Efficient Text-to-Code Retrieval with Cascaded Fast and Slow Transformer Models

Anonymous Author(s)

ABSTRACT

The goal of semantic code search or text-to-code search is to retrieve a semantically relevant code snippet from an existing code database using a natural language query. When constructing a practical semantic code search system, existing approaches fail to provide an optimal balance between retrieval speed and the relevance of the retrieved results. We propose an efficient and effective text-to-code search framework with cascaded fast and slow models, in which a fast transformer encoder model is learned to optimize a scalable index for fast retrieval followed by learning a slow classification-based re-ranking model to improve the accuracy of the top K results from the fast retrieval. To further reduce the high memory cost of deploying two separate models in practice, we propose to jointly train the fast and slow model based on a single transformer encoder with shared parameters. Empirically our cascaded method is not only efficient and scalable, but also achieves state-of-the-art results with an average mean reciprocal ranking (MRR) score of 0.7795 (across 6 programming languages) on the CodeSearchNet benchmark as opposed to the prior state-of-the-art result of 0.744 MRR. Our codebase will be made publicly available.

1 INTRODUCTION

Building automatic tools that can enhance software developer productivity has recently garnered a lot of attention in the deep learning and software engineering research communities. Code retrieval systems can make developers more productive by enabling them to search and reuse from the enormous volume of open-source repositories available online, thus speeding up the software development lifecycle. Code search systems can be particularly of great value for organizations with internal proprietary code. Indexing source code data internally for search can prevent redundancy and boost programmer productivity. A study by Xu et al. [61] surveys developers to understand the effectiveness of code generation and code retrieval systems. Their results indicate that the two systems serve complementary roles and developers prefer retrieval modules over generation when working with complex functionalities, thus advocating the need for better code search systems. Beyond their direct utility in improving developer productivity, code search solutions have also been leveraged to improve the performance of code generation systems, [36, 42, 47, 65] when used as sub-components, thus adding to the significance of research on improving text-to-code retrieval.

Our primary focus in this work is to improve the performance of text-to-code search solutions, as evaluated by these two aspects: the speed of retrieval and the relevance of the retrieved results to the input query. We propose to bring this improvement by cascading two approaches with complementary strengths - fast retrieval systems that sacrifice relevance but offer high retrieval speed, and slow retrieval systems that sacrifice speed but return results with higher relevance.

We find inspiration for this multi-stage approach from recent progress in the text-to-image retrieval domain [12, 29, 30, 38]. Researchers have shown impressive results on the traditional document retrieval problem (text-to-text search) using transformer based models [24, 43, 60]. This progress has in turn guided a lot of research in the text-to-code retrieval domain [13, 18]. Parallel to the progress in natural language processing (NLP), language models (LMs) pre-trained on code like CodeBERT [13], CodeGPT [37], InstructGPT [46], Codex [8], PLBART [1] and CodeT5 [59] have now been proposed for understanding and generation tasks involving programming languages. However, there has been very limited research [58] on studying the similarities between the two problem settings of text-to-image and text-to-code retrieval despite their common theme of aligning data from two different modalities. We believe further improvements in text-to-code search can be achieved using the two stage paradigm that has been shown to be effective for text-to-image search.

One could question the pertinence of text-to-code search given the current state of research on code generation using transformer based large language models (LLMs). Chen et al. [8]'s 12B parameter Codex, Li et al. [31]'s 41B parameter AlphaCode, Nijkamp et al. [44]'s 16B parameter CodeGen and Austin et al. [2]'s 137B parameter LM use large scale autoregressive language models to demonstrate impressive capabilities of generating multiple lines of code from natural language descriptions, well beyond what previous generation models like GPT-C [52] could accomplish. Would developers need text-to-code search when LLMs trained on code can generate correct looking programs for a natural language prompt? We argue that text-to-code retrieval would still be a valuable offering for developers for the following reasons: The impressive performance of code generation systems is often predicated on being able to draw many samples from the model [28] and machine-check them for correctness. This setup will often not be the case in practice [10]. Code generation models also entail security implications (possibility of producing vulnerable or misaligned code) [8, 26, 48], making their adoption tricky. Besides, some recent studies [34] have found limitations with popular benchmarks like HumanEval that have been relied on to measure the correctness of model generated programs, suggesting that the synthesized program correctness scores of code LLMs have been overestimated.

Given this current landscape, code retrieval systems can serve as attractive alternatives when building tools to assist developers. With efficient implementations, code search for a single query can typically be much faster for most practical index sizes than generating code with large scale LMs. As opposed to code generation, code retrieval offers interpretability and the possibility of a much greater control over the quality of the result as the index entries can be verified beforehand. Another benefit with code search systems is

ESEC/FSE 2023, 11 - 17 November, 2023, San Francisco, USA
2023. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

the ability to leverage additional data post training as this simply requires extending the index by encoding new instances. Moreover, a code generation system can be augmented with a code retrieval system to improve the generation ability [47].

For semantic code search, deep learning based approaches [16, 18, 51, 62] involve encoding query and code independently into dense vector representations in the same semantic space. Retrieval is then performed using the representational similarity (based on cosine or euclidean distances) of these dense vectors. This framework is often referred with different terms like representation/embedding based retrieval [21], dense retrieval [24], two tower [15], or *fast*/dual encoder [12, 40] approach in different contexts. An orthogonal approach involves encoding the query and the code jointly and training semantic code search systems as binary classifiers that predict whether a code answers a given query [20, 37] (referred to as monoBERT style [45] or as *slow* classifier). With this approach, the model processes the query paired with each candidate code sequence, meaning the text and code snippets are concatenated at the input stage of the neural network. Intuitively, this approach helps to sharpen the cross information between query and code and is a better alternative for capturing matching relationships between the two modalities (natural language (NL) and programming language (PL)) than the simple similarity metric between the *fast* encoder based sequence representations. While this latter approach can be promising for code retrieval, previous works have mostly leveraged it for tasks like text to code generation or binary classification in the form of text-code matching [37]. Directly adapting this approach to code search tasks would be impractical due to the large number of candidates to be considered for each query. Inference with this setup would require each candidate to be combined with the query and passed through the classifier. We depict the complementary nature of these approaches in Figure 1 when using a transformer [55] encoder based model for retrieval and classification.

In order to leverage the potential of such nuanced classifier models for the task of retrieval, we propose a cascaded scheme (CasCode) where we process only a limited number of candidates with the classifier model. This limiting can be achieved by employing the representation based (*fast* encoder) approach and picking its top few candidate choices for processing by the second classifier stage. Our cascaded approach leads to state of the art performance on the CodeSearchNet benchmark with an overall mean reciprocal ranking (MRR) score of 0.7795, significantly improving over previous results (best reported MRR score of 0.744 from Guo et al. [17]). We propose a variant of the cascaded scheme with shared parameters, where a single transformer model can serve in both the modes - encoding in the representation based retrieval stage and classification in the second stage. This shared variant can be achieved by multi-task training [6, 50] using the sum of the two objectives corresponding to these two distinct task settings. CasCode's shared variant substantially reduces the memory requirements, while offering retrieval performance that is comparable to the separate variant with an MRR score of 0.7700. We also show improvements with our CasCode approach for the AdvTest python dataset popularised by the CodeXGLUE benchmark [37] to assess the generalization abilities of code retrieval models when the function and variable names of a program are normalised and thus unrelated to its semantics.

Figure 2 illustrates the trade off involved between inference speed and retrieval performance (MRR) for different algorithmic choices, where we have the (*fast*) encoder model on one extreme, and the (*slow*) classifier model on the other. With CasCode, we offer performance comparable to the optimal scores attained by the classifier model, while requiring substantially lesser inference time, thus making it computationally feasible.

Our key contributions in this paper are the following.

- We first show that the performance of existing dense retrieval models (CodeBERT and GraphCodeBERT) trained with contrastive learning can be significantly improved when trained with larger batch-size, these serve as stronger baselines for code retrieval.
- To further push retrieval performance, we propose the cascaded code search scheme (CasCode) that performs code retrieval in two stages, and we analyze the trade-off of the inference speed and retrieval performance.
- We show that the transformer models in the two stages of CasCode can be shared by training in a multi-task manner, which significantly reduces the memory requirements.
- With CasCode, we report state of the art text-to-code retrieval performance on public benchmarks of CodeSearchNet and the normalised AdvTest (Python) dataset from CodeXGLUE

2 RELATED WORK

Our work is heavily inspired by recent progress in neural search and ranking for natural language, where pre-trained transformer language models have been extensively used. Karpukhin et al. [24] finetune BERT [11] based encoders to build the passage retrieval component of their open domain question answering (QA) system, where the goal is to develop systems capable of answering questions without any topic restriction. Efficient passage retrieval to select candidate contexts is a critical step in such pipelines. Xiong et al. [60] show improvements in transformer based dense retrieval of text by using globally retrieved hard negatives when finetuning the encoders, resulting in effective performance on web search and QA. Chang et al. [7] propose novel pre-training objectives to train transformer models that specialize at embedding-based large-scale text retrieval.

Lin et al. [33] provide an exhaustive survey on the use of pre-trained language models for text ranking and study the trade offs involved in the different alternatives. In the single stage fashion, a common approach is representation based ranking, where BERT-based models (bi-encoders or *fast* encoders) are trained to independently encode the query and documents, and inference involves dot product based similarity search for retrieval [13, 18]. Another single stage approach is monoBERT [20, 45] (*slow* classifier), where query-document pairs are passed jointly to a BERT encoder and the model predicts whether the input document is relevant to the query or not. The monoBERT approach is computationally more expensive, but also tends to be more accurate than the bi-encoder approach. However, with the bi-encoder approach we can index all the document representations offline. Thus at inference time, we simply need to encode the query, making it a very attractive retrieval setup. Achieving this inference speedup by caching representations is not possible in the monoBERT setting, as it jointly

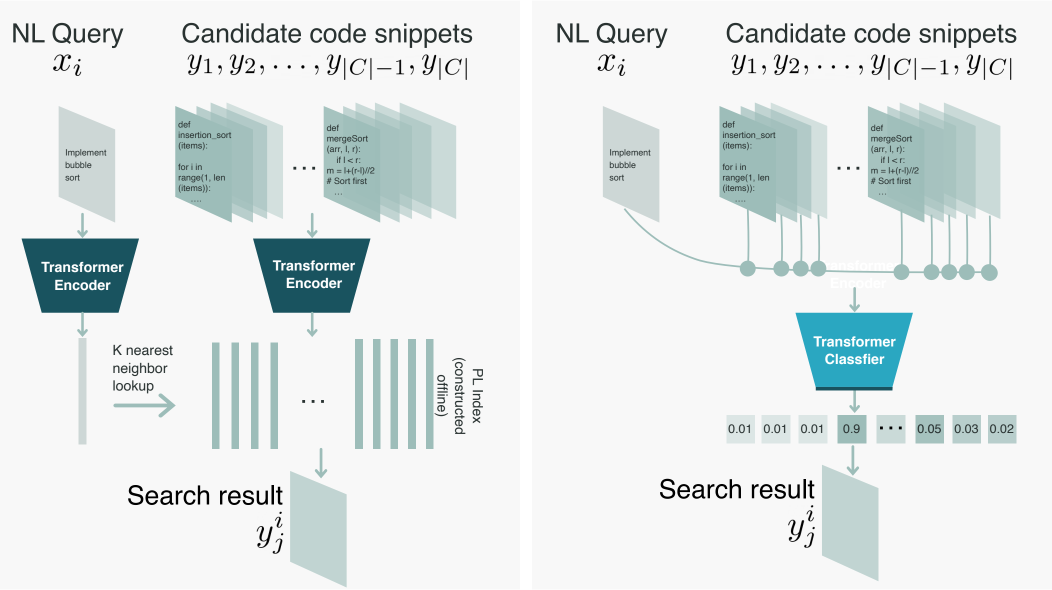
NL Query x_i
Candidate code snippets $y_1, y_2, \ldots, y_{|C|-1}, y_{|C|}$
Implement bubble sort
def insertion_sort (items): for i in range(1, len (items)): ...
def mergeSort (arr, l, r): if l < r: m = l+(r-l)//2 # Sort first ...
Transformer Encoder
Transformer Encoder
K nearest neighbor lookup
PL Index (constructed offline)
Search result y_j^i
NL Query x_i
Candidate code snippets $y_1, y_2, \ldots, y_{|C|-1}, y_{|C|}$
Implement bubble sort
Transformer Classfier
0.01 0.01 0.01 0.9 ... 0.05 0.03 0.02
Search result y_j^i

Figure 1: Illustration of the the inference stage of *fast encoder* (left) and *slow classifier* (right) based semantic code search approaches. With the encoder based approach, we independently compute representations of the natural language (NL) query and candidate code sequences. The code snippet with representation nearest to the query vector is then returned as the search result. With the classifier based approach, we jointly process the query with each code sequence to predict the probability of the code matching the query description. The code sequence corresponding to the highest classifier confidence score is then returned as the search result.

processes the query and document strings. As an alternative to these two frameworks, Khattab and Zaharia [25] propose ColBERT which performs late interaction between a query and document after their independent encoding. This leads to performance that is comparable to the monoBERT approach, but is less computationally expensive during inference. However, ColBERT requires storing per token representations of all the document candidates as inputs to the late interaction, and this can demand impractically high storage. The limitations of monoBERT when handling a large number of candidate documents inspire the need for multi-phase retrieval, where the first phase can retrieve candidate documents with the cost effective bi-encoder approach (dot-product retrieval), followed by the second stage where only the top candidates from the first stage are processed by a more expensive monoBERT model. For code retrieval, we experimentally show that these two models can share a majority of their parameters. Thus, a single encoder backbone can serve in the two stages - first as the bi-encoder for fast retrieval, and then as the more powerful monoBERT.

Early work on neural approaches to code search [51] leveraged unsupervised word embeddings [39, 49] to represent code snippets as textual documents. Subsequently, supervised approaches using LSTM architectures showed improvements [5, 56] by also leveraging data augmentation strategies to transform code snippets while preserving their semantics [4]. Later with the advent of the transformer architecture in natural language processing [55], several works [13, 18, 23, 57, 59] employed transformer models for code retrieval tasks and reported significant gains in performance over previous approaches. Across a majority of these recent works, CodeSearchNet by Husain et al. [22] has emerged as a standard benchmark for calibrating code search performance. Researchers have attempted to modify the pre-training of transformer models for the code domain by embedding the structural information associated with programs in different forms. This has led to a string of code pre-trained models like CodeBERT [13] which introduced novel pre-training tasks for bimodal datasets containing text and code, GraphCodeBERT [18] which is pre-trained on code using tasks that embed structural information from the abstract syntax trees (ASTs) of code inputs and SynCoBERT [57], a syntax aware encoder architecture. In a related line of work, Lu et al. [37] propose a benchmark (NL-code-search-WebQuery) where natural language code search is framed as the problem of analysing a query-code pair to predict whether the code answers the query or not. More recently, Guo et al. [17] released UniXCoder. Unike CodeBERT's encoder-only pre-training (that uses the masked language modeling (MLM) and replaced token detection (RTD) objectives only), UniXCoder is pre-trained with a set of tasks like MLM, unidirectional language modeling, span denoising, cross-modal contrastive learning and cross-modal generation and has shown to be a competitive alternative for several code understanding and generation tasks.

In contrast to the research theme of finding optimal pre-training strategies for code, we focus on the adaptation or fine-tuning stages of pre-trained models. Similar to the pre-training stage, this fine-tuning stage also offers different training choices, which have been underexplored so far. One could adapt a pre-trained model in the

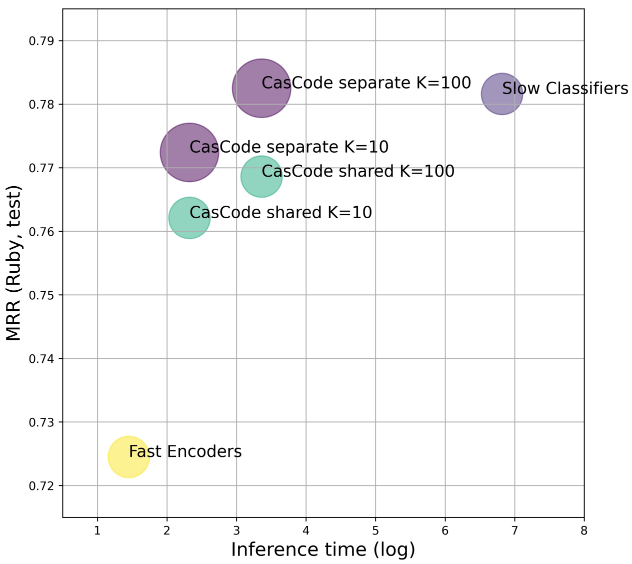

Figure 2: Overview of the speed versus performance (MRR metric 0-1: higher is better) trade-off of current code search approaches. With CasCode, we are able to achieve performance comparable to the optimal classifier based approach (top right), while requiring substantially lesser inference time. Areas of the circles here are proportional to model sizes. For reference Fast Encoders require 125M parameters.

fast encoder style or through the slow classifier style for retrieval. CasCode proposes to combine these two approaches to achieve optimal retrieval performance (speed and relevance) with any given pre-trained code understanding model.

3 CASCADING TRANSFORMER MODELS FOR TEXT-TO-CODE RETRIEVAL (CASCODE)

In this section, we describe our proposed CasCode approach including details of training and inference phases of the fast encoder stage (3.1), the slow classifier stage (3.2), our cascading scheme (3.3), and the shared variant of CasCode (3.4).

3.1 Stage I: Fast Encoders

For the first stage of *fast (bi-)* encoders, we use the contrastive learning framework [9], similar to the fine-tuning by Guo et al. [18], who leverage pairs of natural language and source code sequences to train text-to-code retrieval models. The representations of natural language (NL) and programming language (PL) sequences that match in semantics (a positive pair from the bimodal dataset) are pulled together, while representations of negative pairs (randomly paired NL and PL sequences) are pushed apart. The infoNCE loss (a form of contrastive loss function [19]) used for this approach can be defined as follows:

$$\mathcal{L}_{\text{infoNCE}} = \frac{1}{N}\sum_{i=1}^{N} -\log \frac{\exp(f_\theta(x_i)^T f_\theta(y_i)/\sigma)}{\sum_{j\in\mathcal{B}} \exp(f_\theta(x_i)^T f_\theta(y_j)/\sigma)} \qquad (1)$$

where $f_\theta(x_i)$ is the dense representation for the NL input x_i, and y_i is the corresponding semantically equivalent PL sequence. N is the number of training examples in the bimodal dataset, σ is a temperature hyper-parameter to control the sharpness of the model's output probability distribution, and $\mathcal{B}$ denotes the current training minibatch.

While the above approach applies for any model architecture, Guo et al. [18] employ GraphCodeBERT and CodeBERT for f_θ in their experiments. We refer to this approach as *fast* as it benefits from caching of candidate encodings before query time. During inference, we are given a set of candidate code snippets $C = \{y_1, y_2, \ldots y_{|C|}\}$, which are encoded offline into an index $\{f_\theta(y_j) \forall j \in C\}$. For a test NL query x_i, we then compute $f_\theta(x_i)$ and return the code snippet from C corresponding to the nearest neighbor (as per the cosine similarity distance metric) in the index. During inference, we are only required to perform the forward pass associated with $f_\theta(x_i)$ and the nearest neighbor lookup in the PL index, as the PL index itself can be constructed offline. This makes the approach very suitable for practical scenarios where the number of candidate code snippets $|C|$ could be very large.

Interestingly, a single encoder - either CodeBERT and GraphCodeBERT can be used to process the two modalities of text ($f_\theta(x_i)$) and code ($f_\theta(y_i)$). This could be attributed to the NL-PL pre-training of these models. Given this observation with the two code pre-trained models, in all our experiments we process the NL and PL inputs in the same manner, agnostic to their modality.

3.2 Stage II: Slow Classifiers

Although the above retrieval approach is efficient for practical scenarios, the independent encodings of the query and the code make it less effective as these do not allow for self-attention style interactions between NL and PL tokens. Similar to the monoBERT approach, we could instead encode the query and the code candidate jointly within a single transformer encoder and perform binary classification for ranking. In particular, the model could take as input the concatenation of NL and PL sequences $[x_i; y_j]$ and predict whether the two match in semantics.

The training batches for this binary classification setup can again be constructed using the bimodal dataset (positive pairs denoting semantic matches), and the negative pairs (mismatch) can be constructed artificially. Given a set of N paired NL-PL semantically equivalent sequences $\{x_i, y_i\}_{i=1}^{N}$, the cross-entropy objective function for this training scheme would be:

$$\mathcal{L}_{\text{CE}} = -\frac{1}{N}\sum_{i=1, j\neq i}^{N} \log p_\theta(x_i, y_i) + \log(1 - p_\theta(x_i, y_j)) \qquad (2)$$

where $p_\theta(x_i, y_j)$ represents the probability that the NL sequence x_i semantically matches the PL sequence y_j, as predicted by the classifier. With a minibatch $\mathcal{B}$ of positive pairs $\{x_i, y_i\}\ \forall i \in \mathcal{B}$, we can randomly pick y_j ($j \in \mathcal{B}; j \neq i$) from the PL sequences in the minibatch and pair it with x_i to serve as a negative pair. When using a transformer encoder based classifier, the interactions between the NL and PL tokens in the self-attention layers can help in improving the precision of this approach over the previous (independent encoding) one.

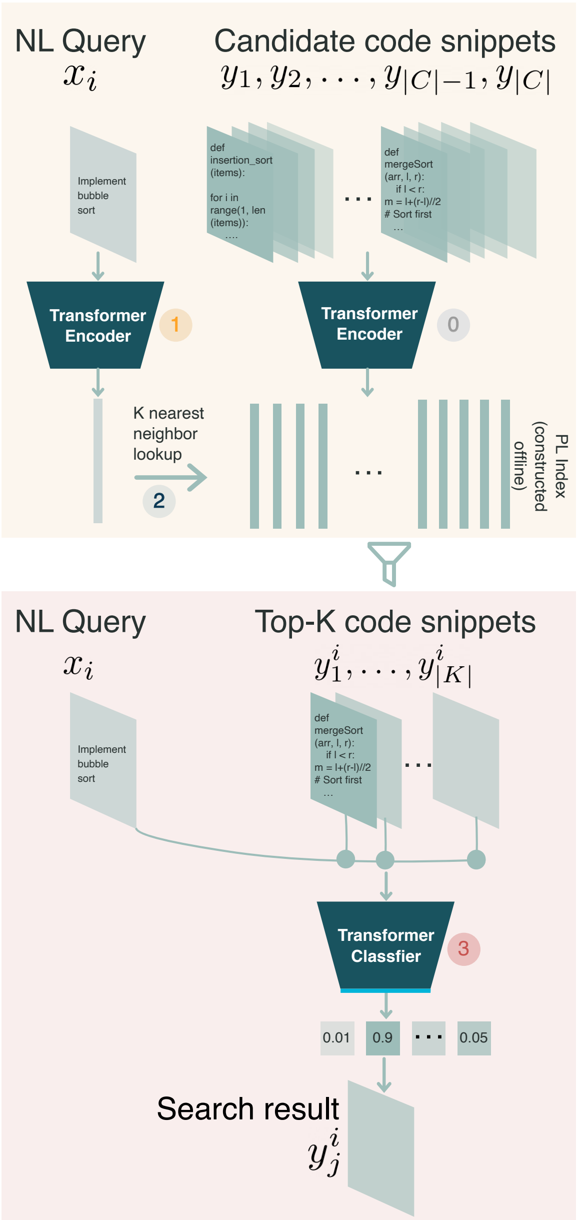

Figure 3: CasCode's Inference stage: The query x_i and the code snippets are first encoded independently by the transformer encoders. The top K candidates (based on nearest neighbor lookup) are then passed to the classifier which jointly processes the query with each of the filtered candidates to predict the probability of their semantics matching.

During inference, we can pair the NL sequence x_i with each of the y_j from C and rank the candidates as per the classifier's confidence scores of the pair being a match. This involves C forward passes (each on a joint NL-PL sequence, thus longer inputs than the previous approach), making this approach computationally infeasible when dealing with large retrieval sets. We refer to this approach as the one using *slow classifier* for retrieval.

3.3 Cascading of the two stages

With a cascaded scheme (that we call CasCode), we can unify the strengths of the two approaches - the speed of the *fast encoders* with the precision of the *slow classifiers*. To build CasCode, we first independently train the two stages discussed above - the fast encoder stage using the infoNCE objective and the slow classifier stage using the cross entropy loss. While these two approaches are alternatives to each other, we employ them in a sequential manner to perform retrieval. Figure 3 shows the overall framework of our approach. This hybrid strategy combines the strengths of the two approaches in the following manner - Given a query at test time (inference stage), the first stage of *fast encoders* provides a similarity score (based on the cosine distance between the query and candidate encodings) for each candidate from the set C of code snippets. In practice, the size of the retrieval set ($|C|$) can often be very large, and varies from 4360 to 52660 for the CodeSearchNet datasets we study in our experiments. The top K candidates based on the similarity scores from the first stage are then passed to the second stage of *slow classifiers* where each of them is paired with the NL input (query) x_i and fed to the model. For a given pair, this second stage classifier will return the probability of the NL and PL components of the input matching in semantics. Using these as confidence scores, the rankings of the K candidates are refined.

The resulting scheme is preferable for $K << |C|$, as this would add a minor computational overhead on top of what is required by the *fast encoder* based retrieval. The second stage of refinement can then improve retrieval performance provided that the value of K is set such that the recall of the *fast encoder* is reasonably high. K would be a critical hyper-parameter in this scheme, as setting a very low K would lead to high likelihood of missing the correct snippet in the set of inputs passed to the second stage *slow classifier*, while a very high K would make the scheme infeasible for retrieval. As we discuss later in Section 4, CasCode with a K as small as 10 already offers significant gains in retrieval performance over the baselines, with marginal gains as we increment K to 100 and beyond.

3.4 Making CasCode memory efficient

In order to minimize the memory overhead incurred by the two stage model, we propose to share the weights of the transformer layers of the *fast encoders* and the *slow classifiers*, by training a model with the joint (sum) objective $\mathcal{L}_{\text{shared}} = \mathcal{L}_{\text{infoNCE}} + \mathcal{L}_{\text{CE}}$. Thus, a single transformer model is trained to perform both encoding based retrieval and classification based retrieval in this multi-task learning setting. While the number of parameters in this shared variant would be nearly half of the separate (non-shared) case, the computational cost at inference would be the same. Note that we would need some exclusive parameters for the classifier model, specifically the classification head (a linear layer) on top of the encoder hidden states output. Thus, in this shared parameter variant of CasCode, the transformer model consuming the three kinds of inputs - NL only and PL only (for the *fast encoder* stage) and NL-PL (for the *slow classifier* stage) is identical except for the classification head in the second stage.

4 EXPERIMENTS

4.1 Setup and Fast retrieval baseline

We use the CodeSearchNet corpus from Husain et al. [22] that includes six programming languages - Ruby, Javascript, Go, Python, Java and Php. Our pre-processing and train-val-test splits are identical to the setting from Guo et al. [18], who filter low-quality queries and expand the retrieval set to make the code search task more challenging and realistic. Figure 2 shows 2 examples of bimodal pairs from the resulting dataset and the statistics of the dataset after pre-processing are provided in Table 1. Our primary evaluation metric is Mean Reciprocal Ranking (MRR), computed as $\frac{1}{N_{test}} \sum_{i=1}^{N_{test}} \frac{1}{r_i}$, where the r_i is the rank assigned to the correct code snippet (for the i-th query x_i) from the set of retrieval candidates C. We report MRR on the scale of 0-1, some works (eg. [57]) use the 0-100 scale.

Our *fast encoder* baseline is based on the CodeBERT model from Feng et al. [13] that is pre-trained on programming languages. In order to have a strong baseline, we use a newer CodeBERT checkpoint that we pre-train (using masked language modeling and replaced token detection tasks) for longer, after we found that the CodeBERT checkpoint from Feng et al. [13] was not trained till convergence. When starting from our new checkpoint, we find that the CodeBERT baseline, if fine-tuned with a larger batch-size (largest possible that we can fit on 8 A100 GPUs) and for a larger number of epochs, is able to perform substantially better than the results reported before. We report the baselines from Guo et al. [18] in Table 3 along with the results for our replication of two of these baselines. Previous studies have emphasized this effect - larger batch sizes are known to typically work well when training with the infoNCE loss in a contrastive learning framework, due to more negative samples from the batch [9].

We also finetune GraphCodeBERT [18] as a structure aware model pre-trained on programming languages. GraphCodeBERT leverages data flow graphs during pre-training to incorporate structural information into its representations. However, for the code search task, we report (Table 3) that GraphCodeBERT does not offer any significant improvements in performance over CodeBERT, when both variants are trained with a large batch size. As CodeBERT performs competitively and has a relatively simpler architecture (equivalent to RoBERTa-base[35] model with 12 layers, 768 dimensional hidden states and 12 attention heads), we chose it as the *fast encoder* baseline for the remainder of our experiments.

For finetuning on code search, we begin with the baseline implementation of GraphCodeBERT (https://github.com/microsoft/CodeBERT/tree/master/GraphCodeBERT) and adapt their setup to also implement the CodeBERT model. For the cascaded schemes, many of our training design decisions are therefore the same as GraphCodeBERT. We use 8 A100 GPUs (each with 40 GB RAM) to train our baselines and CasCode variants. During training, we set the batch-size to a value that occupies as much available GPU RAM as possible, which is 576 for the CodeBERT and GraphCodeBERT baseline finetuning with the infoNCE loss.

MRR scores on the test set for the CodeBERT baseline (*fast encoder*) along with several other baselines including sparse methods like BM25 (implemented using Pyserini [32]), fine-tuned CNN, BiRNN, multi-head attention models are shown in Table 3. Interestingly, BM25 outperforms all other methods on the Python dataset, this could be attributed to the simplicity of Python and its similarity with natural language [54]. For the CodeBERT baseline and the CasCode variants that we have proposed, along with MRR, we also report Recall@K for $K = \{1, 2, 5, 8, 10\}$, that indicates the hit rate (ratio of instances where we find the correct output in the top K results). We encourage future work on code search to report these additional metrics, as these are important in evaluating the utility of a retrieval system and are commonly reported in similar work in text retrieval and text based image or video retrieval [3, 38]. As alluded to in Section 3, for designing the cascaded scheme, we need to pick a K that is large enough to provide reasonably high recall, and small enough for the second stage to be reasonably fast. To guide our choice of K, we show in Figure 4 the Recall@K (K varied over the horizontal axis) for the 6 different programming languages, with the *fast encoder* models, over the validation set. For our experiments, we pick $K = 10$ and 100 where the recall for all 6 datasets is over 85% and 90%, respectively. Note that CasCode is a general framework and several different models can be employed in the two stages. We pick fine-tuned CodeBERT for the fast encoder phase of CasCode, as it is a simpler architecture than GraphCodeBERT or UniXCoder [17] and gives strong performance on its own when evaluated in the first stage only.

4.2 Results with CasCode

To build the model for the second phase of CasCode (separate) on top of the CodeBERT based (*fast*) encoders, we train the *slow* classifiers independently but evaluate them by cascading with the first phase. For this second stage model, we finetune the CodeBERT pre-trained checkpoint (detailed above) with a classification head on top (a linear layer on top of the hidden-states output) using the CodeSearchNet dataset. On the validation set, we study the performance of this finetuned classifier for retrieval and report the MRR scores in Figure 5 for different values of K, where K is the number of top candidates passed from the first (*fast encoder*) stage to the second. Interestingly, the retrieval performance of this joint classifier does not improve significantly beyond certain values of K. For example, increasing K from 10 to 100 only marginally improves the MRR for Ruby, Javascript and Java, while for other languages there is no significant improvement beyond $K = 10$. In CasCode's separate variant, we pair the fast encoder with this second stage classifier model and the MRR scores for this approach and the relevant baslines are provided in Table 3. With our cascaded approach, we observe significant improvements over the *fast encoder* baselines, the overall MRR averaged over the six programming languges for CasCode (separate) is 0.7795, whereas the *fast encoder* baseline (CodeBERT) reaches 0.7422. The improvements with CasCode are noticeably greater over the baseline for Ruby, Javascript, Python and Java. We report modest improvements on the Go dataset, where the *fast encoder* baseline is already quite strong (0.9145 MRR).

We also train *fast* and *slow* models with **shared parameters**, denoted by CasCode (shared). The training objective for this model is the sum of the binary cross-entropy loss $\mathcal{L}_{\text{CE}}$ and the infoNCE loss $\mathcal{L}_{\text{infoNCE}}$ as described in Section 3. The shared variant of CasCode attains an overall MRR score of 0.77, which is comparable to the separate variant. This slight difference can be attributed to the limited model capacity in the shared case, as the same set of transformer layers serve in the encoder and classifier models.

-	Ruby	Javascript	Go	Python	Java	PHP
Training examples	24,927	58,025	167,288	251,820	164,923	241,241
Dev queries	1,400	3,885	7,325	13,914	5,183	12,982
Testing queries	1,261	3,291	8,122	14,918	10,955	14,014
Candidate codes	4,360	13,981	28,120	43,827	40,347	52,660

Table 1: Data statistics of the filtered CodeSearchNet corpus for Go, Java, Javascript, PHP, Python and Ruby programming languages. For each query in the dev and test sets, the answer is retrieved from the set of candidate codes (last row).

Docstring: Prompt the user to continue or not
Code Snippet:

```
def continue_prompt(message = ""):
  answer = False
  message = message + """\n"Yes" or "No" to continue: """
  while answer not in ("Yes", "No"):
      answer = prompt ( message, eventloop = eventloop())
      if answer == "Yes":
          break
      if answer == "No":
          break
  return answer
```

Docstring: Sends a message to the framework scheduler.
Code Snippet:

```
def message(self, data):
  logging.info("""Driver sends framework
  message {}""".format(data))
  return self.driver.sendFrameworkMessage(data)
```

Table 2: Examples of bimodal pairs (natural language/docstring with corresponding code sequence) from CodeSearchNet (Python).

We also evaluate the MRR scores for the CasCode (shared) model when used in the *fast encoder* stage only, and the test set MRR scores were 0.7308, 0.6634, 0.9048, 0.7193, 0.7244, 0.6803 for Ruby, Javascript, Go, Python, Java and PHP respectively, with the overall MRR being 0.7372. Thus the cascaded model that was trained in a multi-task manner with a joint objective, gives competitive retrieval performance, even when used only in its first stage.

The improvements in the MRR scores of both CasCode variants - shared and separate over the CodeBERT fast encoder baseline are statistically significant for all 6 programming languages with $p < 0.0001$ as per the one-tailed student's t-test (recommended for retrieval by Urbano et al. [53]) for both $K = 10$ and $K = 100$. We also report the Recall@K metric for CasCode separate and shared variants in Figure 6. For all six programming languages, we observe improvements over the *fast encoder* baseline with our cascaded scheme. Similar to our observation from Table 3, the shared variant of CasCode is slightly worse than the separate one.

CasCode training details: For training the joint NL-PL classifier of CasCode (separate), we are able to use a batch size of 216. This batch-size is lower than the fast encoder finetuning batch-size because we are required to process joint NL-PL sequences ($f_\theta([x_i; y_i])$) which will be much longer in length than a NL only or PL only sequence. For CasCode's shared variant, we need to further reduce the training batch size to 160, as we are required to store activations from multiple forward passes for a given bimodal pair - NL only $f_\theta(x_i)$, PL only $f_\theta(y_i)$ and joint NL-PL $f_\theta([x_i; y_i])$. All models are trained for 100 epochs. For all our experiments we use a learning rate of 2e-5, and use the Adam optimizer [27] to update model parameters. For both the CasCode variants, when performing evaluation on the development set (for early stopping), we use $K = 100$ candidates from the fast encoder stage.

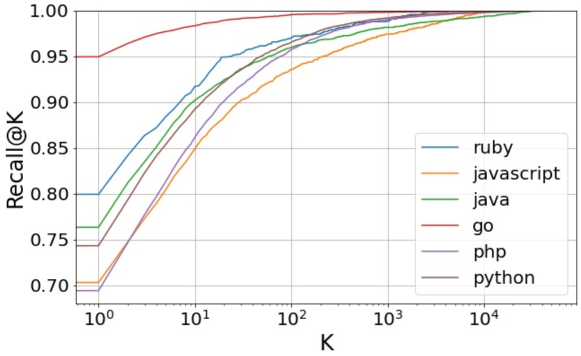

Figure 4: Recall at different values of K over the validation set of CodeSearchNet [22] when using a finetuned CodeBERT encoder (*fast*) for text-code retrieval.

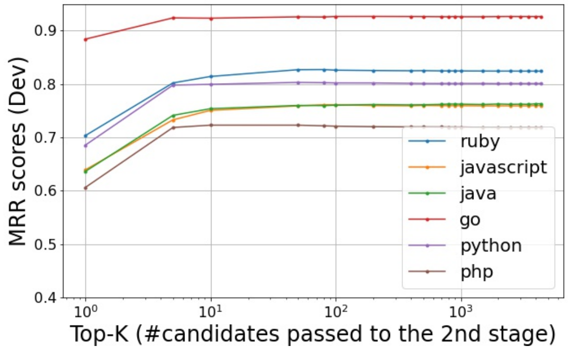

Figure 5: Mean reciprocal ranking (MRR) at different values of K over the validation set of CodeSearchNet [22] when using a finetuned CodeBERT (*slow*) binary classifier (match or not) for text-code retrieval. Note that with an increase in the number of top candidates passed to the second stage, the inference time would also increase, however we do not observe substantial gains in MRR beyond top-K of 10.

Model/Method	Ruby	Javascript	Go	Python	Java	Php	Overall
BM25	0.3859	0.3259	0.4978	**0.9454**	0.3272	0.3725	0.4758
As reported by Guo et al. [18]							
NBow	0.162	0.157	0.330	0.161	0.171	0.152	0.189
CNN	0.276	0.224	0.680	0.242	0.263	0.260	0.324
BiRNN	0.213	0.193	0.688	0.290	0.304	0.338	0.338
selfAtt	0.275	0.287	0.723	0.398	0.404	0.426	0.419
RoBERTa	0.587	0.517	0.850	0.587	0.599	0.560	0.617
RoBERTa (code)	0.628	0.562	0.859	0.610	0.620	0.579	0.643
CodeBERT	0.679	0.620	0.882	0.672	0.676	0.618	0.693
GraphCodeBERT	0.703	0.644	0.897	0.692	0.691	0.649	0.713
CodeT5-base [59]	-	-	-	-	-	-	0.715
UniXCoder [17]	-	-	-	-	-	-	0.744
As reported by Wang et al. [57]							
SynCoBERT	0.722	0.677	0.913	0.724	0.723	0.678	0.740
Replicated with a larger training batch-size							
CodeBERT	0.7245	0.6794	0.9145	0.7305	0.7317	0.681	0.7436
GraphCodeBERT	0.7253	0.6722	0.9157	0.7288	0.7275	0.6835	0.7422
Ours (K=10)							
CasCode (shared)	0.7621	0.6948	0.9193	0.7529	0.7528	0.7001	0.7637
CasCode (separate)	0.7724	0.7087	0.9258	0.7645	0.7623	0.7028	0.7727
Ours (K=100)							
CasCode (shared)	0.7686	0.6989	0.9232	0.7618	0.7602	0.7074	0.7700
CasCode (separate)	**0.7825**	**0.716**	**0.9272**	0.7704	**0.7723**	**0.7083**	**0.7795**

Table 3: Mean Reciprocal Ranking (MRR) scores of different methods on the codesearch task on 6 Programming Languages from the CodeSearchNet corpus (test set). The first row indicates performance with the BM25 scoring using bag-of-words (sparse) representations. The next set consists of four finetuning-based baseline methods (NBow: Bag of words, CNN: convolutional neural network, BiRNN: bidirectional recurrent neural network, and multi-head attention), followed by the second set of models that are pre-trained then finetuned for code search (RoBERTa: pre-trained on text by Liu et al. [35], RoBERTa (code): RoBERTa pre-trained only on code, CodeBERT: pre-trained on code-text pairs by Feng et al. [14], GraphCodeBERT: pre-trained using structure-aware tasks by Guo et al. [18]), CodeT5-base: Encoder-decoder transformer model by Wang et al. [59] pre-trained for code understanding and generation tasks, UniXCoder: unified cross-modal pre-trained model for code by Guo et al. [17]. SynCoBERT: pre-trained using syntax-aware tasks by Wang et al. [57]. CasCode (separate): Our cascaded retrieval scheme using two independent transformer encoder models, first in the *fast*/dual encoder stage and later in the slow classifier (monoBERT-style) stage. CasCode (ours) shared: a single encoder model is used in both the stages of CasCode using model parameter sharing. In the last four rows, we report the results with the shared and separate variants of our CasCode scheme.

When running inference on a single A100 GPU on a single query (batchsize of 1), the CodeBERT style fast encoder occupies 1482 MB of GPU RAM. This is also true for the slow binary classifiers (monoBERT style) and the shared CasCode variants. The inference stage memory requirement gets roughly doubled with CasCode's separate variant where there is no sharing of weights between the fast encoder and slow classifier stages. This is expected because the separate variant stores two different encoder models (identical architecture except the classification head), but different weights) on the GPU RAM.

4.3 Retrieval speed comparison

Having established the improvements in retrieval performance with CasCode, we proceed to analyze the trade-off between inference speed and performance, for the different methods discussed. For each variant, we record the time duration (averaged over 100 instances) required to process (obtain a relevant code snippet from the retrieval set) an NL query from the held-out set. We use the Ruby dataset of CodeSearchNet for this analysis, which contains 4360 candidate code snippets for each NL query. We conduct this study on a single Nvidia A100 GPU. Table 4 shows the results.

For the *fast encoder* approach (using infoNCE-finetuned CodeBERT), we first incur some computational cost to encode all the candidate code snippets and construct the PL index (6.76 seconds for Ruby's retrieval set). This computation is common to all approaches, except the *slow* (binary, joint) classifier one. Since this computation can be performed offline before the model is deployed to serve user queries, we do not include this cost in our results in Table 4. With the PL index constructed beforehand, we report

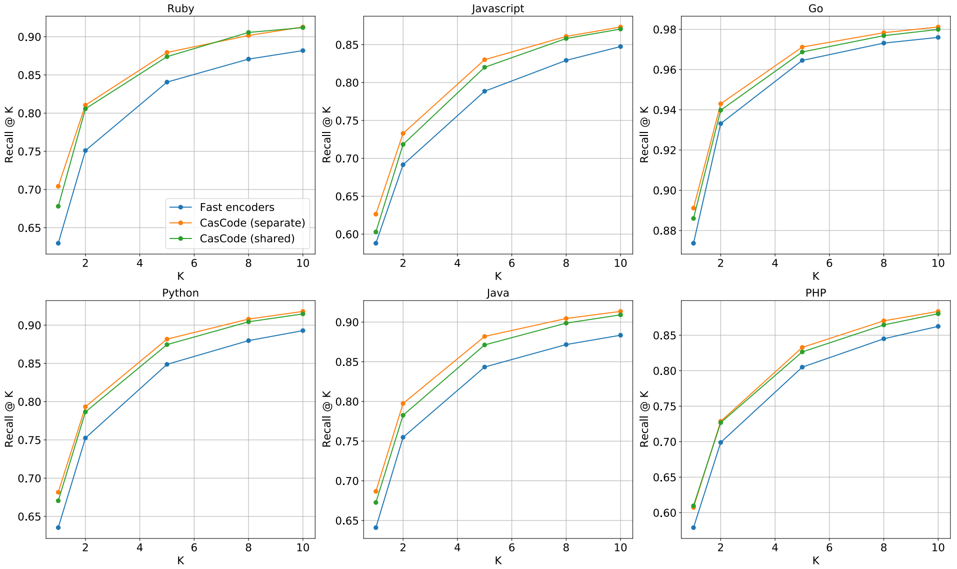

Figure 6: Recall @ K = $\{1, 2, 5, 8, 10\}$ with the *fast encoder* and CasCode (shared and separate) methods on the test set queries of CodeSearchNet dataset.

the time required to encode a user NL query, and perform nearest neighbor lookup on the PL index with the encoding, in the first row of Table 4. This computation is again performed by all the CasCode variants, and thus acts as the lower bound on time taken by CasCode for retrieval. For the analysis to be as close to real world scenarios as possible, we do not batch the queries (which can provide further speed-ups, specially on GPUs) and encode them one by one. Batching them would require assuming that we have the NL queries beforehand, while in practice we would be receiving them on the fly from users when deployed.

With the *slow classifier* approach, we would pair a given query with each of the 4360 candidates, and thus this would lead to the slowest inference of all the variants. For all variants of CasCode, the inference duration listed in Table 4 includes the time taken by the *fast encoder* based retrieval (first stage) along with the second stage. For CasCode's second stage, we can pass the K combinations (query concatenated with each of the top-K candidate from the fast stage) in a batched manner. The shared variant, while requiring half the parameters, incurs the same computational cost when used in the cascaded fashion. We note from Table 4 that at a minor drop in the MRR score, lowering CasCode's K from 100 to 10 can lead to almost 3x faster inference.

4.4 AdvTest (normalized) set evaluation

Previous works [13, 18] have employed a normalised variant of the CodeSearchNet Python dataset called AdvTest to evaluate text-to-code retrieval models. The function and variable names appearing in the code snippets in the test and development sets of this python dataset are normalized (*Func* for function names, *arg-i* for the i-th variable name). An example of this normalization is shown in Table 7. This dataset was processed and released by [37] to test the understanding abilities of code search systems as part of the CodeXGLUE benchmark. We follow their lead in evaluating our proposed CasCode on the AdvTest benchmark to study its retrieval effectiveness in this challenging setting. When experimenting with the AdvTest dataset, our focus is to is to compare the code understanding and retrieval abilities of different approaches in this more rigorous evaluation setting than the regular CodeSearchNet dataset, as the normalization scheme should prevent the models from over-relying on the natural language semantics (English components) of the candidate programs. For instance, the snippet `"def bubble_sort(): <python program here>"` in the regular setting would be easier to retrieve than the candidate `"def Func(): <python program here>"` in the normalized setting for the query `"Implement bubble sort"`. In the normalized setting, the model would have to rely on understanding the program semantics instead of variable or function names, which tend to be closer to natural language or plain English. Note that the AdvTest dataset is not adversarially constructed [41, 64] and it does not involve any gradient based methods (adversarial attacks) to perturb inputs beyond a simple normalisation function. We speculate that our models would inherit the same vulnerabilities to adversarial input perturbations as their NLP counterparts. Evaluating the robustness of code search models to such sophisticated adversarial attacks is beyond the current scope of our work.

The AdvTest dataset contains 251,820 training examples, 9,604 validation set examples and 19,210 test set examples. Each example is a bimodal pair of natural language docstrings and corresponding code snippets. During test time, all the 19,210 code snippets are treated as candidates for a given test query. The code retrieval results achieved by different approaches on this dataset are shown in Table 5. Results in the first two rows are reported from [37] where RoBERTa and CodeBERT are fine-tuned (batch size of 32) with the infoNCE loss discussed before in the fast encoder framework. In our re-implementation of the stronger baseline of CodeBERT, we increase the training batch-size to 512. This leads to an improved

Model	# params	Inference time (secs)	MRR	# queries/s
Fast encoders (CodeBERT style)	125M	0.0427	0.7245	23.42
Slow binary classifiers (monoBERT style)	125M + 0.5M	9.1486	0.7816	0.11
CasCode (separate, K=100)	250M + 0.5M	0.2883	0.7825	3.46
CasCode (shared, K=100)	125M + 0.5M	0.2956	0.7686	3.38
CasCode (separate, K=10)	250M + 0.5M	0.1022	0.7724	9.78
CasCode (shared, K=10)	125M + 0.5M	0.1307	0.7621	7.65

Table 4: Inference speed comparison for different variants of the proposed methods. The number of parameters corresponding to the classifier head are separated with a ‘ + ’ sign in the second column. Inference duration is averaged for 100 queries from the Ruby subset of CodeSearchNet, using a single A100 GPU. Constructing the PL index offline requires 6.76 seconds for the Ruby dataset and is not included in the durations listed here. MRR scores are reported on the entire test set. Throughput of the retrieval model (measured in # queries processed per second) is listed in the last column.

test MRR score of 0.3381. For CasCode's separate variant, we finetune the slow classifier stage with the binary cross entropy loss. This second stage model is initialized from the CodeBERT pre-trained checkpoint and trained on the 251,820 pairs. For each positive pair, we can create a synthetic negative one by pairing a docstring with a random code snippet. For CasCode's shared variant, we train the two stages jointly by tying the weights of the two encoder similar to previous experiments from Section 4.2. The finetuning loss is the sum of the infoNCE loss and binary CE loss computed using the same minibatch.

From Table 5, we see that when using CasCode with $K = 10$ candidates, we observe a substantial imrpovement, the shared variant scores MRR of 0.4005, and the separate one 0.3972. Retrieval performance can be further improved to MRR score of 0.4299 with the shared variant and 0.4398 with the separate, if we increase the number of candidates K to 100. In Figure 8, we show an example from this test set, where, for a given query, the first stage of fast encoder (equivalent to the re-implemented CodeBERT baseline) assigns a rank r_i of 3 to the matching code snippet, and then the slow classifier refines the ranking to 1. In cases when CasCode fails at retrieving the correct code snippet as the top search result, our qualitative analysis suggests that the resulting code snippet is often closely related in semantics and functionality. We present one such case in Figure 9 where the test set query asks for a function returning an array of certain attributes and the CasCode-retrieved code snippet performs a similar operation. The gap in performance for deep learning models between the original unaltered CodeSearchNet test set and the AdvTest one is nonetheless still an open problem that suggests our current models over-rely on the function and variable naming (as done by human programmers) and less on the inherent structure of the code in representing source code. Table 8 lists that UniXCoder [17] and CodeT5-base [59], when used in a single stage (fast encoders), perform competitively on the AdvTest benchmark. UniXCoder's performance with an MRR of 0.413 is significantly better than CodeBERT's MRR of 0.3381, but worse than CasCode's MRR of 0.4398. We expect additional improvements to CasCode's performance on AdvTest by fine-tuning a model like UniXCoder in its two stages. Subsequent to our work, several submissions to the CodexGLUE AdvTest leaderboard seem to have made this improvement. To the best of our knowledge, the details of these approaches have not yet been released, preventing any further analysis or comparison.

Code Snippet:

```
def day_start_ut(self, ut):
# set timezone to the one of gtfs
    old_tz = self.set_current_process_time_zone()
    ut = time.mktime(time.localtime(ut)[:3]
         + (12, 00, 0, 0, 0, -1)) - 43200
    set_process_timezone(old_tz)
    return ut
```

Normalized code snippet:

```
def Func(arg_0, arg_1):
    arg_2 = arg_0.set_current_process_time_zone()
    arg_1 = time.mktime(time.localtime(arg_1)[:3]
          + (12, 00, 0, 0, 0, -1)) - 43200
    set_process_timezone(arg_2)
    return arg_1
```

Figure 7: An example of the normalization performed for constructing the AdvTest dataset. Lu et al. [37] designed the normalization to curate a challenging test set for text based code retrieval that can assess the understanding and generalization abilities of models.

Model/Method	**Test MRR**
RoBERTa	0.1833
CodeBERT (original implementation)	0.2719
CodeBERT (our re-implemention w/ a larger bsz)	0.3381
CodeT5-base [59]	0.393
UniXCoder [17]	0.413
CasCode (shared, K=10)	0.4005
CasCode (separate, K=10)	0.3972
CasCode (shared, K=100)	0.4299
CasCode (separate, K=100)	0.4398

Table 5: Results on the AdvTest set [37] of CodeSearchNet.

5 CONCLUSION AND FUTURE WORK

We propose CasCode, a cascaded text-to-code retrieval scheme consisting of transformer encoder and joint binary classifier stages, which achieves state of the art performance on the CodeSearchNet benchmark with significant improvements over previous results.

Input NL Query: Creates a base Django project

Correct code snippet (retrieved by CasCode's second stage):

```
def Func(arg_0):
    if os.path.exists(arg_0._py):
        arg_1 = os.path.join(arg_0._app_dir, arg_0._project_name)
        if os.path.exists(arg_1):
            if arg_0._force:
                logging.warn('Removing existing project')
                shutil.rmtree(arg_1)
            else:
                logging.warn('Found existing project;
                not creating (use --force to overwrite)')
                return
        logging.info('Creating project')
        arg_2 = subprocess.Popen('cd {0} ; {1} startproject {2}
                                 > /dev/null'.format(
                                 arg_0._app_dir,
                                 arg_0._ve_dir + os.sep + \
                                 arg_0._project_name + os.sep + \
                                 'bin' + os.sep + 'django-admin.py',
                                 arg_0._project_name),
                                 shell=True)
        os.waitpid(arg_2.pid, 0)
    else:
        logging.error('Unable to find Python interpreter
                  in virtualenv')
        return
```

Top code snippet retrieved by CasCode's first stage:

```
def Func():
    arg_0 = Bunch(DEFAULTS)

    arg_0.project_root = get_project_root()
    if not arg_0.project_root:
        raise RuntimeError("No tasks module is imported,
        cannot determine project root")

    # this assumes an importable setup.py
    if arg_0.project_root not in sys.path:
        sys.path.append(arg_0.project_root)
    try:
        from setup import arg_6
    except ImportError:
        from setup import setup_args as arg_6
    arg_0.project = Bunch(arg_6)

    return arg_0
```

Figure 8: An example from the test set of the AdvTest (normalized variant) CodeSearchNet (Py) dataset with retrieved queries from CasCode's two stages. For the NL query "Creates a base Django project", CasCode correctly retrieves the corresponding code snippet as the top result. The fast encoder baseline (first stage of CasCode) presents this snippet as the 3rd result, this is then re-ranked to the top by CasCode's second stage.

We also propose a shared parameter variant of CasCode, where a single transformer encoder can operate in the two different stages when trained in a multi-task fashion. With almost half of the parameter size and memory cost, CasCode's shared variant offers comparable performance to the non-shared (separate) variant.

Despite showing promising results, there are still some areas for improving our method. One limitation of our current cascaded scheme is that the computation spent in generating representations in the first stage of *fast encoders* is not fully leveraged in the second stage. Currently, we process raw token level inputs in the second stage, but ideally the representations from the first stage should be useful for the classification stage too [30]. Our initial attempts along this direction did not turn fruitful, and future work could address this aspect. To improve the inference speed of the two-stage retrieval, future work could explore methods like quantization and model distillation of the transformer models (e.g., employing the ONNX runtime [63]). Another limitation warranting further investigation is associated with the training of the shared variant of CasCode. Here, training with the multitask learning framework (joint objective of infoNCE and binary cross entropy) leads to a model that performs slightly worse than the separate variant (individually finetuned models). We tried augmenting the capabilities of this model with solutions like using independent CLS tokens for the three modes (the model has to operate in NL only, PL only, NL-PL concatenation), and adjusting the relative weight of the two losses involved but failed to obtain any improvement over the separate variant. Lastly, similar to related work in NLP [7], designing innovative pre-training schemes to specifically improve code search performance is also a promising direction for future work.

Input NL Query: Get the thing's actions as an array. action_name - Optional action name to get descriptions for. Returns the action descriptions.

Correct code snippet:

```
def Func(arg_0, arg_1=None):
    arg_2 = []

    if arg_1 is None:
        for arg_3 in arg_0.actions:
            for arg_4 in arg_0.actions[arg_3]:
                arg_2.append(arg_4.as_action_description())
    elif arg_1 in arg_0.actions:
        for arg_4 in arg_0.actions[arg_1]:
            arg_2.append(arg_4.as_action_description())

    return arg_2
```

Top code snippet retrieved by CasCode:

```
def Func(arg_0):
    arg_1 = arg_0.get_data("droplets/%s/actions/" % arg_0.id,
    type=GET)

    arg_2 = []
    for arg_3 in arg_1['actions']:
        arg_4 = Action(**arg_3)
        arg_4.token = arg_0.token
        arg_4.droplet_id = arg_0.id
        arg_4.load()
        arg_2.append(arg_4)
    return arg_2
```

Figure 9: An example from the test set of the AdvTest (normalized variant) CodeSearchNet (Py) dataset where CasCode doesn't retrieve the matching code snippet. The correct code snippet appears 12th in fast encoder's search results and is jumped to the 3rd position in the second stage reranking.

6 DATA AVAILABILITY

We provide our implementation (source code and pointers to datasets) as supplementary material to replicate the experiments (also available at this figshare URL: https://figshare.com/s/c94bf9c7a5aef4222449), which will be open sourced upon acceptance to support further research.

REFERENCES

[1] 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*. Association for Computational Linguistics, Online, 2655–2668. https://doi.org/10.18653/v1/2021.naacl-main.211

[2] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program Synthesis with Large Language Models. *arXiv preprint arXiv:2108.07732* (2021).

[3] Max Bain, Arsha Nagrani, Gül Varol, and Andrew Zisserman. 2021. Frozen in Time: A Joint Video and Image Encoder for End-to-End Retrieval. In *IEEE International Conference on Computer Vision*.

[4] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2021. Self-Supervised Contrastive Learning for Code Retrieval and Summarization via Semantic-Preserving Transformations. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 511–521.

[5] Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When deep learning met code search. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 964–974.

[6] Rich Caruana. 1997. Multitask learning. *Machine learning* 28 (1997), 41–75.

[7] Wei-Cheng Chang, Felix X. Yu, Yin-Wen Chang, Yiming Yang, and Sanjiv Kumar. 2020. Pre-training Tasks for Embedding-based Large-scale Retrieval. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=rkg-mA4FDr

[8] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harri Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[9] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. 2020. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*. PMLR, 1597–1607.

[10] Ernest Davis. 2022. A short comment on AlphaCode. https://cs.nyu.edu/~davise/papers/AlphaCode.html

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[12] Zi-Yi Dou, Aishwarya Kamath, Zhe Gan, Pengchuan Zhang, Jianfeng Wang, Linjie Li, Zicheng Liu, Ce Liu, Yann LeCun, Nanyun Peng, Jianfeng Gao, and Lijuan Wang. 2022. Coarse-to-Fine Vision-Language Pre-training with Fusion in the Backbone. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., 32942–32956. https://proceedings.neurips.cc/paper_files/paper/2022/file/d4b6ccf3acd6ccbc1093e093df345ba2-Paper-Conference.pdf

[13] 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*. Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[14] 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*. Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[15] Daniel Gillick, Sayali Kulkarni, Larry Lansing, Alessandro Presta, Jason Baldridge, Eugene Ie, and Diego Garcia-Olano. 2019. Learning Dense Representations for Entity Retrieval. In *Proceedings of the 23rd Conference on Computational Natural Language Learning (CoNLL)*. Association for Computational Linguistics, Hong Kong, China, 528–537. https://doi.org/10.18653/v1/K19-1049

[16] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 933–944.

[17] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Dublin, Ireland, 7212–7225. https://doi.org/10.18653/v1/2022.acl-long.499

[18] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2021. Graphcodebert: Pre-training code representations with data flow. *ICLR 2021* (2021).

[19] Michael Gutmann and Aapo Hyvärinen. 2010. Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, 297–304.

[20] Junjie Huang, Duyu Tang, Linjun Shou, Ming Gong, Ke Xu, Daxin Jiang, Ming Zhou, and Nan Duan. 2021. CoSQA: 20,000+ Web Queries for Code Search and Question Answering. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Association for Computational Linguistics, Online, 5690–5700. https://doi.org/10.18653/v1/2021.acl-long.442

[21] Jui-Ting Huang, Ashish Sharma, Shuying Sun, Li Xia, David Zhang, Philip Pronin, Janani Padmanabhan, Giuseppe Ottaviano, and Linjun Yang. 2020. Embedding-based Retrieval in Facebook Search. *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (2020).

[22] 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* (2019).

[23] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2019. Learning and Evaluating Contextual Embedding of Source Code. In *International Conference on Machine Learning*.

[24] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense Passage Retrieval for Open-Domain Question Answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Online, 6769–6781. https://doi.org/10.18653/v1/2020.emnlp-main.550

[25] Omar Khattab and Matei Zaharia. 2020. Colbert: Efficient and effective passage search via contextualized late interaction over bert. In *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval*. 39–48.

[26] Heidy Khlaaf, Pamela Mishkin, Joshua Achiam, Gretchen Krueger, and Miles Brundage. 2022. A Hazard Analysis Framework for Code Synthesis Large Language Models. *ArXiv* abs/2207.14157 (2022).

[27] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1412.6980

[28] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems* 35 (2022), 21314–21328.

[29] Junnan Li, Dongxu Li, Caiming Xiong, and Steven C. H. Hoi. 2022. BLIP: Bootstrapping Language-Image Pre-training for Unified Vision-Language Understanding and Generation. In *International Conference on Machine Learning*.

[30] Junnan Li, Ramprasaath R. Selvaraju, Akhilesh Deepak Gotmare, Shafiq Joty, Caiming Xiong, and Steven Hoi. 2021. Align before Fuse: Vision and Language Representation Learning with Momentum Distillation. In *NeurIPS*.

[31] Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom, Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de, Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey, Cherepanov, James Molloy, Daniel Jaymin Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de, Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with AlphaCode. *Science* 378 (2022), 1092 – 1097.

[32] Jimmy Lin, Xueguang Ma, Sheng-Chieh Lin, Jheng-Hong Yang, Ronak Pradeep, and Rodrigo Nogueira. 2021. Pyserini: A Python Toolkit for Reproducible Information Retrieval Research with Sparse and Dense Representations. In *Proceedings of the 44th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2021)*. 2356–2362.

[33] Jimmy Lin, Rodrigo Nogueira, and Andrew Yates. 2021. Pretrained transformers for text ranking: Bert and beyond. *Synthesis Lectures on Human Language Technologies* 14, 4 (2021), 1–325.

[34] Jiawei Liu, Chun Xia, Yuyao Wang, and Lingming Zhang. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. *ArXiv* abs/2305.01210 (2023).

[35] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *ArXiv* abs/1907.11692 (2019).

[36] Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. ReACC: A Retrieval-Augmented Code Completion Framework. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Dublin, Ireland, 6227–6240. https://doi.org/10.18653/v1/2022.acl-long.431

[37] 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 (2021).

[38] Antoine Miech, Jean-Baptiste Alayrac, Ivan Laptev, Josef Sivic, and Andrew Zisserman. 2021. Thinking Fast and Slow: Efficient Text-to-Visual Retrieval with Transformers. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 9826–9836.

[39] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems*, C.J. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.Q. Weinberger (Eds.), Vol. 26. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2013/file/9aa42b31882ec039965f3c4923ce901b-Paper.pdf

[40] Bhaskar Mitra, Eric Nalisnick, Nick Craswell, and Rich Caruana. 2016. A Dual Embedding Space Model for Document Ranking. https://www.microsoft.com/en-us/research/publication/a-dual-embedding-space-model-for-document-ranking/ This paper is an extended evaluation and analysis of the model proposed in a poster to appear in WWW'16, April 11 - 15, 2016, Montreal, Canada..

[41] John X. Morris, Eli Lifland, Jin Yong Yoo, Jake Grigsby, Di Jin, and Yanjun Qi. 2020. TextAttack: A Framework for Adversarial Attacks, Data Augmentation, and Adversarial Training in NLP. In *Conference on Empirical Methods in Natural Language Processing*.

[42] Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. Retrieval-Based Prompt Selection for Code-Related Few-Shot Learning. In 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE).

[43] Arvind Neelakantan, Tao Xu, Raul Puri, Alec Radford, Jesse Michael Han, Jerry Tworek, Qiming Yuan, Nikolas Tezak, Jong Wook Kim, Chris Hallacy, Johannes Heidecke, Pranav Shyam, Boris Power, Tyna Eloundou Nekoul, Girish Sastry, Gretchen Krueger, David Schnurr, Felipe Petroski Such, Kenny Hsu, Madeleine Thompson, Tabarak Khan, Toki Sherbakov, Joanne Jang, Peter Welinder, and Lilian Weng. 2022. Text and Code Embeddings by Contrastive Pre-Training. arXiv:2201.10005 [cs.CL]

[44] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. *ICLR* (2023).

[45] Rodrigo Nogueira, Wei Yang, Kyunghyun Cho, and Jimmy J. Lin. 2019. Multi-Stage Document Ranking with BERT. *ArXiv* abs/1910.14424 (2019).

[46] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., 27730–27744. https://proceedings.neurips.cc/paper_files/paper/2022/file/b1efde53be364a73914f58805a001731-Paper-Conference.pdf

[47] Md Rizwan Parvez, Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval Augmented Code Generation and Summarization. In *Findings of the Association for Computational Linguistics: EMNLP 2021*. Association for Computational Linguistics, Punta Cana, Dominican Republic, 2719–2734. https://doi.org/10.18653/v1/2021.findings-emnlp.232

[48] Hammond A. Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2021. An Empirical Cybersecurity Evaluation of GitHub Copilot's Code Contributions. *ArXiv* abs/2108.09293 (2021).

[49] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global Vectors for Word Representation. In *Conference on Empirical Methods in Natural Language Processing*.

[50] Sebastian Ruder. 2017. An overview of multi-task learning in deep neural networks. *arXiv preprint arXiv:1706.05098* (2017).

[51] Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. 2018. Retrieval on source code: a neural code search. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 31–41.

[52] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1433–1443.

[53] Julián Urbano, Harlley Lima, and Alan Hanjalic. 2019. Statistical Significance Testing in Information Retrieval: An Empirical Analysis of Type I, Type II and Type III Errors. In *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval* (Paris, France) *(SIGIR'19)*. Association for Computing Machinery, New York, NY, USA, 505–514. https://doi.org/10.1145/3331184.3331259

[54] Guido van Rossum. 1997. Comparing Python to Other Languages. https://www.python.org/doc/essays/comparisons/

[55] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 5998–6008. https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[56] Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip Yu. 2019. Multi-modal attention network learning for semantic source code retrieval. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 13–25.

[57] Xin Wang, Yasheng Wang, Fei Mi, Pingyi Zhou, Yao Wan, Xiao Liu, Li Li, Hao Wu, Jin Liu, and Xin Jiang. 2021. SynCoBERT: Syntax-Guided Multi-Modal Contrastive Pre-Training for Code Representation. *arXiv preprint arXiv:2108.04556*.

[58] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922* (2023).

[59] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021* (2021).

[60] Lee Xiong, Chenyan Xiong, Ye Li, Kwok-Fung Tang, Jialin Liu, Paul Bennett, Junaid Ahmed, and Arnold Overwijk. 2021. Approximate nearest neighbor negative contrastive learning for dense text retrieval. *ICLR 2021* (2021).

[61] 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 (TOSEM)* 31 (2021), 1 – 47.

[62] Xin Ye, Hui Shen, Xiao Ma, Razvan Bunescu, and Chang Liu. 2016. From word embeddings to document similarities for improved information retrieval in software engineering. In *Proceedings of the 38th international conference on software engineering*. 404–415.

[63] Minjia Zhang, Samyam Rajbandari, Wenhan Wang, Elton Zheng, Olatunji Ruwase, Jeff Rasley, Jason Li, Junhua Wang, and Yuxiong He. 2019. Accelerating Large Scale Deep Learning Inference through {DeepCPU} at Microsoft. In *2019 USENIX Conference on Operational Machine Learning (OpML 19)*. 5–7.

[64] W. Zhang, Quan Z. Sheng, Ahoud Abdulrahmn F. Alhazmi, and Chenliang Li. 2019. Adversarial Attacks on Deep-learning Models in Natural Language Processing. *ACM Transactions on Intelligent Systems and Technology (TIST)* 11 (2019), 1 – 41.

[65] Shuyan Zhou, Uri Alon, Frank F. Xu, Zhiruo Wang, Zhengbao Jiang, and Graham Neubig. 2023. DocPrompting: Generating Code by Retrieving the Docs. In *International Conference on Learning Representations (ICLR)*. Kigali, Rwanda. https://arxiv.org/abs/2207.05987