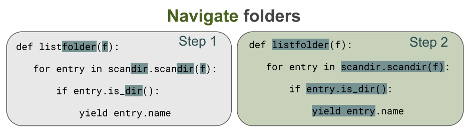
NS^3: NEURO-SYMBOLIC SEMANTIC CODE SEARCH

Anonymous authors
Paper under double-blind review

ABSTRACT

Semantic code search is the task of retrieving a code snippet given a textual description of its functionality. Recent work has been focused on using similarity metrics between neural embeddings of text and code. However, current language models are known to struggle with longer, compositional sentences, and multi-step reasoning. To overcome this limitation, we propose supplementing the query sentence with a layout of its semantic structure. The semantic layout is used to break down the final reasoning decision into a series of lower-level decisions. We use a Neural Module Network architecture to implement this idea. We compare our model - NS^3 (Neuro-Symbolic Semantic Search) - to a number of baselines, including state-of-the-art semantic code retrieval methods, such as CodeBERT, CuBERT and GraphCodeBERT, and evaluate on two datasets - Code Search Net (CSN) and Code Search and Question Answering (CoSQA). On these datasets we demonstrate that our approach results in higher performance. We also perform additional studies to show the effectiveness of our modular design when handling compositional queries.

1 INTRODUCTION



```
def listfolder(f):
    for entry in scandir.scandir(f):
        if entry.is_dir():
            yield entry.name
```

Figure 1: To match query "Navigate folders" on a code snippet, we would start by finding all references to folders, paths or directories, using semantic, syntactic, or linguistic cues. Afterwards, we would look for ques that discovered entities are being iterated through.

With the increasing scale of software repositories, searching for relevant code fragments in large code bases becomes more challenging. Traditionally, searching source code has been limited to keyword search (Reiss, 2009; Lu et al., 2015) or regex search (Bull et al., 2002). This requires the user to know exactly what keywords appear in or around the code they hope to retrieve. With the rise of neural models, the task of *semantic code search (SCS)* – or retrieving code from a textual description of its functionality – is being studied from new perspectives. Most approaches map a database of code snippets and natural language queries to a high-dimensional space with a neural model (e.g., a Transformer (Vaswani et al., 2017)). Relevant code snippets are retrieved by performing search over this embedding space using either predefined similarity metric or a learned distance function (Kanade et al., 2020; Feng et al., 2020; Du et al., 2021). Some of the recent works capitalize on the rich structure of the code, and apply various graph neural networks for obtaining representations (Guo et al., 2021; Liu et al., 2021).

Despite the impressive results, current neural code search approaches are far from satisfactory in dealing with a wide range of natural-language queries. First of all, encoding longer text into a dense vector is an open problem for neural language models, as neural networks are not believed to be extracting systematic rules from data (Beltagy et al., 2020; Bhathena et al., 2020; Baroni, 2019). Not only this potentially affects the performance, but it can drastically reduce model's value for the users, because compositional queries such as "*Checking that directory does not exist before creating it*" may require explicitly capturing compositionality and performing multi-step reasoning on code.

We suggest overcoming these limitations by introducing modular workflow based on the semantic structure of the query. Figure 1 demonstrates an example, of how an engineer would approach the task of searching for code that navigates through folders in Python. They would first only pay attention to code that has cues about operating with paths, directories or folders. Afterwards, they would seek indications of iterating through some of the found objects or other entities in code related to them. In other words, they would perform multiple steps of different nature - i.e. finding indications of specific types of data entities, or specific operations. We attempt to imitate this process in this work.

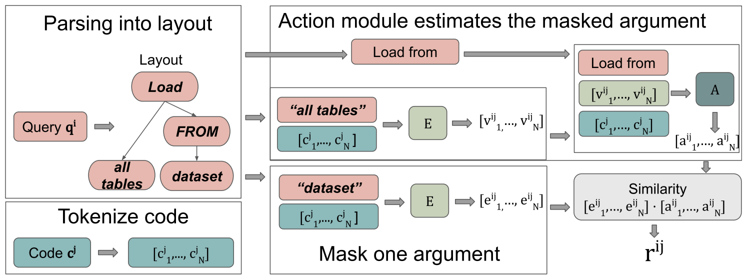


Figure 2: The pipeline of processing for an example query "Load all tables from dataset".

To formalize the decomposition of the query into such steps, we take inspiration from the idea that code is comprised of data entities and transformations over data. Thus, a semantic code search query is also likely to describe the code in terms of data entities and actions. We break down the task of matching the query into smaller tasks of matching individual data entities and actions. For that we try to identify parts of the code that evidence the presence of the corresponding data or action. We tackle each with a distinct type of network, or neural module. Using the semantic parse of the query, we construct the layout of how modules' outputs should be linked according to the relationships between data entities and actions. Correspondingly, this layout specifies how the modules should be combined into a single neural module network(Andreas et al., 2016). This network is "evaluated" on the candidate code snippet to decide whether the query and code are related.

We argue that this approach has the following advantages. Firstly, semantic parse captures the compositionality of a query. Secondly, each of the modules handles only a small portion of the query. This way it is more likely to be represented and detected with higher precision, in other words, less likely to require compositional reasoning from an individual neural module. Finally, applying the neural modules in a succession can potentially mimic staged reasoning necessary for SCS. Figure 2 provides a high-level overview of our entire pipeline.

We evaluate our proposed NS^3 model on two SCS datasets - CodeSearchNet (Husain et al., 2019) and CoSQA/WebQueryTest (Huang et al., 2021). WE also perform additional experimentation after limiting the training set to just 10K or 5K examples. We find that NS^3 provides large improvements upon baselines in all cases. To measure how well the model captures compositional properties of the queries, we experiment with performing adversarial modifications to the query and showing that compared to the baselines, NS^3 is more likely to correctly recognize modified queries. We also research the effect on queries that require multi-staged reasoning over code by breaking down the performance by query's compositional complexity, and demonstrate that our proposed method handles this task better than the baselines.

Our main contributions are: (i) We propose looking at SCS as a compositional task that requires multi-staged reasoning. (ii) We present an implementation of the aforementioned paradigm based on neural module networks. (iii) We evaluate the performance of our proposed model and demonstrate that it provides a large improvement on a number of well-established baseline models. (iv) We perform additional studies to evaluate the capacity of our model to handle compositional queries.

2 BACKGROUND

2.1 SEMANTIC CODE SEARCH (SCS)

Semantic code search is the process of retrieving a relevant code snippet based on a textual description of its functionality, also referred to as *query*. Formally, let $\mathcal{C}$ be a database of code snippets c^i. For each code snippet c^i, there is a textual description of its functionality q^i. In the example in Figure 2 the query q^i is "Load all tables from dataset". Let r be an indicator function such that $r(q^i, c^j) = 1$ if $i = j$ and 0 otherwise. Given some query q^* the goal of the SCS is to find c^* such that $r(q^*, c^*) = 1$. We assume that for each q^* there is exactly one such c^*. In our formulation, the models take as input a pair of query and code: (q^i, c^j) and assign it a probability r^{ij} for being a correct match. Following the common practice in information retrieval, we evaluate the performance of the model based on how high the correct answer c^* is rated among a number of incorrect, or distractor, instances c^j.

2.2 Neural Models for SCS

Most of the previous work has focused on enriching their models with incorporating more semantic and syntactic information from code. More formally, prior works typically either define a neural network f that takes as input a pair (q^i, c^j) and outputs the score r^{ij}: $r^(ij) = f(q^i, c^j)$, or define a number of separate networks for independently representing the query (f), the code (g) and measuring the distance between them (d): $r^{ij} = d(f(q^i), g(c^j))$.

Limitations However, none of the existing approaches has attempted using semantic structure of the *query* to improve their model. We see two main limitations with that.The first one is the fact that they use a single representation for the entire query. This means that details of the query or its compositional properties can get lost in the representation. Another limitation is the fact, that all these approaches make the decision after a single pass over the entire code snippet. This ignores cases where reasoning about a query requires multiple steps, and thus multiple lookups in the code. We address both of these limitations in our proposed approach - NS^3.

3 Neural Modular Code Search

We propose to supplement the query with a loose structure resembling its semantic parse, as illustrated in Figure 2. Based on this semantic structure, we create a query-specific layout of the neural module network (Sec. 3.1). We define two types of neural modules - entity discovery (E) and action module (A) (Sec. 3.2 and 3.3). The entity discovery module estimates how relevant each of code tokens $[c^j_1, \ldots, c^j_N]$ to an entity from the query, e.g. "all tables" or "dataset" in the example in Figure 2. The action module's goal is estimating whether there is evidence of the given action happening in the code snippet. The modules are nested - some are taking as input part of the output of another module - and the order of nesting is decided by the semantic parse layout. Evaluating the resulting neural network on a code snippet gives us a prediction about how well the code matches the original query (Sec. 3.3).

Specifically, each instance in the training set is a 4-tuple $(q^i, s_{q^i}, c^j, r(q^i, c^j))$ which consists of a natural language query q^i, the query's semantic parse s_{q^i}), a candidate code (sequence) c^j, and a binary label $r(q^i, c^j)$ indicating whether the query q^i matches the code — $r(q^i, c^j) = 1$ if the code matches for the query, and $r(q^i, c^j) = 0$ otherwise. The layout $L(s_{q^i})$ of the network is created from the semantic structure of the query s_{q^i}. Given a training example $(q^i, s_{q^i}, c^j, r(q^i, c^j))$ the model instantiates a network based on the layout, passes q^i, c^j and s_{q^i} as inputs, and obtains the model prediction r^{ij}. This pipeline is illustrated in Figure 2.

3.1 Parsing for Module Network Layout

We use a Combinatory Categorial Grammar-based (CCG) semantic parser (Zettlemoyer & Collins, 2012; Artzi et al., 2015) to infer the structural representation s_{q^i} (i.e., semantic parse) for a given natural language query q^i. We look to pair the query, *e.g.*, "*Load all tables from dataset*" (as in Fig. 2), with a simple semantic breakdown that looks similar to: `DO WHAT (to/from/in/...) WHAT, WHEN, WHERE, HOW, etc`. From there we process the resulting fragments of the query through individual neural modules.

Our definitions and roles of modules are inspired by the roles of different parts of speech in the query. Nouns and noun phrases correspond to data entities in code, and verbs describe actions or transformations performed on the data entities. Thus, data and transformations are handled by separate neural modules – an *entity discovery module* E and an *action module* A.

Every noun phrase in the semantic parse will be passed through the entity discovery module E to produce a single score for every token c^j_k in the code snippet c^j: $E(\text{"dataset"}, c^j) = [e^{ij}_1, e^{ij}_2, \ldots, e^{ij}_N]$. For each verb x the layout will have an instance of action module A: $A(x(p^x_1, p^x_2, ..., p^x_N), c^j) =$

$[a_1^{ij}, a_2^{ij}, \dots, a_N^{ij}]$, where p-s are children of x in the semantic parse. The top-level of the semantic parse is always an action module.

In Figure 2, the preposition FROM is used with noun "*dataset*", and thus will be combined with Arg_1 into a single input. We provide more detail on how such tuples of inputs are handled in Section 3.3.

3.2 Entity Discovery Module

The entity discovery module receives a string that references a data entity. Its goal is identifying locations in the code that have high relevance to that string. The architecture of the module is shown in Figure 3. Given an entity string, "dataset" in the example, and a sequence of code tokens $[c_1^j, \dots, c_N^j]$, entity module first obtains contextual code token representation using RoBERTa model that is initialized from CodeBERT-base checkpoint. The resulting embedding is passed through a two-layer MLP to obtain a score for every individual code token $c_k^j : 0 \leq e_k^{ij} \leq 1$. Thus, the total output of the module is a a vector of scores: $[e_1^{ij}, e_2^{ij}, \dots, e_N^{ij}]$. To prime the entity discovery module for measuring relevancy between code tokens and input, we fine-tune it with noisy supervision, as detailed below.

Noisy Supervision for Entity Discovery Module We create noisy supervision for the entity discovery module by using outputs of the keyword matching and Python static code analyzer. For the keyword matching, if a code token is an exact match for one or more tokens in the input string, its supervision label is set to 1, otherwise it is 0. Same is true if the code token is a substring or a superstring of one or more input string tokens. For some very common nouns we also manually added their "synonyms", e.g. "map" for "dict", etc.

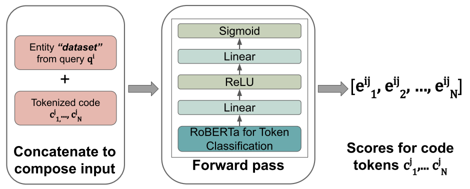

Entity "dataset" from query q^i
+
Tokenized code $c^j_{1,\dots,} c^j_N$
Concatenate to compose input
Sigmoid
Linear
ReLU
Linear
RoBERTa for Token Classification
Forward pass
$[e^{ij}_1, e^{ij}_2, \dots, e^{ij}_N]$
Scores for code tokens $c^j_1, \dots c^j_N$

Figure 3: Model architecture used for the entity discovery module

We used the static code analyzer to extract information about statically known data types. We cross-matched this information with the query to discover whether the query references any datatypes found in the code snippet. If that is the case, the corresponding code tokens are assigned supervision label 1, and all the other tokens are assigned to 0. For noisy supervision, we used equal numbers of $(query, code)$ pairs from the dataset, as well as random pairs of queries and code snippets.

3.3 Action Module

The action module produces a probability score that measures the amount of evidence of the given action happening in the code. The rest of the discussion considers the case where the action module only has data entity arguments. In the scenario of nested action modules, we flatten such layout and take conjunction of individual action scores. For example, for a layout with two nested actions x and y, and p_i^x and p_j^y are their corresponding arguments: $x(p_1^x, ..., p_{i-1}^x, y(p_1^y, ...p_l^y), p_{i+1}^x, ..., p_k^x)$. The final output is the conjunction of individual module outputs: $A(x(p_1^x, ..., p_{i-1}^x, p_{i+1}^x, p_k^x)) \cdot A(y(p_1^y, ..., p_l^y))$, where all remaining p-s are data entities, and A is the function computing action module score.

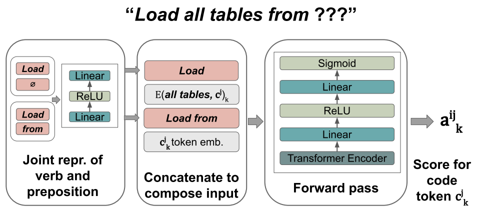


Figure 4: Diagram of estimation process for action module score.

The design of the action module is inspired by fill-in-the-blank style question answering, Figure 4 provides a high-level illustration of the module. In the example, for the query "*Load all tables from dataset*", the action module receives only part of the full query "*Load all tables from ???*". In this case, the desired output for the action module should correspond to the argument "*dataset*". If the code snippet corresponds to the query, then the action module should be able to deduce this missing part from the code and the rest of the query. We pre-train the action module using the output scores of the entity discovery module as supervision. The similarity score between the outputs of two modules measures how well the query matches the code. More formally, the action module performs three steps: masking, estimating, and computing similarity. For consistency, we always mask the last data entity argument.

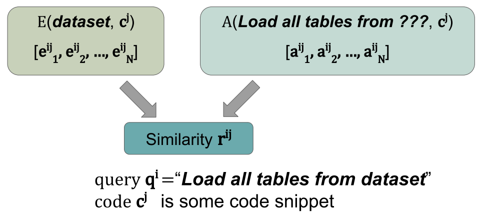

E(*dataset*, c^j)
[e^ij_1, e^ij_2, ..., e^ij_N]
A(*Load all tables from ???*, c^j)
[a^ij_1, a^ij_2, ..., a^ij_N]
Similarity r^ij
query q^i = "*Load all tables from dataset*"
code c^j is some code snippet

Figure 5: The similarity for query q_i and code c_j is computed as similarity score between the score outputs of entity discovery and action modules.

After masking one input, we compute joint embeddings for verb and preposition for each data entity argument. This is done with a 2-layer MLP model, as illustrated in the left-most part of Figure 4. If a data entity does not have a preposition associated with it, that part of the input is filled with zeros. The joint verb-preposition embedding is stacked with the code token embedding t_k and entity discovery module output for that token, this is referenced in the middle part of Figure 4. This vector is passed through a transformer encoder model, followed by a 2-layer MLP and a sigmoid layer to output score a_k^{ij}, illustrated in the right-most part of the Figure 4.

The similarity of action module output scores to entity discovery module output scores is computed with a dot product, as shown in Figure 5. We normalize this similarity to make it a probability, which is the final output of the action module. If this is the only action in the query, this probability score will also be the output of the entire model for (q_i, c_j) pair: r_{ij}, otherwise r_{ij} will be the product of probability scores of all nested actions in the layout.

3.4 Training and Inference

We train our model through supervised pre-training, as is discussed in Sections 3.2 and 3.3, followed by end-to-end training. End-to-end training objective is binary classification - given a pair of query q^i and code c^j, the model predicts probability r^{ij} that they are related. In the end-to-end training, we use positive examples taken directly from the dataset - (q^i, c^i), as well as negative examples composed through the combination of random queries and code snippets. The goal of end-to-end training is fine-tuning parameters of entity discovery and action modules, including the weights of the RoBERTA models used for code token representation.

Batching is hard to achieve for our model, so for the interest of time efficiency we do not perform inference on all distractor code snippets in the code dataset. Instead, for a given query we evaluate top-K highest ranked code snippets as outputted by a baseline model. Essentially, we evaluate our model in a re-ranking setup, and thus deploy ranking evaluation metrics. We interpret the probabilities outputted by the model as ranking scores. More details about this procedure are provided in Section 4.1.

4 Experiments

4.1 Experiment Setting

Dataset We conduct experiments on two datasets: **CodeSearchNet** (Husain et al., 2019) and **CoSQA** (Huang et al., 2021). We use the Python portion of CodeSearchNet, which contains over 500K functions with their docstring, and CoSQA has over 20K natural language web queries. We parse all queries with CCG parser, as described in Section 4.1, and leave out examples that could not be parsed. After this preprocessing, we train our model on 40% of the CodeSearchNet dataset and 70% of the CoSQA dataset, the exact sizes of datasets are shown in Table 1. For our baselines, we use the same reduced dataset for fine-tuning. In addition, we also experiment with fine-tuning all models on an even smaller subset of CodeSearchNet dataset, using only 5000 and 10000 examples for fine-tuning. The goal of this experiment is to see whether the modular design can help with generalizing on smaller datasets.

Dataset	Train	Valid	Test
CodeSearchNet	162801	8841	8905
CoSQA	14210	-	-
WebQueryTest	-	-	662

Table 1: Dataset statistics after parsing.

The results reported are obtained on the test set of CodeSearchNet for models trained on CodeSearchNet training data, or WebQueryTest (Lu et al., 2021) - a small natural language web query dataset of document-code pairs - for models trained on CoSQA data.

Compared Methods We compare NS^3 with various state-of-the-art methods, including some traditional approaches for document retrieval and pretrained large NLP language models. (1) **BM25** is a ranking method to estimate the relevance of documents to a given query. (2) **RoBERTa (code)** is a variant of RoBERTa (Liu et al., 2019) pretrained on the CodeSearchNet corpus. (3) **CuBERT** (Kanade et al., 2020) is a BERT Large model pretrained on 7.4M Python files from GitHub. (4) **CodeBERT** is an encoder-only Transformer model trained on unlabeled source code via masked language modeling (MLM) and replaced token detection objectives. (5) **GraphCodeBERT** is a pretrained Transformer model using MLM, data flow edge prediction, and variable alignment between code and the data flow.

Evaluation and Metrics We follow CodeSearchNet's original approach for evaluation for test instance (q, c), comparing the output against outputs over a fixed set of 999 distractor code snippets. We use two evaluation metrics: Mean Reciprocal Rank (MRR) and Precision@K (P@K). (1) **MRR** evaluates a list of possible code snippets retrieved for a sample query. The snippets are ordered by the predicted probability. (2) **P@K** is the proportion of the top-*K* documents closest to the given query. For each query, if the correct code snippet is among the first *K* retrieved code snippets, we score it 1, otherwise 0. We report this metric for K=1, 3, and 5.

Following a common approach in information retrieval, we perform two-step evaluation. In the first step, we obtain CodeBERT's output against 999 distractors. In the second step, we use NS^3 to re-rank the top 10 predictions of CodeBERT and to rank the correct answer. This way the evaluation is much faster, since unlike our modular approach, CodeBERT can be fed examples in batches. And as we will see from the results, we see improvement in final performance in all scenarios.

Parsing We used the NLTK Python package for our implementation of CCG parser. We built a vocabulary of predicates by identifying some commonly used verbs, such as "*convert*", "*find*", "*remove*", "*get*", etc. We also manually added commonly used nouns and their synonyms in Python to the vocabulary, e.g. "*dict*", and "*map*" as a synonym to it. Afterwards, we constructed the lexicon (rules) of the parser using the predicates we defined in the previous step. Besides the predicates that we had defined manually, we have also included "catch-all" predicates, for less-common verbs or nouns and adjectives. In attempt to parse as much of the datasets as possible, we preprocessed the queries by removing preceding question words, punctuation marks, and some specific words and phrases, e.g. those that specify a programming language or version, such as "in Python", "Python 2.7", etc. Full implementation of our parser including its entire lexicon and vocabulary can be found at `https://anonymous.4open.science/r/ccg_parser-4BC6`.

Pretrained Models Action and entity discovery modules each embed code tokens with a RoBERTa model, that has been initialized from a checkpoint of pretrained CodeBERT model [1]. We fine-tune these models during the pretraining phases, as well as during final end-to-end training phase.

Hyperparameters The MLPs in entity discovery and action modules have 2 layers with input dimension of 768. We use dropout in these networks with ratio 0.1. The learning rate for pretraining and end-to-end training phases was chosen from the range of 1e-6 to 6e-5. We use early stopping with evaluation on unseen validation set for model selection during action module pretraining and end-to-end training. For entity discovery model selection we performed manual inspection of produced scores on unseen examples. For evaluating the CuBERT, CodeBERT and GraphCodeBERT baselines we use the hyperparameters as reported in their original papers. For RoBERTa (code), we perform the search for learning rate during fine-tuning stage in the same interval as for our model. For model selection on baselines we also use early stopping.

4.2 Results

Performance Comparison Tables 2 and 3 present the performance evaluated on testing portion of CodeSearchNet dataset, and WebQueryTest dataset correspondingly. As it can be seen, our proposed model outperforms the baselines.

[1] `https://huggingface.co/microsoft/codebert-base`

Method	CSN				CSN-10K				CSN-5K			
	MRR	P@1	P@3	P@5	MRR	P@1	P@3	P@5	MRR	P@1	P@3	P@5
BM25	0.209	0.144	0.230	0.273	0.209	0.144	0.230	0.273	0.209	0.144	0.230	0.273
RoBERTa (code)	0.842	0.768	0.905	0.933	0.461	0.296	0.545	0.664	0.29	0.146	0.324	0.438
CuBERT	0.225	0.168	0.253	0.294	0.144	0.081	0.166	0.214	0.081	0.03	0.078	0.118
CodeBERT	0.873	0.803	0.939	0.958	0.69	0.550	0.799	0.873	0.680	0.535	0.794	0.870
GraphCodeBERT	0.812	0.725	0.880	0.919	0.786	0.684	0.859	0.901	0.773	0.677	0.852	0.892
NS^3	**0.924**	**0.884**	**0.959**	**0.969**	**0.826**	**0.753**	**0.886**	**0.908**	**0.823**	**0.751**	**0.881**	**0.913**
Upper-bound	0.979				0.939				0.936			

Table 2: Mean Reciprocal Rank (MRR) scores (higher is better), Precision @1, @3, and @5 (higher is better) for methods trained on different subsets from CodeSearchNet dataset.

Because of our evaluation strategy, we cannot improve the performance in the cases where the correct code snippet was not ranked among the top-10 results returned by the CodeBERT model. The rows labelled "Upper-bound" in Tables 2 and 3 measure the best performance that is possible to achieve with this evaluation strategy.

Perturbed Query Evaluation In this section we study how well our model can capture the difference between a query and its perturbed version. This experiment is designed to see how sensitive the models are to small changes in the query. Our expectation is that a sensitive model will not rate the perturbed query as high as the original one, because it no longer correctly describes the code snippet. Whereas a model that tends to over-generalize and ignore details of the query will likely rate the perturbed query similar to the original. We start from pairs of (query, code), that both our model and CodeBERT predict correctly. We then proceed to introduce small perturbations to the query, we used 100 different (query, code) pairs as a starting point, and for each query we generated 20 different perturbations. To account for calibration of the models, we measure this sensitivity through ratio of the perturbed query score over original query score. We generated two types of perturbations. For the first type, we used queries that had a single verb and a single data entity argument to that verb. To generate its perturbations, we replaced the correct data argument with a data argument sampled randomly from another query. The result is shown in Figure 6 in bar labelled "$V(arg_1) \to V(arg_2)$". For the next experiment, we again used queries that had a single verb and a single data argument to that verb. This time, to generate perturbations we replace the verb argument with a randomly sampled one from another query. The results for this scenario are demonstrated in Figure 6 under in bar labelled "$V_1(arg) \to V_2(arg)$".

Method	CoSQA			
	MRR	P@1	P@3	P@5
BM25	0.103	0.05	0.119	0.142
RoBERTa (code)	0.279	0.159	0.343	0.434
CuBERT	0.127	0.067	0.136	0.187
CodeBERT	0.345	0.175	0.42	0.54
GraphCodeBERT	0.435	0.257	0.538	0.628
NS^3	**0.551**	**0.445**	**0.619**	**0.668**
Upper-bound	0.736			

Table 3: Mean Reciprocal Rank(MRR) scores (higher is better), Precision @1, @3, and @5 (higher is better) for different methods trained on CoSQA dataset.

Query Complexity Effect on Performance Here we present the breakdown of the performance for our method vs baselines, using two proxies for the complexity and compositionality of the query. The first one is the maximum depth of the query. We define the maximum depth as the maximum number of nested action modules in the query. The results for this experiment are presented in Figure 7a. As we can see, NS^3 provides improvement over the baseline in all scenarios. It is interesting to note, that while CodeBERT achieves the best performance on queries with depth 3+, our model's performance peaks at depth = 1. We hypothesize that this can be

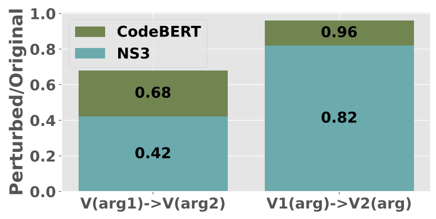
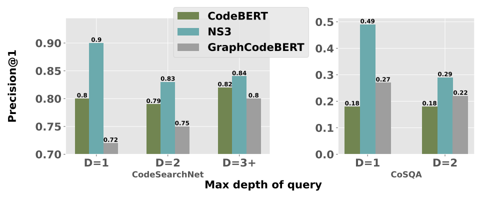
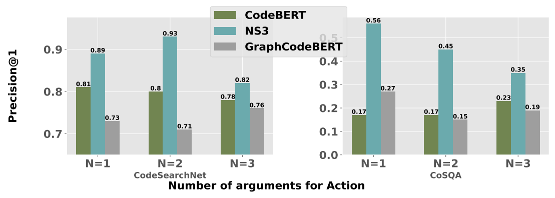


Figure 6: Ratio of the perturbed query score to the original query score (lower is better) on CSN dataset.

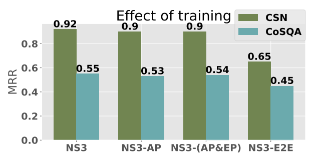


(a)

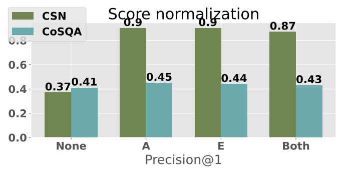


(b)

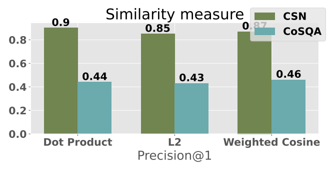
Figure 7: We report Precision@1 scores. (a) Performance of our proposed method and baselines broken down by average number of arguments per action in a single query. (b) Performance of our proposed method and baselines broken down by number of arguments in queries with a single action.



(a)



(b)



(c)

Figure 8: Performance of NS^3 on the test portion of CSN dataset with different ablation variants. (a) Skipping one, or both pretraining procedures, and only training end-to-end. (b) Using no normalization on output scores (**None**), L1 norm or sigmoid. (c) Performance with different options for computing action and entity discovery output similarities.

related to the automated parsing procedure, as parsing errors are more likely to be propagated in deeper queries. Further studies with carefully curated manual parses are necessary to understand this phenomenon better.

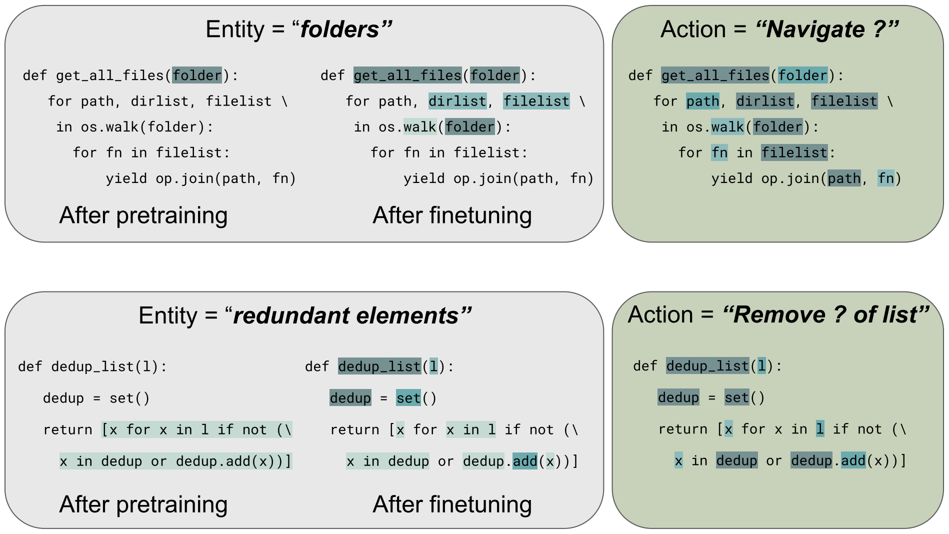
Another proxy for the query complexity we consider, is the number of data arguments to a single action module. If the previous scenario is breaking down the performance by the depth of the query, in this case we are considering its "width". We measure the average number of entity arguments per action module in the query. In the parsed portion of our dataset we have queries that range from 1 to 3 textual arguments per action verb. The results for this evaluation is presented in Figure 7. As it can be seen, there is no significant difference in performances between two groups of queries in either CodeBERT or our proposed method - NS^3.

Effect of Pretraining In an attempt to better understand the individual effect of the two modules as well as the roles of their pretraining and training procedures, we performed two additional ablation studies. In the first one, we compare the final performance of the original model with two versions where we skipped part of the pretraining. The model noted as ($NS^3 - AP$) was trained with pretrained entity discovery module, but no pretraining was done for action module, instead we proceeded to the end-to-end training directly. For the model called $NS^3 - (AP\&EP)$, we skipped both pretrainings of the entity and action modules, and just performed end-to-end training. Figure 8a demonstrates that combined pretraining is important for the final performance.

Additionally, we wanted to measure how effective the pretraining was on its own, i.e. without end-to-end training. The results for this variant are reported in Figure 8a under name $NS^3 - E2E$. Figure 8a shows that there is a huge performance dip in this scenario, and while the performance is better than random, it is obvious that end-to-end training is crucial for the NS^3 model.

SCORE NORMALIZATION We performed some experiments to determine the importance of normalizing the output of our modules to a proper probability distribution. In Figure 8b we demonstrate the performance achieved using no normalization at all, normalizing either action or entity discovery module or normalizing both. In all cases we used L1 normalization, since our output scores are non-negative. The version that is not normalized at all performs the worst on both datasets. The performances of the other three versions are close on both datasets.

SIMILARITY METRIC Additionally, we experimented with replacing the dot product similarity with a different similarity metric. In particular, in Figure 8c we compare the performance achieved using



```
def get_all_files(folder):
   for path, dirlist, filelist \
    in os.walk(folder):
      for fn in filelist:
          yield op.join(path, fn)
```

```
def get_all_files(folder):
   for path, dirlist, filelist \
    in os.walk(folder):
      for fn in filelist:
          yield op.join(path, fn)
```

```
def get_all_files(folder):
   for path, dirlist, filelist \
    in os.walk(folder):
      for fn in filelist:
          yield op.join(path, fn)
```

```
def dedup_list(l):
   dedup = set()
   return [x for x in l if not (\
     x in dedup or dedup.add(x))]
```

```
def dedup_list(l):
   dedup = set()
   return [x for x in l if not (\
     x in dedup or dedup.add(x))]
```

```
def dedup_list(l):
   dedup = set()
   return [x for x in l if not (\
     x in dedup or dedup.add(x))]
```

Figure 9: The leftmost column shows output scores of the entity discovery module after pretraining for the ***emphasized part*** of the query. The middle column shows the scores after completing the end-to-end training. The rightmost column shows the scores of the action module, ***emphasized text*** is the input the module received for each query. Darker highlighting demonstrates higher score.

dot product similarity, L2 distance, and weighted cosine similarity. The difference in performance among different versions is marginal.

QUALITATIVE CASE STUDY Finally, we demonstrate some examples of the scores produced by our modules at different stages of training. Figure 9 shows module score outputs for two different queries and with thier corresponding code snippets. The first column shows the output of the entity discovery module after pretraining, while the second and third columns demonstrate the outputs of entity discovery and action modules after the end-to-end training. We can see that in the first column the model identifies syntactic matches, such as "folder" and a list comprehension, which "elements" could be related too. After fine-tuning we can see there is a wider range of both syntactic and some semantic matches present, e.g. "dirlist" and "filelist" are correctly identified as related to "folders".

5 RELATED WORK

Different deep learning models have proved quite efficient when applying to programming languages and code. Prior works have studied and reviewed the uses of deep learning for code analysis in general and code search in particular (Xu et al., 2021; Lu et al., 2021).

A number of approaches to deep code search is based on creating a relevance-predicting model between text and code. Gu et al. (2018) propose using RNNs for embedding both code and text to the same latent space. Several variants of large pre-trained models for code were introduced, e.g., CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2021), and CuBERT (Kanade et al., 2020). These methods also embed the query and code jointly. On the other hand, DGMS (Ling et al., 2021) capitalizes the inherent graph-like structure of programs to formulate code search as graph matching. A few works propose enriching the models handling code embedding by adding additional code analysis information, such as semantic and dependency parses (Du et al., 2021; Akbar & Kak, 2019), variable renaming and statement permutation (Gu et al., 2020), as well as structures such as abstract syntax tree of the program (Haldar et al., 2020; Wan et al., 2019). A few other approaches have dual formulations of code retrieval and code summarization (Chen & Zhou, 2018; Yao et al., 2019; Ye et al., 2020; Bui et al., 2021) In their work (Heyman & Cutsem, 2020) propose considering the code search scenario where short annotative descriptions of code snippets are provided.

6 CONCLUSION

We presented NS^3 a modular method for semantic code search. In contrast to existing code search methods, NS^3 can better capture the compositional nature of queries. In an extensive evaluation, we show that this method works better than strong but unstructured baselines.

REFERENCES

Shayan A. Akbar and Avinash C. Kak. SCOR: source code retrieval with semantics and order. In Margaret-Anne D. Storey, Bram Adams, and Sonia Haiduc (eds.), *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*, pp. 1–12. IEEE / ACM, 2019. doi: 10.1109/MSR.2019.00012. URL https://doi.org/10.1109/MSR.2019.00012.

Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. Neural module networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pp. 39–48. IEEE Computer Society, 2016. doi: 10.1109/CVPR.2016.12. URL https://doi.org/10.1109/CVPR.2016.12.

Yoav Artzi, Kenton Lee, and Luke Zettlemoyer. Broad-coverage CCG semantic parsing with AMR. In Lluís Màrquez, Chris Callison-Burch, Jian Su, Daniele Pighin, and Yuval Marton (eds.), *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015, Lisbon, Portugal, September 17-21, 2015*, pp. 1699–1710. The Association for Computational Linguistics, 2015. doi: 10.18653/v1/d15-1198. URL https://doi.org/10.18653/v1/d15-1198.

Marco Baroni. Linguistic generalization and compositionality in modern artificial neural networks. *CoRR*, abs/1904.00157, 2019. URL http://arxiv.org/abs/1904.00157.

Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer. *CoRR*, abs/2004.05150, 2020. URL https://arxiv.org/abs/2004.05150.

Hanoz Bhathena, Angelica Willis, and Nathan Dass. Evaluating compositionality of sentence representation models. In Spandana Gella, Johannes Welbl, Marek Rei, Fabio Petroni, Patrick S. H. Lewis, Emma Strubell, Min Joon Seo, and Hannaneh Hajishirzi (eds.), *Proceedings of the 5th Workshop on Representation Learning for NLP, RepL4NLP@ACL 2020, Online, July 9, 2020*, pp. 185–193. Association for Computational Linguistics, 2020. doi: 10.18653/v1/2020.repl4nlp-1.22. URL https://doi.org/10.18653/v1/2020.repl4nlp-1.22.

Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations. In Fernando Diaz, Chirag Shah, Torsten Suel, Pablo Castells, Rosie Jones, and Tetsuya Sakai (eds.), *SIGIR '21: The 44th International ACM SIGIR Conference on Research and Development in Information Retrieval, Virtual Event, Canada, July 11-15, 2021*, pp. 511–521. ACM, 2021. doi: 10.1145/3404835.3462840. URL https://doi.org/10.1145/3404835.3462840.

R. Ian Bull, Andrew Trevors, Andrew J. Malton, and Michael W. Godfrey. Semantic grep: Regular expressions + relational abstraction. In Arie van Deursen and Elizabeth Burd (eds.), *9th Working Conference on Reverse Engineering (WCRE 2002), 28 October - 1 November 2002, Richmond, VA, USA*, pp. 267–276. IEEE Computer Society, 2002. doi: 10.1109/WCRE.2002.1173084. URL https://doi.org/10.1109/WCRE.2002.1173084.

Qingying Chen and Minghui Zhou. A neural framework for retrieval and summarization of source code. In Marianne Huchard, Christian Kästner, and Gordon Fraser (eds.), *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pp. 826–831. ACM, 2018. doi: 10.1145/3238147.3240471. URL https://doi.org/10.1145/3238147.3240471.

Lun Du, Xiaozhou Shi, Yanlin Wang, Ensheng Shi, Shi Han, and Dongmei Zhang. Is a single model enough? mucos: A multi-model ensemble learning approach for semantic code search. In Gianluca Demartini, Guido Zuccon, J. Shane Culpepper, Zi Huang, and Hanghang Tong (eds.), *CIKM '21: The 30th ACM International Conference on Information and Knowledge Management, Virtual Event, Queensland, Australia, November 1 - 5, 2021*, pp. 2994–2998. ACM, 2021. doi: 10.1145/3459637.3482127. URL https://doi.org/10.1145/3459637.3482127.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. In Trevor Cohn, Yulan He, and Yang Liu (eds.), *Findings of the Association* *for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pp. 1536–1547. Association for Computational Linguistics, 2020. doi: 10.18653/v1/2020.findings-emnlp.139. URL `https://doi.org/10.18653/v1/2020.findings-emnlp.139`.

Wenchao Gu, Zongjie Li, Cuiyun Gao, Chaozheng Wang, Hongyu Zhang, Zenglin Xu, and Michael R. Lyu. Cradle: Deep code retrieval based on semantic dependency learning. *CoRR*, abs/2012.01028, 2020. URL `https://arxiv.org/abs/2012.01028`.

Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (eds.), *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pp. 933–944. ACM, 2018. doi: 10.1145/3180155.3180167. URL `https://doi.org/10.1145/3180155.3180167`.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. Graphcodebert: Pre-training code representations with data flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL `https://openreview.net/forum?id=jLoC4ez43PZ`.

Rajarshi Haldar, Lingfei Wu, Jinjun Xiong, and Julia Hockenmaier. A multi-perspective architecture for semantic code search. In Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel R. Tetreault (eds.), *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pp. 8563–8568. Association for Computational Linguistics, 2020. doi: 10.18653/v1/2020.acl-main.758. URL `https://doi.org/10.18653/v1/2020.acl-main.758`.

Geert Heyman and Tom Van Cutsem. Neural code search revisited: Enhancing code snippet retrieval through natural language intent. *CoRR*, abs/2008.12193, 2020. URL `https://arxiv.org/abs/2008.12193`.

Junjie Huang, Duyu Tang, Linjun Shou, Ming Gong, Ke Xu, Daxin Jiang, Ming Zhou, and Nan Duan. Cosqa: 20,000+ web queries for code search and question answering. In Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli (eds.), *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), ACL 2021, Online, August 1-6, 2021*, pp. 5690–5700. Association for Computational Linguistics, 2021. doi: 10.18653/v1/2021.acl-long.442. URL `https://doi.org/10.18653/v1/2021.acl-long.442`.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. abs/1909.09436, 2019. URL `https://arxiv.org/abs/1909.09436`.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pp. 5110–5121. PMLR, 2020. URL `http://proceedings.mlr.press/v119/kanade20a.html`.

Xiang Ling, Lingfei Wu, Saizhuo Wang, Gaoning Pan, Tengfei Ma, Fangli Xu, Alex X. Liu, Chunming Wu, and Shouling Ji. Deep graph matching and searching for semantic code retrieval. *ACM Trans. Knowl. Discov. Data*, 15(5):88:1–88:21, 2021. doi: 10.1145/3447571. URL `https://doi.org/10.1145/3447571`.

Shangqing Liu, Xiaofei Xie, Lei Ma, Jing Kai Siow, and Yang Liu. Graphsearchnet: Enhancing gnns via capturing global dependency for semantic code search. *CoRR*, abs/2111.02671, 2021. URL `https://arxiv.org/abs/2111.02671`.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. abs/1907.11692, 2019. URL `https://arxiv.org/abs/1907.11692`.

Meili Lu, Xiaobing Sun, Shaowei Wang, David Lo, and Yucong Duan. Query expansion via wordnet for effective code search. In Yann-Gaël Guéhéneuc, Bram Adams, and Alexander Serebrenik (eds.), *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*, pp. 545–549. IEEE Computer Society, 2015. doi: 10.1109/SANER.2015.7081874. URL `https://doi.org/10.1109/SANER.2015.7081874`.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shouv, Long Zhou, Michele Tufano, MING GONG, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie LIU. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1), Online, Dec 7-10, 2021*. OpenReview.net, 2021. URL `https://openreview.net/forum?id=6lE4dQXaUcb`.

Steven P. Reiss. Semantics-based code search demonstration proposal. In *25th IEEE International Conference on Software Maintenance (ICSM 2009), September 20-26, 2009, Edmonton, Alberta, Canada*, pp. 385–386. IEEE Computer Society, 2009. doi: 10.1109/ICSM.2009.5306319. URL `https://doi.org/10.1109/ICSM.2009.5306319`.

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

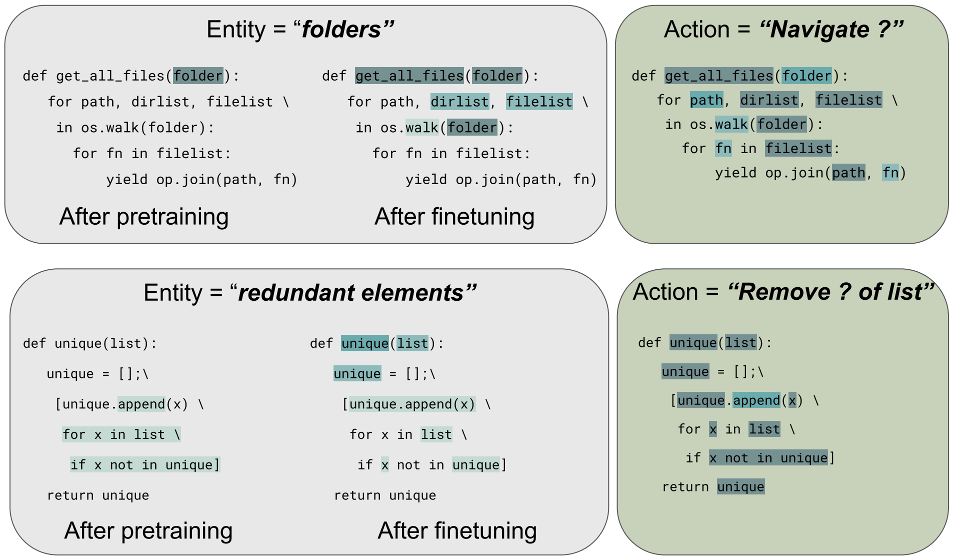
Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip S. Yu. Multi-modal attention network learning for semantic source code retrieval. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pp. 13–25. IEEE, 2019. doi: 10.1109/ASE.2019.00012. URL `https://doi.org/10.1109/ASE.2019.00012`.

Frank F. Xu, Bogdan Vasilescu, and Graham Neubig. In-ide code generation from natural language: Promise and challenges. *CoRR*, abs/2101.11149, 2021. URL `https://arxiv.org/abs/2101.11149`.

Ziyu Yao, Jayavardhan Reddy Peddamail, and Huan Sun. Coacor: Code annotation for code retrieval with reinforcement learning. In Ling Liu, Ryen W. White, Amin Mantrach, Fabrizio Silvestri, Julian J. McAuley, Ricardo Baeza-Yates, and Leila Zia (eds.), *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, pp. 2203–2214. ACM, 2019. doi: 10.1145/3308558.3313632. URL `https://doi.org/10.1145/3308558.3313632`.

Wei Ye, Rui Xie, Jinglei Zhang, Tianxiang Hu, Xiaoyin Wang, and Shikun Zhang. Leveraging code generation to improve code retrieval and summarization via dual learning. In Yennun Huang, Irwin King, Tie-Yan Liu, and Maarten van Steen (eds.), *WWW '20: The Web Conference 2020, Taipei, Taiwan, April 20-24, 2020*, pp. 2309–2319. ACM / IW3C2, 2020. doi: 10.1145/3366423.3380295. URL `https://doi.org/10.1145/3366423.3380295`.

Luke S. Zettlemoyer and Michael Collins. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. *CoRR*, abs/1207.1420, 2012. URL `http://arxiv.org/abs/1207.1420`.



```
def get_all_files(folder):
   for path, dirlist, filelist \
    in os.walk(folder):
      for fn in filelist:
          yield op.join(path, fn)
```

```
def get_all_files(folder):
   for path, dirlist, filelist \
    in os.walk(folder):
      for fn in filelist:
          yield op.join(path, fn)
```

```
def get_all_files(folder):
   for path, dirlist, filelist \
    in os.walk(folder):
      for fn in filelist:
          yield op.join(path, fn)
```

```
def unique(list):
   unique = [];\
    [unique.append(x) \
     for x in list \
      if x not in unique]
   return unique
```

```
def unique(list):
   unique = [];\
    [unique.append(x) \
     for x in list \
      if x not in unique]
   return unique
```

```
def unique(list):
   unique = [];\
    [unique.append(x) \
     for x in list \
      if x not in unique]
   return unique
```

Figure 10: The leftmost column shows output scores of the entity discovery module after pretraining for the ***emphasized part*** of the query. The middle column shows the scores after completing the end-to-end training. The rightmost column shows the scores of the action module, ***emphasized text*** is the input the module received for each query. Darker highlighting demonstrates higher score.

A Appendix

Additional examples Figure 10 contains more illustrations of the output scores of the action and entity discovery modules captured at different stages of training. The queries shown here are the same, but this time they are evaluated on different functions.