度が良いのが、対象コードが Pred+Explain の場合で、他の変換方法に比べてほとんどの場合で精度が高い。特に、次の行の予測をする Pred は RepoCoder [4] で提案された手法であり、説明を加えることで精度が向上していることから、提案手法の有効性を確認できる。

**対象コードが Human, Auto で低精度**　反対に、対象コードが Human, Auto の場合はベースラインを下

---

[3] https://platform.openai.com/docs/models/gpt-3-5
[4] https://huggingface.co/roberta-base
[5] https://huggingface.co/microsoft/unixcoder-base
[6] https://huggingface.co/microsoft/codebert-base
[7] https://platform.openai.com/docs/models/embeddings
[8] https://huggingface.co/datasets/tianyang/repobench-r

表 1: Python データセットの結果

| Retrieval | Model | Candidate | Unfinished Code | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Human | | Auto | | Original | | Pred | | Pred+Explain | |
| | | | acc@1 | acc@3 | acc@1 | acc@3 | acc@1 | acc@3 | acc@1 | acc@3 | acc@1 | acc@3 |
| BERTScore | RoBERTa | Human | 17.06 | 48.77 | 15.92 | 48.95 | 18.29 | 51.09 | 20.85 | 53.64 | 24.27 | 55.39 |
| | CodeBERT | | 16.36 | 48.65 | 15.36 | 48.25 | 18.09 | 50.39 | 19.97 | 52.27 | 21.60 | 53.71 |
| | UniXcoder | | 20.52 | 52.92 | 18.94 | 51.84 | 25.90 | 58.74 | 30.57 | 61.91 | **31.34** | **63.09** |
| | RoBERTa | Auto | 16.55 | 47.75 | 15.69 | 47.72 | 17.44 | 49.44 | 18.09 | 50.45 | 20.49 | 50.99 |
| | CodeBERT | | 16.54 | 47.72 | 15.84 | 46.85 | 17.60 | 49.71 | 17.64 | 50.21 | 18.24 | 49.90 |
| | UniXcoder | | 19.79 | 52.32 | 17.47 | 50.57 | 24.94 | 58.80 | 28.79 | 60.76 | 29.74 | 61.52 |
| | RoBERTa | Original | 16.59 | 47.92 | 15.91 | 47.36 | 17.24 | 47.91 | 17.97 | 49.57 | 19.12 | 50.29 |
| | CodeBERT | | 16.21 | 46.99 | 15.74 | 46.50 | 16.84 | 48.51 | 17.86 | 49.34 | 18.26 | 49.06 |
| | UniXcoder | | 19.86 | 52.34 | 18.02 | 50.81 | 25.00 | 59.06 | 28.79 | 60.72 | 29.37 | 62.12 |
| Embedding | RoBERTa | Human | 16.52 | 48.14 | 16.17 | 48.57 | 16.45 | 47.50 | 16.95 | 47.97 | 19.41 | 50.19 |
| | CodeBERT | | 15.84 | 47.85 | 16.07 | 47.70 | 15.39 | 47.32 | 16.62 | 47.47 | 18.34 | 48.85 |
| | UniXcoder | | 20.29 | 53.64 | 19.00 | 51.75 | 25.06 | 58.90 | 29.79 | 61.32 | 30.65 | 62.46 |
| | ada-002 | | 19.40 | 52.81 | 17.81 | 51.35 | 28.40 | 61.92 | 33.36 | 64.80 | 33.71 | 65.56 |
| | RoBERTa | Auto | 16.35 | 47.51 | 15.72 | 47.14 | 15.85 | 47.74 | 16.55 | 47.77 | 17.37 | 47.71 |
| | CodeBERT | | 16.31 | 47.71 | 15.54 | 47.97 | 15.86 | 46.76 | 16.04 | 47.15 | 16.51 | 46.75 |
| | UniXcoder | | 19.70 | 52.24 | 17.16 | 50.15 | 24.16 | 58.12 | 27.82 | 60.61 | 29.40 | 61.35 |
| | ada-002 | | 19.14 | 52.42 | 18.32 | 50.07 | 29.06 | 62.49 | 34.55 | 65.38 | 34.00 | 65.58 |
| | RoBERTa | Original | 16.86 | 48.11 | 16.34 | 47.64 | 15.97 | 47.05 | 16.75 | 48.42 | 17.07 | 48.65 |
| | CodeBERT | | 16.15 | 46.91 | 16.04 | 46.85 | 15.77 | 46.26 | 15.95 | 47.37 | 16.00 | 47.12 |
| | UniXcoder | | 19.85 | 51.74 | 17.66 | 49.96 | 24.15 | 58.85 | 27.66 | 60.00 | 28.10 | 59.95 |
| | ada-002 | | 19.64 | 52.56 | 17.62 | 50.49 | 27.95 | 63.31 | 33.66 | 65.76 | **34.82** | **66.61** |

回り低い精度となった。これはコードをテキストに変換する際に、コード全体に関する変換を行ったため、次の行に関する情報が減少したためと考えられる。対象コードが Original の時は RepoBench [3] に従い、最後から 3 行をモデルに入力したため、次の行に関する情報が多く残ったと考えられる。

**人手プロンプトによるコード候補の変換で高精度** コード候補のテキスト変換に着目すると、精度の全体的な傾向としては Original と Auto がおおよそ同じで、Human の精度が高い結果となった。これはテキスト変換による精度向上の有効性は確認できたが、自動プロンプトによる精度の向上は確認できなかったことを表している。プロンプトの設計は、本実験条件では人手で行った方が精度が高く、プロンプトの自動化は今後の課題である。

**UniXcoder と ada-002 が高精度** モデルごとに精度を評価すると、UniXcoder と ada-002 では全体的に精度が高かったが、RoBERTa や CodeBERT ではいずれのプロンプトでも精度が低かった。CodeBERT と UniXcoder は RepoBench [3] で示されていた傾向と似ており、それぞれ CodeBERT が低精度、UniXcoder が高精度となった。RoBERTa はコードで事前学習されていないが、テキスト変換によりコードがテキストに変換されるので精度が高くなると予測したが、低い結果となった。ada-002 は CodeSearchNet [17] データセットでも精度が高く9)、repobench-r でも高い精度が得られた。

**Embedding より BERTScore が高精度** 類似度の計算方法で比較を行うと、同じモデル同士では Embedding よりも BERTScore [15] が高精度である。BERTScore [15] は Embedding と比べて、類似度を計算する際にトークンごとに類似度を取得するため、Embedding よりも情報量が多く、高精度になったと考えられる。しかし、最も精度が高かったのは acc@1, acc@3 ともに Embedding の ada-002、コード候補が Original、対象コードが Pred+Explain の場合である。ada-002 のような非公開のモデルでは BERTScore [15] は適用できないことに留意が必要である。

## 5 おわりに

本研究ではコード補完タスクにおけるコード検索で、コードをテキストに変換して類似度を取得する方法を提案した。また、プロンプトを LLM に作成させる自動プロンプトの提案も行った。テキスト変換については精度の向上が確認できたが、自動プロンプトに関しては精度の向上を確認できなかった。

今後はコード補完タスク以外のデバッグなどのタスクや、リポジトリレベルのタスク以外にもテキスト変換を応用して適用したいと考えている。本研究がリポジトリレベルでのコード生成タスクの発展に寄与することを願っている。

---

9) https://openai.com/blog/new-and-improved-embedding-model

## 謝辞

本研究は日本電信電話株式会社と早稲田大学の共同研究により実施した。

## 参考文献

[1] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? **arXiv preprint arXiv:2310.06770**, 2023.

[2] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. **Advances in neural information processing systems**, Vol. 30, , 2017.

[3] Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. **arXiv preprint arXiv:2306.03091**, 2023.

[4] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. RepoCoder: Repository-level code completion through iterative retrieval and generation. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, **Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing**, pp. 2471–2484, Singapore, December 2023. Association for Computational Linguistics.

[5] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In Trevor Cohn, Yulan He, and Yang Liu, editors, **Findings of the Association for Computational Linguistics: EMNLP 2020**, pp. 1536–1547, Online, November 2020. Association for Computational Linguistics.

[6] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. UniXcoder: Unified cross-modal pre-training for code representation. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio, editors, **Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)**, pp. 7212–7225, Dublin, Ireland, May 2022. Association for Computational Linguistics.

[7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

[8] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! **arXiv preprint arXiv:2305.06161**, 2023.

[9] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. **arXiv preprint arXiv:2308.12950**, 2023.

[10] Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. CodeT5+: Open code large language models for code understanding and generation. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, **Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing**, pp. 1069–1088, Singapore, December 2023. Association for Computational Linguistics.

[11] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021.

[12] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In **Proceedings of the 2014 international symposium on software testing and analysis**, pp. 437–440, 2014.

[13] Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. Cocomic: Code completion by jointly modeling in-file and cross-file context. **arXiv preprint arXiv:2212.10007**, 2022.

[14] Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. Repository-level prompt generation for large language models of code. In **International Conference on Machine Learning**, pp. 31693–31715. PMLR, 2023.

[15] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. Bertscore: Evaluating text generation with BERT. In **8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020**. OpenReview.net, 2020.

[16] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. **Advances in Neural Information Processing Systems**, Vol. 35, pp. 24824–24837, 2022.

[17] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. **arXiv preprint arXiv:1909.09436**, 2019.

# A　プロンプト

実際に用いたプロンプトを表 2, 3 に示す。表の種類の共通はすべてのプロンプトの最初に入力するプロンプト文を表し、Human, Auto, Pred, Pred+Explain はそれぞれ本研究で用いたプロンプト文で、メタ情報はすべてのプロンプトの後に入力される情報である。表のプロンプトはそれぞれの種類で、使用したプロンプト文を表す。プロンプトの構成として共通の文が最初で、その次に Human, Auto, Pred, Pred+Explain のいずれかの文、最後にメタ情報となる。これらのプロンプトの後にコードが入力される。

表 2: コード候補の変換に用いたプロンプト

| 種類 | プロンプト |
|---|---|
| 共通 | You will be given the code snippet. |
| Human | Your task is to summarize the code into text for code retrieval. The length should be around 500 characters. |
| Auto | Translate the following code to text for code search: |
| メタ情報 | Repository name: リポジトリ名<br>File path: ファイルパス |

表 3: 対象コードの変換に用いたプロンプト

| 種類 | プロンプト |
|---|---|
| 共通 | You will be given the unfinished code snippet. |
| Human | Your task is to summarize the code into text for predicting the next line of the code. The length should be around 500 characters. |
| Auto | Convert the given incomplete code snippet into natural language text: |
| Pred | Predict the next line of the following code and output it. Make sure to only output the prediction. |
| Pred+Explain | Please predict and output the next line of the following code. Then, explain why you made that prediction. |
| メタ情報 | Repository name: リポジトリ名<br>File path: ファイルパス |

# B　Java データセットの結果

表 4 に Java の結果を示す。おおむね Python と同じ傾向だが、Pred+Explain よりも Pred のほうが精度が高い場合が多い。コード候補は変換したほうが良い結果を得られているため本手法は有効であるといえる。

表 4: Java データセットの結果

| Retrieval | Model | Candidate | Unfinished Code | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Human | | Auto | | Original | | Pred | | Pred+Explain | |
| | | | acc@1 | acc@3 | acc@1 | acc@3 | acc@1 | acc@3 | acc@1 | acc@3 | acc@1 | acc@3 |
| BERTScore | RoBERTa | Human | 13.90 | 45.26 | 12.96 | 43.82 | 16.86 | 49.22 | 20.21 | 52.31 | 20.75 | 52.97 |
| | CodeBERT | | 14.55 | 46.27 | 13.32 | 45.10 | 17.11 | 48.99 | 18.84 | 51.22 | 18.61 | 50.70 |
| | UniXcoder | | 13.87 | 45.79 | 13.19 | 43.89 | 20.70 | 53.96 | 24.95 | 56.76 | 23.55 | 56.89 |
| | RoBERTa | Auto | 15.11 | 46.64 | 14.19 | 45.47 | 16.75 | 48.61 | 19.59 | 51.10 | 19.76 | 52.22 |
| | CodeBERT | | 15.42 | 47.95 | 14.34 | 45.75 | 16.64 | 48.46 | 18.16 | 49.87 | 18.25 | 49.76 |
| | UniXcoder | | 14.12 | 45.47 | 12.97 | 44.09 | 19.80 | 53.69 | 23.40 | 55.32 | 22.91 | 55.72 |
| | RoBERTa | Original | 14.75 | 46.65 | 13.99 | 45.74 | 16.46 | 49.45 | 17.31 | 50.85 | 16.96 | 50.30 |
| | CodeBERT | | 14.92 | 47.29 | 14.25 | 46.85 | 16.61 | 49.72 | 17.20 | 50.45 | 17.22 | 49.34 |
| | UniXcoder | | 14.85 | 46.27 | 13.55 | 45.14 | 20.24 | 54.04 | 24.06 | 57.62 | 22.75 | 56.30 |
| Embedding | RoBERTa | Human | 14.80 | 46.30 | 14.57 | 45.54 | 16.39 | 49.60 | 16.96 | 49.82 | 18.27 | 50.26 |
| | CodeBERT | | 14.95 | 47.46 | 14.89 | 46.19 | 15.85 | 48.29 | 16.67 | 49.21 | 17.04 | 49.14 |
| | UniXcoder | | 14.00 | 46.00 | 13.10 | 44.11 | 20.20 | 53.91 | 24.21 | 56.62 | 23.46 | 56.69 |
| | ada-002 | | 12.90 | 44.74 | 12.34 | 43.15 | 21.61 | 55.97 | 25.81 | 58.71 | 24.70 | 58.86 |
| | RoBERTa | Auto | 14.99 | 46.56 | 15.17 | 45.92 | 16.45 | 48.86 | 17.19 | 48.99 | 17.37 | 49.66 |
| | CodeBERT | | 15.62 | 47.46 | 14.81 | 46.29 | 15.87 | 48.60 | 16.72 | 48.35 | 16.70 | 48.57 |
| | UniXcoder | | 14.14 | 45.95 | 13.15 | 44.16 | 20.32 | 54.30 | 23.04 | 55.67 | 22.96 | 56.21 |
| | ada-002 | | 13.39 | 44.99 | 12.46 | 43.71 | 22.25 | 56.95 | 26.44 | 59.21 | 25.90 | 59.16 |
| | RoBERTa | Original | 14.77 | 46.34 | 14.21 | 46.26 | 16.21 | 47.79 | 15.96 | 47.80 | 15.30 | 47.06 |
| | CodeBERT | | 15.42 | 47.16 | 14.99 | 47.10 | 15.47 | 47.86 | 16.07 | 48.14 | 15.54 | 47.62 |
| | UniXcoder | | 14.89 | 47.16 | 13.84 | 45.94 | 20.05 | 52.97 | 22.66 | 55.92 | 21.00 | 54.69 |
| | ada-002 | | 13.12 | 45.12 | 12.16 | 43.56 | 22.10 | 57.20 | 27.91 | 61.24 | 26.41 | 60.52 |