Nearest Neighbor Code Generation and Summarization

Alex Xie[1], **Vincent Hellendoorn**[2]
School of Computer Science[1], Institute of Software Research[2]
Carnegie Mellon University
Pittsburgh, PA 15213
alexx@cs.cmu.edu, vhellendoorn@cmu.edu

Abstract

Retrieval-based approaches have recently been shown to be effective for many tasks within NLP. In this work, we investigate the utility of neural probabilistic retrieval (kNN-LM) for the tasks of code generation and code summarization; we additionally propose an adversarial training objective to improve representations for nearest neighbor search. We find that while kNN retrieval yields marginal improvements for code generation, it provides a +0.44 average BLEU increase for code summarization. We further find that low-resource languages benefit from retrieving from high-resource languages; on Ruby, retrieving from a datastore of Java and Python examples outperforms a Ruby-only datastore by +0.3 BLEU. Finally, we find that retrieval aids in zero-shot language model adaptation across programming languages. Code is available at https://github.com/axie66/codet5-knn.

1 Introduction

There has been a great deal of interest in the tasks of automated code generation, completion, and summarization—especially, in recent years, through neural methods—with the aims of streamlining the workflow and increasing the productivity of software developers and programmers (Xu et al., 2021; Lu et al., 2021). In particular, large transformer language models of code have very recently shown great promise on these tasks, yielding state of the art results that approach human performance (Chen et al., 2021; Wang et al., 2021; Nijkamp et al., 2022). However, like natural language, source code contains many long-tailed patterns that even large parametric models often struggle with. Moreover, as the size of state-of-the-art models grows, so too does the cost of training them, making it potentially infeasible to adapt a model trained on a certain domain or language to a new one.

Inspired by the NLP literature, we investigate the use of non-parametric neural probabilistic retrieval for code summarization, generation and language modeling. Such retrieval mechanisms are appealing as they empower a base parametric model with the ability to leverage a large knowledge base of cached examples at inference time, yet add no parameters and thus require no additional training. However, retrieval-based methods also come with significant drawbacks, including a sizeable memory footprint due to the large retrieval datastore as well as slower inference as a result of potentially expensive nearest neighbor search operations. Hence, it is important to determine under what circumstances the benefits of retrieval outweigh the negatives. In this work, we explore three respects in which non-parametric retrieval may offer advantages over parametric approaches for code-related sequence generation tasks:

1. We investigate the extent to which memorization of rare patterns via retrieval aids in code generation and summarization tasks.
2. Through experiments on code language modeling, we study the utility of retrieval for efficient adaptation across PLs.
3. Through code summarization experiments, we study the degree to which cross-lingual datastore retrieval enables high-resource to low-resource PL transfer.

2 Related Work

While earlier neural approaches for code generation and summarization often incorporated explicit structure as inductive biases (Rabinovich et al., 2017; Yin and Neubig, 2018), recent work on the task, as in NLP at large, has moved toward large pretrained transformer models. Through large-scale unsupervised pretraining on code and text,

these models (Codex, PL-BART, CodeT5, *inter alia*) yield state-of-the-art performance on a variety of code benchmarks despite using only linear, unstructured representations of code (Chen et al., 2021; Ahmad et al., 2021; Wang et al., 2021; Xu et al., 2022; Nijkamp et al., 2022; Fried et al., 2022).

Simultaneously, retrieval-based approaches have received significant attention within NLP, with applications ranging from question answering to language modeling (Lewis et al., 2020; Borgeaud et al., 2021). An approach of particular relevance to this work is kNN-LM, which augments a non-parametric token-level neural retrieval mechanism (Khandelwal et al., 2020, 2021). For code generation and summarization in particular, Parvez et al. propose REDCODER, a retrieval-augmented model which generates code and summaries conditioned on examples retrieved based on the input sequence (Parvez et al., 2021).

3 Background: kNN-LM

In this work, we consider a specific instantiation of non-parametric neural retrieval, kNN-LM, which augments a language model at inference time with a token-level retrieval mechanism over a datastore of cached examples (Khandelwal et al., 2020). These examples are pairs (c, w) where c is a context (i.e. the first i tokens in a sentence) and w is the target next token (i.e. the ith token in the sentence). Concretely, given a context encoder f that maps contexts to fixed-length vectors and a set of context-token pairs $\mathcal{D} = \{(c^{(i)}, w^{(i)}\}_{i=1}^{N}$, the datastore $(\mathcal{K}, \mathcal{V})$ is constructed as follows:

$$(\mathcal{K}, \mathcal{V}) = \{(f(c^{(i)}), w^{(i)}) \,|\, (c^{(i)}, w^{(i)}) \in \mathcal{D}\}$$

The context representation $f(c^{(i)})$ is generally taken to be some intermediate representation produced by the model, for example the self-attention output of the final transformer decoder block.

During inference, for each token generated, kNN-LM retrieves from the datastore the k nearest neighbors $\mathcal{N} = \{(k_i, v_i)\}_{i=1}^{k}$ of the current context (i.e. the previously generated tokens), represented using the same encoder f that was used to create the datastore. Through a softmax over distances, kNN-LM converts the neighbors to a distribution over the vocabulary:

$$p_{k\text{NN}}(w_t|w_{<t}) \propto \sum_{(k,v)\in\mathcal{N}} \mathbb{1}[w_t = v] \exp\left(\frac{-d(f(w_{<t}), k)}{T}\right)$$

Note that the distance metric d denotes squared L2 distance and the temperature T is a hyperparameter which influences the sharpness of the distribution.

The kNN distribution is then linearly interpolated according to a tuned hyperparameter λ with the distribution of the parametric model θ:

$$p(w_t|w_{<t}) = \lambda\, p_{k\text{NN}}(w_t|w_{<t}) + (1-\lambda) p_\theta(w_t|w_{<t})$$

For sequence-to-sequence tasks, we use a slight generalization of this mechanism, kNN-MT, which considers both the full input sequence and the previous tokens of the output sequence as context (Khandelwal et al., 2021). For an encoder-decoder transformer, this can be trivially done by taking an arbitrary decoder hidden state as the context representation $f(c)$.

4 Task Description

In this section, we formally present the code-related sequence generation tasks studied in this work.

4.1 Code Generation

Code generation is the task of generating a piece of source code given a natural language specification or intent. This can be done at various levels of granularity; in this work, we study generation at the line and function levels. Training is performed by minimizing the (conditional) negative log likelihood of the code sequence given the specification:

$$\mathcal{L}_{\text{NLL}}(\theta) = \mathbb{E}_{(s,c)\sim D}\left[-\sum_{t=1}^{|c|} \log p_\theta(c_t|c_{<t}, s)\right]$$

where D is a dataset consisting of parallel specification and code pairs (s, c).

4.2 Code Summarization

Code summarization refers to the task of generating a natural language summary (i.e. a function docstring) given a piece of source code. We again train by maximizing conditional log likelihood, this time of the summary given the code:

$$\mathcal{L}_{\text{NLL}}(\theta) = \mathbb{E}_{(c,s)\sim D}\left[-\sum_{t=1}^{|s|} \log p_\theta(s_t|s_{<t}, c)\right]$$

While summarization appears to be the inverse task of code generation, summaries are generally higher-level and more ambiguous than natural language specifications for code generation; as such,

datasets for code summarization may not be appropriate for code generation and vice versa (see Appendix A.1.3 for an example of this).

Further, as we have access to a summarization dataset across multiple programming languages (see Section 5.1), we train a single multilingual summarization model rather than a different model for each language. Note that "multilingual" refers to the source/PL side and not to the output/NL side, which is always in English. To reduce dataset imbalance across languages, we follow (Lample and Conneau, 2019) and sample training examples from the N languages according to the following multinomial distribution $\{q_i\}_{i=1}^{N}$:

$$q_i = \frac{p_i^\alpha}{\sum_{j=1}^{N} p_j^\alpha} \text{ where } p_i = \frac{|D_i|}{\sum_{j=1}^{N} |D_j|}$$

Note that we use $\alpha = 0.3$ in this work.

4.2.1 Adversarial Training for Language Invariant Representations

As we train on multilingual data, we hypothesize that it may be beneficial to normalize kNN datastore representations across languages, both to prevent representations from being biased toward higher-resource languages and to facilitate retrieval across languages. Hence, we propose to add an adversarial objective during fine-tuning, jointly training a domain discriminator along with the base model (Ganin et al., 2016; Lample et al., 2018). Specifically, the discriminator is trained to discern the identity of the source PL given the kNN representation. As a result, model representations should be homogenized across languages, while still carrying information about underlying code semantics.

Formally: given a discriminator θ_{Adv}, the kNN context encoder f, and a summarization dataset D in which each example (c, s, ℓ) is (trivially) annotated with the source language ℓ, the adversarial loss is as follows:

$$\mathcal{L}_{\text{Adv}}(\theta) = \mathbb{E}_{(c,s,\ell)\sim D}\left[-\sum_{t=1}^{|s|} \log p_{\theta_{\text{Adv}}}(\ell | f(c, s_{<t}))\right]$$

The discriminator is trained to minimize the above value, whereas the model is trained to maximize it. The overall model loss is then

$$\mathcal{L}(\theta) = \lambda_{\text{Adv}}\mathcal{L}_{\text{Adv}}(\theta) + \lambda_{\text{NLL}}\mathcal{L}_{\text{NLL}}(\theta)$$

We use $\lambda_{\text{Adv}} = 0.33$ and $\lambda_{\text{NLL}} = 0.67$.

4.3 Code Language Modeling

Language modeling is the task of assigning probabilities to sequences of tokens—in this case, function bodies. Training is performed by minimizing the negative log likelihood:

$$\mathcal{L}_{\text{NLL}}(\theta) = \mathbb{E}_{c\sim D}\left[-\sum_{t=1}^{|c|} \log p_\theta(c_t | c_{<t})\right]$$

In this work, we specifically study code language modeling with respect to domain adaptation: given a trained language model, we wish to adapt it to a new language while keeping performance loss on the original language(s) (i.e. catastrophic forgetting) to a minimum.

5 Experimental Setup

5.1 Datasets

Code Generation: We use two datasets at different levels of granularity: CoNaLa, a Python dataset consisting of natural language intents paired with one-liners (Yin et al., 2018), and Concode, a Java dataset containing intents and class contexts (i.e. attribute names and types) paired with full method bodies (Iyer et al., 2018). For CoNaLa, we augment the small human-annotated training set of 2k examples with 12k examples extracted from Python documentation and 100k examples automatically mined from StackOverflow (Xu et al., 2020).

Code Summarization: We use a subset of the CodeSearchNet corpus provided by the CodeXGLUE benchmark (Lu et al., 2021). The CodeSearchNet data contains over 900k paired function-docstring examples mined from Github across six programming languages. As Ruby is the lowest resource language within the dataset, we use it as the target language for datastore transfer as well as for unsupervised summarization experiments (see Appendix A.2).

Code Language Modeling: We reuse the CodeSearchNet data, taking only the functions and discarding the summaries. As with summarization, we specifically study domain adaptation to Ruby.

5.2 Evaluation Metrics

Following previous work, we evaluate code generation using BLEU-4 and code summarization using smoothed BLEU-4. For language modeling, we evaluate using perplexity (exponentiation of the

	Dataset	Train	Val	Test
	CoNaLa	2,179	200	500
	+ Doc	12,574	-	-
	+ Mined	106,060	-	-
	Concode	100,000	2,000	2,000
CodeSearchNet	Ruby	24,927	1,400	1,261
	Javascript	58,025	3,885	3,291
	Go	167,288	7,325	8,122
	Python	251,820	13,914	14,918
	Java	164,923	5,183	10,955
	PHP	241,241	12,982	14,014

Table 1: Dataset statistics.

negative log likelihood), a standard metric for language models that has repeatedly been found to correlate strongly with downstream performance.

5.3 Base Models

For code generation and code summarization experiments, we fine-tune CodeT5, a state-of-the-art pretrained code language model (Wang et al., 2021) consisting of 12 encoder and 12 decoder layers. We finetune for 5 epochs on the relatively small code generation datasets and roughly 80k steps on the larger code summarization data.

For code language modeling, an unconditional generation task, we use a 12 layer decoder-only transformer, initialized from a GPT-2 pretrained checkpoint (Radford et al., 2019). In order to evaluate language adaptation to Ruby, we initially train the model for 100k steps on the other five PLs in CodeSearchNet. Afterward, we retrain the model for a maximum of 1k steps on Ruby, saving checkpoints every 100 steps to evaluate performance on both Ruby and the original PLs. Further, we outfit certain model checkpoints with kNN-LM to measure the effects of retrieval on domain adaption.

We use the Pytorch model and tokenizer implementations and weights provided by Huggingface Transformers[1].

5.4 Datastore Construction and Retrieval

For each task, we construct our datastores from the training data. For code summarization in particular, we experiment with both retrieving within the test PL only and retrieving across PLs. Due to computational/storage constraints, for tasks with particularly large training data, namely summarization and language modeling, we randomly sample a subset of the data (between 10% and 50%) for the datastore. While we anticipate that this may hurt kNN performance, previous work has shown that this decrease is generally relatively minor (Khandelwal et al., 2020). We use the `faiss` library for approximate nearest neighbor search (Johnson et al., 2017).

6 Results and Discussion

6.1 Code Generation

	CoNaLa	Concode
CodeT5 base	38.56	39.6
+ kNN	38.60	39.6

Table 2: BLEU results for code generation experiments

We find that the kNN mechanism underperforms for code generation, yielding only marginal improvements over the base model on both the CoNaLa and Concode datasets. We suspect that this may be due to a combination of insufficient relevant neighbors within the datastore, noise among the automatically mined examples and inflexibility in the retrieval mechanism. We report qualitative results as well as additional code generation experiments in Appendix A.1.

6.2 Code Summarization

As shown in the top two rows of Table 3, for five of the six languages, retrieval (of in-language examples) yields improvements over the base model; gains are surprisingly large on Javascript in particular. Also contrary to expectations, on two of the highest resource languages, Go and Python, kNN provides little to no benefit. The causes of these positive and negative results are not immediately apparent; it may be that the neural kNN mechanism is simply discovering "shortcuts" that alternately help or hurt performance.

Meanwhile, the relatively poor kNN performance for Ruby is somewhat expected given the relatively small size of the datastore. However, exploiting the multilingual nature of the model and the fact that the output language is invariably English, we experiment with retrieving examples across PLs, allowing the model to draw from a much larger knowledge base. The results, shown in Table 4, demonstrate that cross-lingual

[1] https://huggingface.co/docs/transformers

	Ruby	Javascript	Go	Python	Java	PHP
Base	16.01	16.24	19.74	19.43	20.29	26.23
Base + kNN	16.04	**17.92**	19.46	19.46	**20.73**	**26.59**
Base + Adv	15.54	16.16	19.41	19.21	19.98	25.89
Base + Adv + kNN	**16.22**	16.42	**19.77**	**19.52**	20.42	26.50

Table 3: BLEU results for summarization on CodeSearchNet, with and without adversarial training

	BLEU	Datastore tokens
Base Model + Ruby	16.04	391,270
+ Python datastore	16.24	2,199,034
+ Java datastore	16.31	3,548,944
+ all datastores	16.30	7,474,138

Table 4: Effects of expanding the Ruby kNN datastore with other languages

retrieval helps, but only up to a certain point. Interestingly, combining all datastores slightly hurts performance, perhaps because of excessive noise in the combined datastore or because certain representations fail to generalize across languages.

Additionally, we observe that while adversarial training generally hurts the base model, the corresponding increases from kNN retrieval are more uniform across languages and particularly pronounced for Ruby. Surprisingly, however, upon repeating the Ruby datastore expansion experiments, we find that adversarial training offers little to no benefit for cross-lingual retrieval.

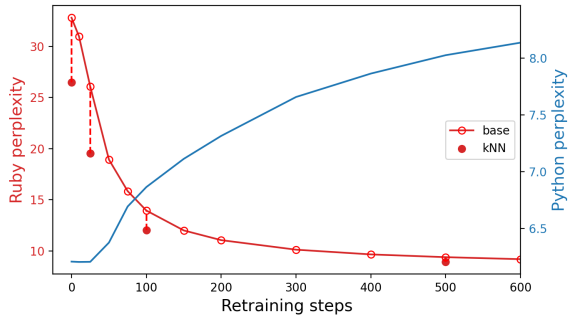

Figure 1: Ruby LM adaptation results. Checkpoints after 0, 50, 100, and 500 steps are outfitted with kNN-LM (solid red dots).

6.3 Code Language Modeling

We find that adding kNN-LM improves adaptation to Ruby at all points in retraining, though it helps most while the model is still severely underfitting the new data. Notably, with no training whatsoever, kNN immediately yields a 20% decrease in perplexity from 33 to 26.5. However, after a few hundred steps, the improvements appear fairly marginal.

Moreover, the gains from kNN seem equivalent to those of simply training an additional 50-100 steps. While this continued training does have the effect of reducing performance on the original Python data, the overall perplexity increase (6.5 to 8) does not appear intolerably large; in cases where preserving performance on the original data is essential, it may be preferable to train a copy of the model on the new data. Hence, the current results, while promising for zero-shot scenarios, may not be of practical use in the vast majority of settings.

7 Conclusion and Future Work

We find that while non-parametric retrieval is largely ineffective for code generation, it yields promising results for code summarization and language modeling, particularly in terms of cross-lingual transfer and adaptation. Most notably, we find that (1) for tasks with a common output distribution such as summarization, low-resource PLs benefit from retrieving examples belonging to high-resource PLs and (2) retrieval can facilitate zero-shot adaptation to previously unseen PLs. Additionally, we introduce an adversarial objective which may inform future approaches to kNN-LM representation learning or language model pretraining.

An obvious and likely fruitful avenue for further work is scaling up our approach to larger models and datastores. For instance, the effects of kNN on domain adaptation may be more pronounced on a high capacity, massively pretrained model such as Polycoder (Xu et al., 2022) than the relatively small transformer model used in this work. In a similar vein, expanding to billion-scale datastores commonplace in the literature (Khandelwal et al., 2020, 2021) (several orders of magnitude larger than those used in this work) would almost certainly yield significantly stronger results as well.

8 Reflection

I have found this semester-long research project to be an immensely rewarding and edifying experience. Perhaps the most valuable lesson I have learned is the importance of flexiblity in research. At the beginning of the project, I was narrowly focused on code generation, but I soon hit a brick wall when my initial results proved less than promising. Fortunately, guided by Prof. Hellendoorn, I reexamined the underlying goals and questions of my work and altered the project direction accordingly; this has since opened up a variety of new and exciting research directions, only a fraction of which I have had the opportunity to explore so far.

9 Acknowledgements

I would like to express my gratitude to my mentor, Professor Vincent Hellendoorn, for his invaluable insight, encouragement, and generosity throughout this (thoroughly enjoyable!) project, and to Professors Leila Wehbe and Bogdan Vasilescu for their guidance over the course of the semester.

References

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, Online. Association for Computational Linguistics.

Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George van den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, Diego de Las Casas, Aurelia Guy, Jacob Menick, Roman Ring, Tom Hennigan, Saffron Huang, Loren Maggiore, Chris Jones, Albin Cassirer, Andy Brock, Michela Paganini, Geoffrey Irving, Oriol Vinyals, Simon Osindero, Karen Simonyan, Jack W. Rae, Erich Elsen, and Laurent Sifre. 2021. Improving language models by retrieving from trillions of tokens.

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. 2021. Evaluating large language models trained on code.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis.

Yaroslav Ganin, Evgeniya Ustinova, Hana Ajakan, Pascal Germain, Hugo Larochelle, François Laviolette, Mario March, and Victor Lempitsky. 2016. Domain-adversarial training of neural networks. *Journal of Machine Learning Research*, 17(59):1–35.

Junxian He, Graham Neubig, and Taylor Berg-Kirkpatrick. 2021. Efficient nearest neighbor language models. In *Proceedings of EMNLP*.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1643–1652, Brussels, Belgium. Association for Computational Linguistics.

Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2017. Billion-scale similarity search with gpus. *arXiv preprint arXiv:1702.08734*.

Urvashi Khandelwal, Angela Fan, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. 2021. Nearest neighbor machine translation. In *International Conference on Learning Representations (ICLR)*.

Urvashi Khandelwal, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. 2020. Generalization through Memorization: Nearest Neighbor Language Models. In *International Conference on Learning Representations (ICLR)*.

Guillaume Lample and Alexis Conneau. 2019. Cross-lingual language model pretraining. *Advances in Neural Information Processing Systems (NeurIPS)*.

Guillaume Lample, Alexis Conneau, Ludovic Denoyer, and Marc'Aurelio Ranzato. 2018. Unsupervised machine translation using monolingual corpora only. In *International Conference on Learning Representations (ICLR)*.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474. Curran Associates, Inc.

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. *CoRR*, abs/2102.04664.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. A conversational paradigm for program synthesis. *arXiv preprint*.

Md Rizwan Parvez, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval augmented code generation and summarization. In *EMNLP-Findings*.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1139–1149, Vancouver, Canada. Association for Computational Linguistics.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021*.

Frank F Xu, Uri Alon, Graham Neubig, and Vincent J Hellendoorn. 2022. A systematic evaluation of large language models of code. *arXiv preprint arXiv:2202.13169*.

Frank F. Xu, Zhengbao Jiang, Pengcheng Yin, and Graham Neubig. 2020. Incorporating external knowledge through pre-training for natural language to code generation. In *Annual Conference of the Association for Computational Linguistics*.

Frank F. Xu, Bogdan Vasilescu, and Graham Neubig. 2021. In-ide code generation from natural language: Promise and challenges. *ACM Transactions on Software Engineering and Methodology*, 30.

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.

Pengcheng Yin and Graham Neubig. 2018. TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation. In *Conference on Empirical Methods in Natural Language Processing (EMNLP) Demo Track*.

A Appendix

A.1 Additional Code Generation Results

A.1.1 Qualitative Results

We report selected qualitative results on CoNaLa below, comparing the outputs of the base model with those of the base model augmented with kNN retrieval. As the tuned value of λ for code generation is relatively small at just 0.05, the kNN mechanism has a relatively small impact on model predictions, meaning the majority of model predictions (both correct and incorrect) are unchanged, as in the third example below.

Intent	get the integer location of a key var0 in a pandas data frame
GT	`df.index.get_loc('var0')`
Base	`df['var0'].apply(` `  lambda x:  x.index.get_loc(var0))`
kNN	`df.index.get_loc(var0)`
Intent	convert a raw string var0 into a normal string
GT	`var0.decode('string_escape')`
Base	`var0.decode('string_escape')`
kNN	`unicodedata.normalize('NFKD',` `  var0).encode('ascii', 'ignore')`
Intent	add a header to a csv file
GT	`writer.writeheader()`
Base	`writer.writerow(row)`
kNN	`writer.writerow(row)`

Table 5: Qualitative results on CoNaLa

A.1.2 Adaptive Retrieval

In our analysis of CoNaLa performance, we found that roughly 50% of the time, the kNN mechanism was increasing the probability of the correct token; in theory, if we could accurately identify these cases, we should be able to make much better use of retrieval. To this end, we experiment with an adaptive retrieval mechanism (He et al., 2021). Specifically, we add a lightweight MLP on top of the model to estimate the interpolation coefficient λ given the kNN query vector (unlike He et al., we do not use additional text features). However, we find that adaptive retrieval significantly hurts generalization performance, dragging dev set performance down from nearly 40 to 22 BLEU.

A.1.3 CodeSearchNet Experiments

To investigate the feasibility of code generation on a code summarization dataset, we run experiments on the Python subset of the CodeSearchNet dataset. As shown in Table 6, the results both with and without kNN are extremely poor and provide little

insight into the model itself; however, these results are not particularly out of place compared to other state-of-the-art models. As discussed in Section 4.2, this can be largely attributed to the assymmetry between the one-line summaries and the far longer and more complex function bodies.

	BLEU	CodeBLEU
CodeT5	2.41	13.14
+ kNN	2.51	13.40
CodeGPT	3.11	11.31
PL-BART	4.89	12.01

Table 6: Results on CodeSearchNet Python. The bottom two rows are included to provide context regarding the results and are reproduced from (Parvez et al., 2021).

A.2 Unsupervised Code Summarization

	Base	Ruby MLM
Ruby	11.77	12.57
JS	12.22	12.62
Go	14.49	15.28
Python	14.80	15.05
Java	14.50	14.78
PHP	19.07	19.46

Table 7: Unsupervised summarization experiments on Ruby.

In addition to code summarization, we also explore the task of *unsupervised* code summarization. In this setting, we are provided paired code-summary data in $N-1$ languages, along with source code-only data in an Nth "target" language (in our experiments, Ruby); we are then tasked with learning to summarize this target language.

In order to instill knowledge of the code-only language into the model, we introduce a masked-language modeling objective for the encoder:

$$\mathcal{L}_{\text{MLM}}(\theta) = \mathbb{E}_{x\sim D}\Bigg[\sum_{t=1}^{|x|} \mathbb{1}[x_t = \langle \texttt{mask} \rangle] \cdot p_{\theta_{\text{Enc}}}(x_t | x_{\neq t})\Bigg] \tag{1}$$

In addition, to prevent a mismatch between representations of existing languages and the code-only language, we again introduce an adversarial loss, this time on the encoder hidden states.

$$\mathcal{L}_{\text{Adv}}(\theta) = \mathbb{E}_{(c,s,\ell)\sim D}\Bigg[-\sum_{i=1}^{|c|} p_{\theta_{\text{Adv}}}(\ell | f_{\text{Enc}}(c)_i)\Bigg] \tag{2}$$

Hence, the overall losses for code-only and paired examples, respectively, are as follows:

$$\begin{aligned}\mathcal{L}_{\text{Code-Only}}(\theta) &= \lambda_{\text{MLM}}\mathcal{L}_{\text{MLM}}(\theta) + \lambda_{\text{Adv}}\mathcal{L}_{\text{Adv}}(\theta) \\ \mathcal{L}_{\text{Code-Text}}(\theta) &= \lambda_{\text{NLL}}\mathcal{L}_{\text{NLL}}(\theta) + \lambda_{\text{Adv}}\mathcal{L}_{\text{Adv}}(\theta)\end{aligned} \tag{3}$$

For unsupervised summarization, we train from scratch a T5 model consisting of 12 encoder and 12 decoder layers. We are unable to use CodeT5 off the shelf for these experiments as it is pretrained on Ruby code from CodeSearchNet, meaning that it has already seen the code-only examples for the unsupervised task. This entangles the effects of our MLM objective with those of pretraining. Moreover, it is more realistic for the unsupervised target language to be one that the model has not previously seen.

Preliminary results in Table 7 show that our MLM objective clearly outperforms a naive baseline that does not use the code-only data, though it remains to be seen how kNN retrieval fits into this picture. Unfortunately, due to time constraints at the end of the semester, I have not been able to complete these experiments (though I plan to revisit this at some point after this semester).