CodeKs: Code Generation Using kNN Models

Alex Xie[1], **Jinqi Chen**[1], **Jingyun Yang**[2*]
School of Computer Science[1], Machine Learning Department[2]
Carnegie Mellon University
Pittsburgh, PA 15213
`{alexx, jinqic, jingyuny}@andrew.cmu.edu`

Abstract

Transformer-based models have achieved state-of-the-art results in the code generation task. Nevertheless, these generation methods have difficulty memorizing rare patterns in the dataset, which often lead to sub-optimal performance in code generation tasks. We present CodeKs, a novel framework for code generation that leverages the non-parametric retrieval models. Our framework consists of two modules: (1) a transformer-based code generation module for predicting future token probabilities and embedding vectors that can be used for non-parametric retrieval; (2) a non-parametric retrieval module based on kNN-LM which uses the embedding vectors by the generation module as context to search for top k tokens at the next timestamp. Our method outperforms the state-of-the-art model on CoNaLa dataset using BLEU metric, proving that non-parametric methods, which store context-token relationships explicitly, has benefits over parametric methods.

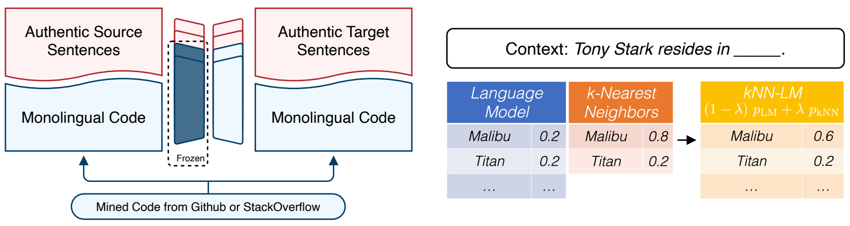
1 Introduction

Natural language to code generation is the task of predicting a program or a code snippet given a natural language intent or description. This is a task with wide practical applications, ranging from in-IDE code auto-completion tools to end-user programming, in which non-programmers are able to specify code using natural language. Earlier code generation methods require domain-specific inductive biases like syntax trees and thus do not easily extend to new domains. More recent code generation methods that adopt the pre-train/fine-tune paradigm have showed promising results (Wang et al., 2021; Chen et al., 2021; Feng et al., 2020). The vast majority of these approaches are *parametric*, making predictions based only upon a set of learned weights. This puts a large burden on these models to capture in their weights all necessary knowledge about both natural language and programming language.

In contrast, *non-parametric approaches* make predictions by explicitly referencing examples in an external datastore and combining a distribution obtained from these examples with the original distribution outputted by a parametric model. One such approach of interest, the kNN-LM, uses an L2 distance metric to measure the similarity between the query context and the contexts in the datastore, and items that are closer to the query will have higher probability to be retrieved. One widely adapted metric is the L2 distance metric because of its simplicity.

In this paper, we introduce **CodeKs**, a novel framework for code generation using non-parametric kNN models (and which *coincidentally* shares the same pronunciation as the well-known code generation model **Codex** (Chen et al., 2021)). While OpenAI Codex is trained by GPT-3 using parametric methods on billions lines of source code, we show that using non-parametric methods have benefits over the parametric methods and can achieve better performance. In particular, we use a kNN-LM (k-nearest neighbor language model), which computes the top k tokens at the next timestamp given the natural language intent and the previously generated code as the context. The advantages are two-folds: (1) kNN-LM does not implicitly memorize the context-token pairs in the weights of a large neural network, but explicitly store them in a kNN datastore, which lessen the burden of the last decoding layer and the training model itself; (2) kNN-LM directly optimize the distance between the context and the next token, which is a more effective way to achieve higher performance especially when the evaluation metric is not differentiable (i.e. BLEU score).

[*]Alphabetical order


Authentic Source Sentences
Monolingual Code
Frozen
Authentic Target Sentences
Monolingual Code
Mined Code from Github or StackOverflow
Context: *Tony Stark resides in* _____.
Language Model
k-Nearest Neighbors
kNN-LM
$(1-\lambda)\ p_{\text{LM}} + \lambda\ p_{\text{kNN}}$
Malibu 0.2
Titan 0.2
Malibu 0.8
Titan 0.2
Malibu 0.6
Titan 0.2

(a) TAE

(b) KNN-LM

Figure 1: Our method combines the benefit of generation-based and retrieval-based models for code generation by leveraging **BERT-TAE** as the base seq2seq code generation model and **kNN-LM** for non-parametric retrieval.

We evaluate our method on the CoNala dataset, a commonly-used dataset for code generation tasks, and use BLEU score as the performance metric. We use the pretrained BERT-TAE, a state-of-the-art model on CoNala dataset, and freeze its weights for non-parametric retrieval in our experiments. We find that CodeKs achieves higher performance on the CoNala dataset than the BERT-TAE model, proving that non-parametric methods bring benefits over parametric methods. All our code is available in the GitHub repository.[1]

2 Related Work

Approaches to natural language to code generation can be broadly separated into two categories, **generation-based** and **retrieval-based**. Generation based approaches rely on the model to predict code using information implicitly stored in its parameters; this is an intrinsically difficult and complex task, requiring a model to possess deep lexical, syntactic, and semantic knowledge about both source code and natural language. As such, many approaches have incorporated inductive biases such as syntax trees and grammars as intermediate representations (Rabinovich et al., 2017; Yin and Neubig, 2018; Guo et al., 2019); however, these approaches are often not easily extensible to new domains or settings (i.e. a different programming language) due to the relatively large amount of expert hand-crafting required.

More recent generation approaches have adopted the **pre-train** and **fine-tune** paradigm, pre-training a model in an semi-supervised or unsupervised manner on large amounts of code and text before fine-tuning it for specific downstream code generation tasks. For instance, Xu et al. introduce a framework for pre-training models on noisy NL/code pairs along with external API documentation and apply it to an existing tree-structured model, yielding significant improvements (Xu et al., 2020).

Further, pre-training approaches have been shown to be able to at least in part make up for a lack of inductive biases in models. Large pre-trained transformer models such as CodeBERT, PL-BART, and CodeT5 yield state-of-the-art performance on a number of benchmarks despite using only linear representations of code and having little notion of explicit structure (Feng et al., 2020; Ahmad et al., 2021; Wang et al., 2021). Similarly, Norouzi et al. find that an encoder-decoder transformer model pre-trained to autoencode source code outperforms methods with heavy inductive bias, setting a new state-of-the-art for the CoNaLa code generation dataset in the process (Norouzi et al., 2021).

Meanwhile, code retrieval require a model to select the most relevant code examples from a database given an intent or query. In practice, developers most commonly perform code retrieval using Google or other traditional search engines. However, various code-specific information retrieval as well as neural network based approaches have been proposed for the task (Husain et al., 2019). Neural approaches generally attempt to learn alignments between natural language queries and source code examples; Gu et al. propose using pseudo-Siamese networks for code search, with the objective of minimizing the distance between relevant code and query representations (Gu et al., 2021).

More recently, there have been efforts to combine generation and retrieval methods, for both NLP tasks in general (Lewis et al., 2020) and code generation in particular (Hayati et al., 2018). One such approach is the k nearest neighbor language model (**kNN-LM**), which interpolates between the next-token distribution of a pre-trained language model and a distribution corresponding to the k most similar contexts retrieved from a datastore of

[1]https://github.com/Kathryn-cat/TAE

context/next token pairs (Khandelwal et al., 2020). While originally formulated for language modeling, recent work has shown that kNN-LMs are readily adapted to sequence-to-sequence tasks such as machine translation (Khandelwal et al., 2021) and that they are effective for code language modeling (an unconditional generation task) (Xu et al., 2021). Parvez et al. propose a retrieval-augmented model they term REDCODER, which generates code conditioned upon examples retrieved prior to generation from searching a database with the natural language intent (Parvez et al., 2021).

3 Methods

We illustrate our method in Figure 2. In the following sections, we first describe how we leverage non-parametric models to perform next token prediction in section 3.1, then describe how we build an appropriate context vector for our non-parametric model in section 3.2.

3.1 Non-parametric Retrieval and kNN-LM

In this work, we use a kNN-LM for retrieval (Khandelwal et al., 2020). This model linearly interpolates the output of a pretrained model using a k-nearest neighbors (kNN) model, and the distances are computed based on the distances in the embedding space. kNN-LM has been proved to have more efficient domain adaptation, because it only needs the varying of the kNN datastore without further training of the pretrained model in order to adapt to larger training sets. kNN-LM can also achieve higher perplexity and output more reasonable results, because the model explicitly recognize patterns in a datastore instead of implicitly memorizing them in model parameters, which also allows the model to capture rare patterns well. We will further demonstrate the benefits of kNN-LM over linear layer decoding in the Results and Conclusion sections.

In our architecture as illustrated in Figure 2, we use a kNN-LM component on top of the baseline architecture, BERT-TAE, which consists of a BERT encoder and 4-layer decoder with copy attention (Norouzi et al., 2021). We create a kNN datastore for retrieval, where the keys are the embeddings of the natural language intent and previous code output which are taken from the last hidden layer of the TAE decoder, and the values are the embeddings of the target next token. Hence, our datastore contains an entry for every target code token. During retrieval, we use a pretrained TAE model and make one complete pass over the data used in the datastore to obtain the desired context embeddings. We use the `faiss` library (Johnson et al., 2017) for efficient k-nearest neighbors search. `faiss` library initially separates the vectors in datastore into a pre-defined number of clusters, and probes a set of clusters for quick k-nearest neighbors search during retrieval. Then, at test time, along with standard greedy or beam search, we iteratively find the next token distribution using kNN-LM until we get the full code output.

To formalize non-parametric retrieval and kNN-LM, consider the language modeling problem, where we aim to estimate $p(w_t \mid c_t)$, the probability distribution of the token at the next timestamp w_t over the context $c_t = (w_1, w_2, ..., w_{t-1})$. In kNN-LM, we take the contexts c_t and the target next tokens w_t from the data collection D, map the contexts c_t using a frozen pre-trained TAE model to obtain $f(c_t)$, and store them in a datastore. The datastore consists of key-value pairs

$$(\mathcal{K}, \mathcal{V}) = \{(f(c_i), w_i) | (c_i, w_i) \in \mathcal{D}\} \qquad (1)$$

During non-parametric retrieval, let $d(c_q, k_i)$ denote the L2 distance between the embedding of query context c_q and one of the keys k_i in datastore. The k-nearest neighbors objective is to find the k keys k_i which corresponds to the k shortest distances $d(c_q, k_i)$. We further normalize the k distances using a softmax-like approach using temperature T, where $p(k_i \mid c_q) = \frac{e^{-d(c_q,k_i)/T}}{\sum_{i=1}^{k} e^{-d(c_q,k_i)/T}}$. However, the k keys might contain duplicates, so we further aggregate the normalized probability mass over each token, resulting in the following kNN-LM:

$$p_{kNN}(w_i \mid c_q) = \sum_{i=1}^{k} \mathbb{1}_{w_i = k_i} \frac{e^{-d(c_q,k_i)/T}}{\sum_{i=1}^{k} e^{-d(c_q,k_i)/T}} \qquad (2)$$

In order to obtain non-zero probabilities for every possible next token like a normal language model does, we use a interpolation parameter λ to linearly interpolate between the results of a kNN-LM and the results of the pre-trained TAE. The final probabilistic distribution for the non-parametric retrieval is

$$\begin{aligned} p(w_i \mid c_q) =& \lambda p_{kNN}(w_i \mid c_q) + \\ & (1-\lambda) p_{TAE}(w_i \mid c_q) \end{aligned} \qquad (3)$$

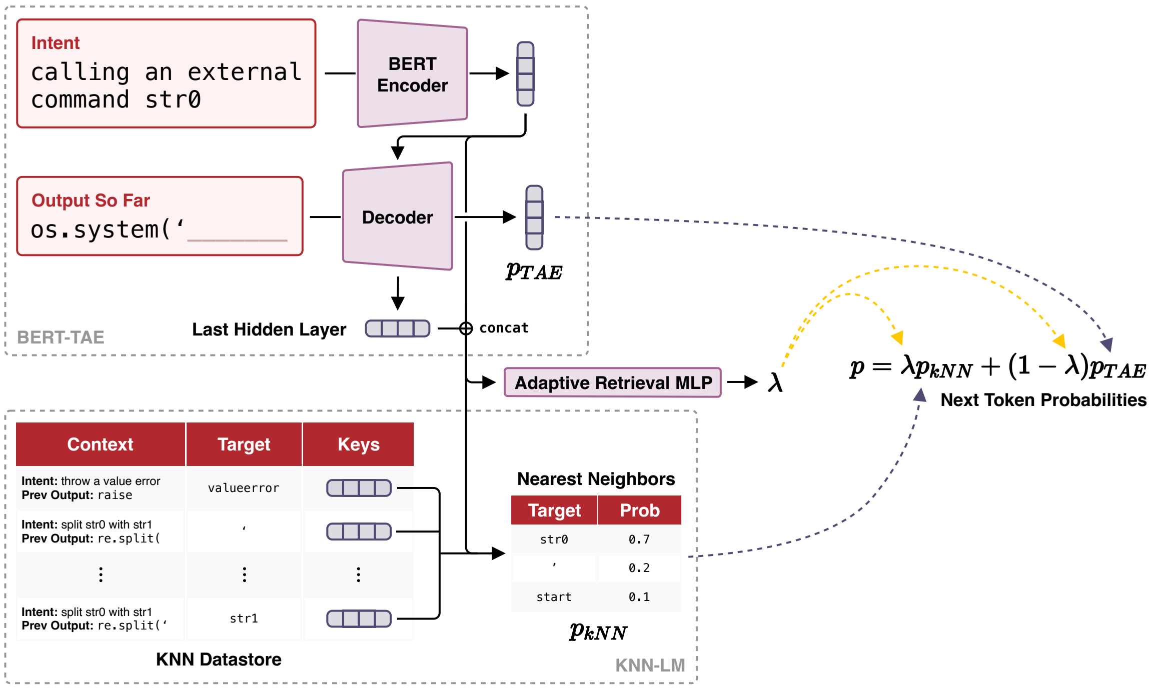


Figure 2: Overview of our method.

3.2 Using Encoder Context

In the vanilla kNN-MT (Khandelwal et al., 2021), datastore keys are derived from some intermediate decoder representation, $f(s, t_{1:i-1}) = f_{dec}(f_{enc}(s), t_{1:i-1})$, where f_{enc} denotes the encoder network and f_{dec} denotes the decoder network (or some subset of the decoder network, typically the output of the final self-attention layer). This simple approach has been found to be sufficient for machine translation, in which the base seq2seq models are already of relatively high quality and have been highly optimized and widely studied (Ng et al., 2019). However, in the case of code generation, particularly on the relatively challenging CoNaLa dataset, existing models such as BERT-TAE are less mature and of lower quality. As such, we hypothesize that the intermediate decoder representation is less capable of capturing necessary information, particularly that of both the previously generated output tokens and the input sequence. This leads to subpar representations which may fail to map similar contexts to the same space. In particular, this might cause the kNN component to ignore the input intent as well as earlier generated tokens altogether, causing retrieval to become influenced solely by recently generated code.

To address this problem, we propose using the concatenation of the intermediate decoder representation with the final encoder output as our datastore keys. Specifically, we take the contextual embedding corresponding to the ⟨CLS⟩ token as a pooled representation of the input intent. Hence, we derive datastore keys via the following mapping f'.

$$h = f_{enc}(s) \tag{4}$$

$$f'(s, t_{1:i-1}) = [h_{CLS}; f_{dec}(h, t_{1:i-1})] \tag{5}$$

However, this doubles the length of each key from 768 to 1536, greatly slowing down datastore retrieval. Thus, we perform PCA on the datastore keys to reduce dimensionality to 512. This follows He et al., who find that reducing datstore dimensionality via PCA yields very marginal decreases (and in some cases, increases) in performance, while significantly speeding up retrieval (He et al., 2021).

4 Experimental Evaluation

4.1 Dataset and Preprocessing

In all experiments, we use the CoNaLa dataset (Yin et al., 2018), which consists of 2k human-annotated train pairs, 500 annotated test pairs, and an additional 600k train pairs automatically mined from StackOverflow (of which we use the 100k most probable as assigned by the mining model, following BERT-TAE). Each pair consists of a natural language intent specifying some action along with a code snippet implementing the action.

We construct our kNN datastores also using CoNaLa data, with one version containing only annotated train data and the other containing annotated train data along with mined data. As shown in Table 1, the mined data greatly outnumbers the original train data. While this increases the likelihood that patterns at test time will be present in the datastore, it also introduces a significant amount of noise, as the mined data may include incorrect pairs where the code snippet does not correspond to the intent. However, we hypothesize that on average, the positive effect of having more data will outweigh the negative effect of increased noise, particularly as the kNN component should in theory be capable of filtering out noisy data by not selecting it as a nearest neighbor.

	Examples	Datastore Size
Train	2k	45k
Test	500	-
Mined	100k	2086k

Table 1: Dataset and datastore sizes

4.2 Experiments

We evaluate the kNN-LM with a frozen pretrained BERT-TAE model (Norouzi et al., 2021), which is the state-of-the-art model on CoNala dataset; to ensure comparability, we use the pretrained checkpoint weights provided by the authors[2]. We create datastore for both the training data and the mined data of CoNaLa, then we use the method of incorporating the encoder context and evaluate the kNN-LM on the pretrained TAE as discussed in the Methods section. For comparability with past work, we use two evaluation metrics: exact match (proportion of generated snippets that are exactly equal to the ground truth) and BLEU score. However, due to the difficulty of the dataset, exact match accuracy is almost always very low, providing relatively little insight on the performance of the model. Hence, in practice, BLEU score, while not perfect, is a much more reasonable measure of the quality of the generated code.

We have the following hyperparameters: k (the number of nearest neighbors), λ (the interpolation parameter between p_{kNN} and p_{TAE}), T (temperature in kNN-LM objective), N_{probe} (number of clusters to consider in `faiss` library). In all experiments, unless otherwise specified, we use $k = 64$, $\lambda = 0.05$, and $T = 1000$. Empirically, we keep $N_{probe} = 32$ fixed. The full results for different hyperparameters and their BLEU score are shown in Figures 3, 4, 5. Additionally, we use beam search in inference with beam size 10.

[2]https://github.com/BorealisAI/code-gen-TAE

5 Results and Discussion

5.1 Quantitative Results

We compare the performance of the BERT-TAE baseline with various kNN-enhanced variants. Contrary to our expectations, we observe no improvements from our kNN approach when naively adding (i.e. without adding encoder context) either the annotated or combined annotated+mined datastore. In particular, using only annotated training data in the datastore significantly reduces BLEU score performance (though interestingly, it does not affect exact match). This is likely due to the small size of the annotated training set; as a consequence, the datastore might not contain any relevant entries for a large proportion of the queries made at test time, instead returning a noisy distribution that corrupts the parametric model's predictions.

Similarly, when using the datastore consisting of both annotated and mined data, we observe virtually zero improvement over the baseline. We attribute this in part to the failure of the naive encoding scheme to fully encode the necessary information in the context, as we find in qualitative assessments of nearest neighbors that contexts very different from the current generation context are often retrieved. Additionally, many queries at test time still fail to retrieve suitable contexts from the data store; this suggests a gap between the distribution of test data and the distribution of mined data, which cannot be easily addressed without simply collecting more data.

Finally, we see modest improvement over the baseline when we add encoder context to the annotated+mined datastore. This indicates that explicitly adding the encoder representation of the input sequence enhances the datastore representation, in part addressing the issues of the naive decoder-only representation. However, given the somewhat limited improvement, this representation may still be less than ideal.

	BLEU	Exact Match
BERT-TAE	33.41	**3.4**
+ annotated datastore	33.17	3.4
+ mined datastore	33.41	2.6
+ encoder context	**33.50**	2.6

Table 2: Quantitative Results

5.2 Effects of Hyperparameters

In Figure 3, 4, 5, we show the effect of each hyperparameter individually, keeping other hyperparameters fixed. For the number of nearest neighbors k, we observe that the model performs best when $k = 64$. However, we observe significant variability between k and model performance, with a sharp peak at 64 and significant drop-offs on either side. However, the drop-off as k-increases is not as large as k increases, which somewhat follows the intuition that increasing k increases the probability that ground truth tokens are included in the retrieval results. At the same time, however, increasing k also increases the noisiness of the retrieval. Hence, $k = 64$ appears to be the "sweet spot" in this tradeoff.

For the interpolation parameter λ, we find that $\lambda = 0.05$ behaves optimally. The value of λ can be interpreted as a reflection of the relative performance between the kNN-LM and BERT-TAE models. Thus, the fact the optimal λ is relatively small indicates that the BERT-TAE contributes much more strongly to predicting the correct token than the kNN component. Indeed, in qualitative analyses of the nearest neighbors, we found that in the overwhelming majority of cases, the nearest neighbors distribution dragged down the parametric model's confidence in the correct next token, likely due to the noisiness of the retrieval mechanism. Nevertheless, from our quantitative results, we see that the kNN component does in certain cases guide the model toward the correct answer. In addition, it is possible that the fixed value λ prevents the model from making the best use of the nearest neighbors distribution; to that end, a dynamic λ based on the query might yield better results. We further discuss a method for choosing λ adaptively in the Appendix.

Lastly, the temperature T for the softmax-like function in kNN-LM shows a general trend that higher T results in better (or roughly equivalent) performance. When T gets higher, the softmax distribution gets softer, which means that kNN-LM pushes the prediction of each token to be a little more equal. This helps to prevents the model from overfitting to the nearest neighbors distribution.

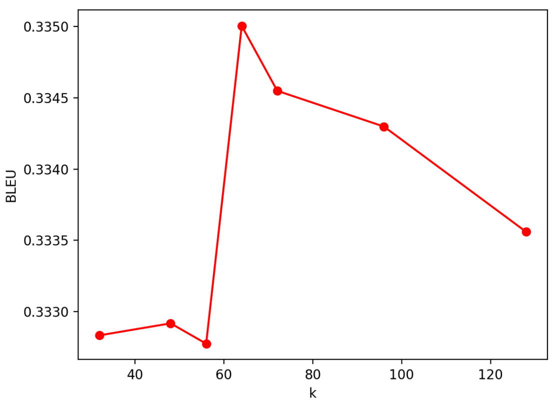


Figure 3: Effect of number of nearest neighbors k for $k \in \{32, 48, 56, 64, 72, 96, 128\}$.

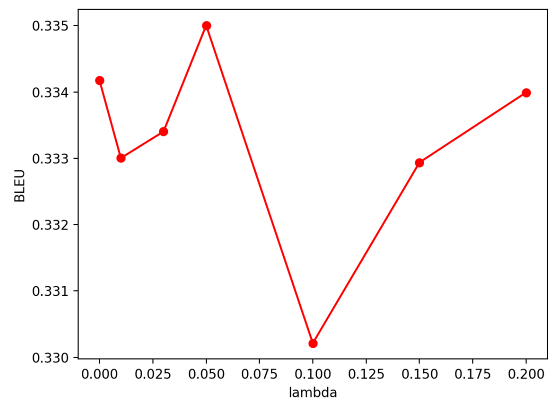


Figure 4: Effect of interpolation parameter λ for $\lambda \in \{0, 0.01, 0.03, 0.05, 0.1, 0.15, 0.2\}$. Note that $\lambda = 0$ is equivalent to the baseline model.

5.3 Qualitative Results

In table 3, we show qualitative outputs of our framework on the test set. In each row, we show, in order, the intent used as input to our model, the ground truth code, and the prediction produced by our model CodeKs. In rows 1-5 of the table, we show successful cases, where our framework produces a predicted code sequence that is identical to ground truth. In row 6 of the table, we show a sample where our model produces a code sequence that is correct but is substantially different from the ground truth.

#	Intent	Ground Truth	Prediction
1	`get the length of list var0`	`len(var0)`	`len ( var0 )`
2	`change working directory to var0`	`os.chdir(var0)`	`os. chdir ( var0 )`
3	`get attribute str0 from object var0`	`getattr(var0, 'str0')`	`getattr ( var0,'str0')`
4	`split string var0 based on occurrences of regex pattern str0`	`re.split('str0', var0)`	`re. split ('str0 ', var0 )`
5	`check if object var0 is a string`	`isinstance(var0, str)`	`isinstance ( var0, str )`
6	`read keyboard - input`	`input('enter your input:')`	`print ( input ('please enter something :') )`
7	`get the position of item var0 in list var1`	`print(var1.index(var0))`	`print ( var1. var0 )`
8	`make a delay of 1 second`	`time.sleep(1)`	`sleep ( 1 )`
9	`append list var0 to var1`	`var1.extend(var0)`	`var1. append ('var0')`
10	`get an absolute file path of file str0`	`os.path.abspath('str0')`	`os. path. expanduser ('str0')`

Table 3: Qualitative Results of Our Framework

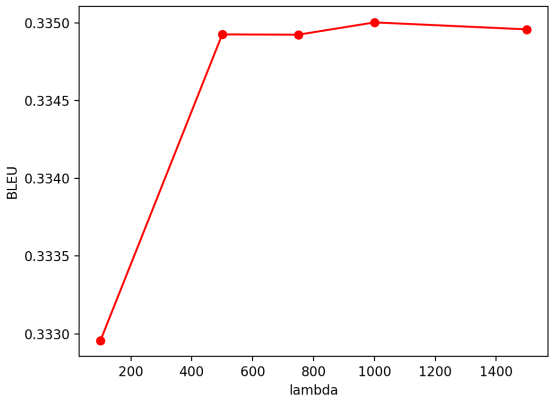


Figure 5: Effect of temperature T for $T \in \{100, 500, 750, 1000, 1500\}$

In rows 7-10, we show some failure cases of our framework. In rows 7-8, we show that our model lacks basic notion of syntax. The model does not understand that variables such as `var0` cannot be used as an attribute of `var1` and is not aware that the method `sleep` cannot be called without mentioning the package name `time` before it. In row 9, we show a sample where the model confuses variables with strings, appending the name of the variable to the list rather than the variable itself. In the last row, we show one other common failure of our model, where the kNN retrieval gave a related output token, which in this case is `expanduser`, but fails to generate the correct output token.

6 Conclusion and Future Work

We presented CodeKs, a novel framework for code generation that leverages non-parametric models. We proposed to use BERT-TAE to produce context vectors and use kNN-LM as the method to perform non-parametric next token retrieval. In the CoNaLa dataset, we showed that our framework outperforms competing baselines that do not use non-parametric models. Extending our framework to train an embedding vector for kNN retrieval, making our framework aware of syntax correctness, and extending our framework to datasets with more complicated code sequences while using non-parametric models to retrieve longer reusable code sequences are clear avenues for future work.

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.

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.

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

Jian Gu, Zimin Chen, and Martin Monperrus. 2021. Multimodal representation for neural code search. *ICSME*.

Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2019. Towards complex text-to-sql in cross-domain database with intermediate representation. *arXiv preprint arXiv:1905.08205*. Version 2.

Shirley Anugrah Hayati, Raphael Olivier, Pravalika Avvaru, Pengcheng Yin, Anthony Tomasic, and Graham Neubig. 2018. Retrieval-based neural code generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 925–930. Association for Computational Linguistics.

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

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

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)*.

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.

Nathan Ng, Kyra Yee, Alexei Baevski, Myle Ott, Michael Auli, and Sergey Edunov. 2019. Facebook fair's wmt19 news translation task submission. In *Proc. of WMT*.

Sajad Norouzi, Keyi Tang, and Yanshuai Cao. 2021. Code generation from natural language with less prior knowledge and more monolingual data. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 776–785, Online. Association for Computational Linguistics.

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.

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, Junxian He, Graham Neubig, and Vincent J. Hellendoorn. 2021. Capturing structural locality in non-parametric language models.

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*.

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 Adaptive Retrieval

In this section, we describe a method that we tried but does not work as we expect, which we hope to use to further improve the performance of our results when allowed more time in the future.

Since retrieval from a large datastore can add huge burdens to the inference, in addition to dimensionality reduction through PCA as mentioned in the previous section, we consider adaptive retrieval, a method which estimate the interpolation parameter λ automatically (He et al., 2021). The insight to the approach is that $p_{kNN}(w_i \mid c_q) \geq p_{TAE}(w_i \mid c_q)$ might happen less than half of the time, meaning that the careful choice of λ is very important. In order to estimate the value of λ, we introduce a second neural network g, where $\lambda = g(c_i)$ depends on the context (unlike He et al., we do not use additional textual features). We directly maximize the full objective as follows

$$\mathcal{L} = \frac{1}{N} \sum_i [\log p(w_i \mid c_q; g(c_q)) + a \cdot g(c_q)] \tag{6}$$

The probability distribution $p(w_i \mid c_q; g(c_q))$ is the same as the full objective mentioned earlier despite conditioning on the learned interpolation parameter, and the term $a \cdot g(c_q)$ is introduced as a L1-regularization to the objective to prevent overfitting and prune unnecessary retrievals. For training g, we split the CoNaLa validation data into two parts and take one part for training, without using any of the actual CoNaLa training data.

We further use the adaptive retrieval method as mentioned previously. However, the λ predicted by the g function collapses to either 0 or 1, which is not ideal. A possible explanation for this is that a portion of the CoNaLa validation set contains far few samples, and the g function might experiences some kind of overfit.