COBE: A Natural Language Code Search Robustness Benchmark

Jônatas Wehrmann
Pontifical Catholic University of Rio de Janeiro (PUC-Rio)
Rio de Janeiro, Brazil
wehrmann@puc-rio.br

Otávio Parraga
Pontifical Catholic University of Rio Grande do Sul (PUC-RS)
Porto Alegre, Brazil
otavio.parraga@edu.pucrs.br

Rodrigo C. Barros
Pontifical Catholic University of Rio Grande do Sul (PUC-RS)
Porto Alegre, Brazil
rodrigo.barros@pucrs.br

2022 International Joint Conference on Neural Networks (IJCNN) | 978-1-7281-8671-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/IJCNN55064.2022.9892610

***Abstract*—Benchmark frameworks and datasets allow us to analyze and understand the knowledge that NLP models capture about the world they were trained on. Several Transformer models have recently been adapted to code-related tasks such as code search, in which the goal is to find the most semantically relevant code given a query written in natural language. To achieve satisfactory performance, the retrieval models heavily rely on the quality of the query. In this paper, we introduce the Natural Language Code Search Robustness Benchmark (COBE), which provides a more holistic evaluation of the state-of-the-art models considering several aspects of the retrieval models, such as: (i) retrieval capabilities measured in multiple ranking metrics; (ii) robustness to a plethora of input perturbations; (iii) efficiency in terms of training and retrieval times; and (iv) stability across fine-tuning runs. We shed a light over important questions showing that simply computing performance-based retrieval metrics does not suffice to evaluate this kind of model. The proposed benchmark introduces novel metrics and measurement strategies that allow a rigorous quantitative analysis of input-query robustness while providing an understanding of model generalization behavior. We perform an extensive set of experiments using state-of-the-art models such as CodeBert, GraphCodeBert, and CodeT5. Those models are fine-tuned over many different scenarios in six programming languages. Several models trained in this study outperform their state-of-the-art counterparts, which provides evidence that the standard fine-tuning approach used in code search related work is sub-optimal. The proposed benchmark is a powerful tool to evaluate code search models, providing insights on how they behave during fine-tuning and how they are interpreting the input queries.**

***Index Terms*—natural language code search, robustness, benchmark**

I. Introduction

Most of the state-of-the-art models in Artificial Intelligence (AI) nowadays are trained on broad data at scale to be later adapted to any particular downstream task, a paradigm that has recently been called Foundation Models (FMs) [9]. FMs leverage from well-established techniques such as transfer learning [25], and enjoy the benefits of a parallel neural network architecture like the Transformer [26] to be trained self-supervisedly over massive amounts of data. Well-known examples of foundation models include BERT [4], CLIP [17], GPT-3 [2], and GPT-3 derived Codex [8].

Foundation models are now being designed to deal with all kinds of data, be it natural language text [4], [18], [19], images [3], [6], audio and speech [15], protein and molecular data [20], [21], or tabular data in general [31]. More recently, the research community has focused on designing FMs to source-code tasks [10], [11], [28], and the initial results have shown to be very promising. A partnership between OpenAI and Microsoft have led to the development of GitHub Copilot, an AI-based code completion tool whose inner workings are based on the Codex foundation model [8].

Code completion is but one of the many possible tasks that we can address when building machine learning models that learn from source code. Examples include natural language code search [10], which is the task of searching for code snippets through queries written in natural language; documentation generation (also known as code summarization) [14], which is basically generating natural language semantic descriptions to an input code snippet; unit test case generation [5], another generation task but this time focused on generating test cases to verify the correctness of functions and/or methods; and several others like program synthesis [13], refactoring [1], and vulnerability analysis [23].

The focus of this paper will be on natural language code search (NLCS), a task that has gain popularity considering the gains in productivity it can bring when done properly [22], [24], [27]. Typically, NLCS is performed by software engineers with the help of Q&A sites such as StackOverflow, which rely on qualified answers from other software engineers. The goal of machine learning based NLCS is to automate that task by retrieving source-code snippets from a data repository after semantically aligning both natural language descriptions and code snippets into a common latent semantic space.

NLCS is often evaluated as a typical retrieval task, making use of evaluation measures that are computed over rankings, such as the Mean Reciprocal Rank (MRR), the recall at k ($R@K$), or the average ranking position ($meanR$). However, most of the research community overlooks important aspects that should be taken into account when evaluating and deploying such models into real-world applications. For instance, little is known in terms of how the current foundation models for source-code tasks behave when dealing with insignificant input perturbations. For equally-performing models in the benchmark validation sets, are there differences across random initializations, fine-tune runs, or zero-shot performance in distinct programming languages?

978-1-7281-8671-9/22/$31.00 ©2022 IEEE

To answer some of those questions and shed a light over the real robustness of NLCS models, we introduce the Natural Language Code Search Robustness Benchmark (COBE). We argue that COBE allows us to analyze the current state-of-the-art approaches for NLCS in a more holistic fashion. The proposed benchmark introduces novel metrics and measurement strategies that allow a rigorous quantitative analysis of input-query robustness, while also providing a much better understanding of the actual generalization behavior of some of the main FMs that are pretrained on source code data.

We make use of COBE to perform an extensive set of experiments with state-of-the-art models CodeBert [10], GraphCodeBert [11], and CodeT5 [28], which are then fine-tuned in six distinct programming languages, spanning many different scenarios. With the help of COBE , we show that the standard fine-tuning approach used in NLCS is sub-optimal, and we provide novel state-of-the-art models that outperform their original published versions.

II. The COBE Benchmark

The Natural Language Code Search Robustness Benchmark (COBE) comprises data, data transformations, and evaluation measures. We describe them in detail next.

A. Dataset

In COBE, we make use of a cleaned version of CodeSearchNet [12] (Table I), a dataset that comprises pairs of natural language descriptions and corresponding code snippets. Originally, the data was collected from open-source projects in GitHub, and the number of instances can be seen in Table II. We choose this particular dataset due to its relevance for NLCS, since it was primarily released as a challenge for the task and widely used after that in many important studies.

B. Transformations

To explore different robustness aspects of the models, our benchmark supports a group of different transformations, each one modifying the natural language query in either character or word-level. To implement most of these transformations, we use the open-source library NLPAug [16], which makes available several NLP augmentations.

We divide the transformations into two main categories:

1) Character-Based: these transformations apply character-level changes in the inputs by replacing, modifying, or adding a specific character. In this category, we implement four transformations and except by the **case** transformation, all have similar functionality of changing $n\%$ of the characters in 30% of the words, where $n\%$ is user-defined.

- Case: changes a specific percentage of the case of the characters (from upper to lower and vice-versa).
- Replace: randomly replaces a group of characters.
- Noise: adds to the text new characters, polluting the text.
- Typo: similar to **replace**, but replacing the random behavior by a keyboard proximity weight.

2) Word-Based: while character-based modifications focus on syntactic robustness, word-based transformations focus on semantic-level robustness, by analyzing how changes in the order or the meaning of the words affect the model. Differently from previous transformations, these are modified proportionally to $n\%$, also user-defined.

- Synonym: replaces up to $n\%$ of the words in the sentence by synonyms.
- Swap: swaps the order of $n\%$ of the words in the sentence.

Slightly distinct from previous transformations, we also implement one that changes the description into a question by adding prefix *"How to"* and a question mark at the end. Table III shows examples of the proposed data transformations.

C. Evaluation Measures

We divide the proposed evaluation measures in COBE into two groups: retrieval quality and robustness.

Retrieval quality: following [12], for evaluating retrieval results we use Mean Reciprocal Ranking (MRR) as the main measure. In addition, we employ other ranking-related measures such as $R@K$ (reads "Recall at K"), which is the percentage of queries in which the ground-truth is one of the first K retrieved results; and the ground truth average ranking position (meanR).

Robustness: micro and macro averages of areas under the noise curves. For every transformation, we compute the MRR with different noise ratios, ranging from 0% to 50% in steps of 5%. Hence, we can compute a robustness curve, where the y-axis is the MRR value and the x-axis the noise ratio that is applied in the transformation.

III. Models

With our proposed benchmark, in this paper we evaluate three state-of-the-art models for code-related tasks, all of them pre-trained in large programming and natural language corpora. CodeBert [10] was introduced as the first attempt to use a BERT-like model with both modalities of data, natural and programming language, to perform tasks that require profound understanding of source code. Besides masked language modeling, it also makes use of a task to detect replaced tokens to help in learning bimodal representations.

GraphCodeBert [11] is an optimized version of CodeBert, with the major improvement being the addition of code-specific information during the pre-training by encoding the relationship between the variables. The authors use two structure-aware tasks: one that identifies from where the value of a variable comes from and the other that identifies which identifiers are edges in the AST representation.

Differently from previous BERT-like architectures, CodeT5 [28] is an encoder-decoder transformer adapted to a multi-task learning framework. During pre-training, this model also uses tasks that leverage code-specific information originated from the AST representation together with a more common denoising task such as mask prediction. Considering that for NLCS we only need the encoded deep representation

TABLE I
LENGTH OF THE CODE SNIPPETS FOR TRAINING AND CODEBASE SPLITS BEFORE AND AFTER REMOVAL OF DOCSTRINGS AND CODE COMMENTS.

Language	Training Split			Codebase Split		
	Raw	Filtered	Reduction (%)	Raw	Filtered	Reduction (%)
go	411.0 ± 479.6	372.6 ± 412.8	0.07	354.9 ± 393.7	328.3 ± 334.9	0.09
java	610.2 ± 613.5	559.0 ± 512.0	0.08	557.9 ± 566.3	512.1 ± 481.3	0.08
javascript	643.8 ± 1674.0	555.4 ± 1256.4	0.15	687.1 ± 2984.6	581.6 ± 2282.9	0.14
php	584.1 ± 515.3	539.5 ± 446.8	0.07	593.7 ± 523.5	553.8 ± 466.8	0.08
python	885.6 ± 811.7	547.0 ± 511.6	0.41	953.6 ± 920.7	559.7 ± 539.9	0.38
ruby	430.7 ± 421.5	430.7 ± 421.5	0.00	465.3 ± 479.4	465.3 ± 479.4	0.00

TABLE II
NUMBER OF INSTANCES IN THE DATASET FOR EACH AVAILABLE PROGRAMMING LANGUAGE.

	train	val	test	codebase
go	167,288	7,325	8,122	28,120
java	164,923	5,183	10,955	40,347
javascript	58,025	3,885	3,291	13,981
php	241,241	12,982	14,014	52,660
python	251,820	13,914	14,918	43,827
ruby	24,927	1,400	1,261	4,360
Full dataset	908,224	44,689	52,561	183,295

TABLE III
EXAMPLE OF THE TRANSFORMATIONS IN A TOY SENTENCE FOR DIFFERENT VALUES OF n.

Transformation	Noise ratio ($n\%$)	
	0.1	0.5
original	a simple function	a simple function
case	a simPle fUnction	a sIMPLe FUnctIoN
synonym	a round eyed function	a simple part
replace	a siRple function	a simple fudVMiok
swap	a function simple	function simple a
noise	a simploe function	a sCiom@ple function
typo	a qimple function	a simple fuJcY&0n
question	How to a simple function?	How to a simple function?

of the input, we can get rid of the CodeT5 decoder, which was originally designed to be fine-tuned in tasks where the output can be treated as a sentence.

We use the pre-trained version of those models and fine-tune them on the CodeSearchNet dataset for retrieval. CodeBert and GraphCodeBert are pre-trained for masked token prediction using the CodeSearchNet dataset, while CodeT5 is trained for multiple tasks and uses additional data.

In order to allow such models to perform code search (code retrieval), we add a linear layer on top of the output of the [CLS] token vector and use it as a global representation for both code snippets and natural language docstrings, which are used as textual queries. Let $\mathcal{T}(C) = \overline{\mathbf{c}}$ be the backbone transformer with the additional fully-connected layer that projects code embeddings $t \times d$ onto a latent space $d \in \mathbf{R}^d$. Given that such transformer was pre-trained in both code and docstrings, we reuse the same model to encode both code and comments into vectors $\overline{\mathbf{c}}$ and $\overline{\mathbf{q}}$.

In summary, for training we encode a batch of aligned pairs of codes $\mathcal{T}(\mathbf{C}) = \overline{C}$ and batch of comments $\mathcal{T}(\mathbf{Q}) = \overline{Q}$. $\mathbf{C}$ is an $n \times t \times d$ tensor that comprises a collection of n C matrices. The pairwise similarity matrix is then computed as $s(\overline{Q}, \overline{C})$, and code snippets are ranked accordingly. We can use several distinct similarity functions to define $s(\cdot, \cdot)$, though following recent work we simply use the natural inner product of query and code vectors, or the cosine similarity (inner product but with vectors normalized to have unit norm). We explore such differences in some experiments in later sections. Following, we discuss the loss function options that we consider for training such models.

A. Loss functions

Retrieval models have to be optimized in order to make related code-docstring pairs to present high similarity between each other. Loss functions designed for retrieval tasks often use the remaining not-aligned pairs from the batches as contrastive examples to push unrelated pairs apart in the latent space.

There are two main loss functions used for retrieval models: (i) cross-entropy loss; and (ii) contrastive hinge-loss. Following we discuss both options as well as some hypotheses regarding each of them.

Cross-entropy: it is the standard choice for training classification models, and recently self-supervised models as well. It is straightforward to be used for retrieval systems, since instead of using class logits as input for the softmax operator, we use the similarity matrix $\mathbf{S}$ as prediction scores:

$$\mathcal{J}_{CE}(\overline{\mathbf{c}}, \overline{\mathbf{q}})_i = -\log\Big(\frac{s(\overline{\mathbf{c}}, \overline{\mathbf{q}})_i}{\sum_j^{n_c} s(\overline{\mathbf{c}}, \overline{\mathbf{q}})_{ij}\tau}\Big) \tag{1}$$

where we compute the entropy value for the i^{th} query w.r.t the n_c code snippet vectors from the batch. Note that τ is a temperature hyperparameter used to sharpen or soften the data distribution of the similarity matrix.

Contrastive loss: the contrastive hinge loss is a margin triplet-based optimizer. Its main goal is to make similarity $s(\overline{\mathbf{q}}, \overline{\mathbf{c}})$ of similar pairs larger than $s(\overline{\mathbf{q}}, \overline{\mathbf{c}}')$ of contrastive pairs (denoted with $'$) by a margin α. It usually comes in either of two main versions: sum of all hinges $\mathcal{J}_{SUM}$ and sum of hard contrastive hinges $\mathcal{J}_{MAX}$. The first one is the sum of the contrastive hinge values, which is defined in Equation 2.

$$\mathcal{J}_{SUM}(\overline{\mathbf{c}}, \overline{\mathbf{q}}) = \sum_{\overline{\mathbf{q}}'}[\alpha - s(\overline{\mathbf{c}}, \overline{\mathbf{q}}) + s(\overline{\mathbf{c}}, \overline{\mathbf{q}}')] + \sum_{\overline{\mathbf{c}}'}[\alpha - s(\overline{\mathbf{q}}, \overline{\mathbf{c}}) + s(\overline{\mathbf{q}}, \overline{\mathbf{c}}')] \quad (2)$$

Such formulation is very stable and rarely diverges, although it optimizes the average case and often gets trapped into local minima, and thus can be considered a sub-optimal choice. It is often more useful when training models from scratch rather than for fine-tuning. Its counterpart is the $\mathcal{J}_{MAX}$ that uses only hinge values of the hard-contrastive samples, as seen in Equation 3.

$$\mathcal{J}_{MAX}(\overline{\mathbf{c}}, \overline{\mathbf{q}}) = \max_{\overline{\mathbf{q}}'}[\alpha - s(\overline{\mathbf{c}}, \overline{\mathbf{q}}) + s(\overline{\mathbf{c}}, \overline{\mathbf{q}}')] + \max_{\overline{\mathbf{c}}'}[\alpha - s(\overline{\mathbf{q}}, \overline{\mathbf{c}}) + s(\overline{\mathbf{q}}, \overline{\mathbf{c}})] \quad (3)$$

By calculating gradients from the larger hinge value in the training batch, the network is being optimized to handle and improve upon the hardest case. That yields far better results, though sometimes the training will diverge or suffer from stability issues given that it is more sensitive to noise and may generate large gradients based on unwanted samples.

Recently, we have seen the introduction of an exponentially weighted loss [29], [30] that smoothly combines both functions, namely $\mathcal{J}_{WE}$:

$$\mathcal{J}_{WE} = \lambda(\epsilon) \cdot \mathcal{J}_m + (1 - \lambda(\epsilon)) \cdot \mathcal{J}_s \quad (4)$$

$$\lambda = 1 - \eta^{\epsilon}. \quad (5)$$

$\mathcal{J}_{WE}$ uses all hinge-loss values in the beginning of the training, and gives more weight to the hard cases as the training progresses. It takes the η hyperparameter, which is the exponential growth resistance: the larger its value (e.g., 0.999, 0.9999) more iterations are required to use only hard-contrastive samples. For lower values (e.g., 0.5, 0.9, 0.95) such migration will happen in fewer iterations. For fine-tuning, it may be helpful given that we add a linear layer that is initialized randomly.

Recall that margin-based triplet losses such as the contrastive options we described (i.e., $\mathcal{J}_{MAX}, \mathcal{J}_{SUM}, \mathcal{J}_{WE}$) will not propagate gradients when the hinge value is below the margin. Therefore, there is an infinite set of possible weight matrices configurations that minimizes the loss function, which may end up generating underspecification issues [7]. The cross-entropy loss ($\mathcal{J}_{CE}$) will not propagate gradients when the likelihood prediction of the correctly aligned pair is 1 (and the rest are all zeros). In that case, it is also possible that multiple settings of weights allow for the minimal loss value, although arguably in practice that is much harder to happen.

Finally, both types of functions are contrastive given that they compare related pairs to unrelated ones, and therefore the optimization path took by the gradient descent algorithm will depend on the order of the sampled batches. That being said, it is possible that by simply using different initialization seeds, the features that are learned end up differing greatly and present large variability in retrieval quality, generalization capability, and robustness performance. We perform several experiments to help clarifying that hypothesis.

IV. Methodology

A. Hyper-parameters

All models generated in this paper were trained using the same evaluation protocol and hyper-parameter optimization procedures. We use Adam as our main optimization strategy. All models are trained for 20 epochs, and evaluation results are computed 3 times per epoch. We early-stop training if validation MRR plateaus and does not increase during six consecutive validation runs. Our standard regime for controlling the learning rate is starting it at 2×10^{-5} and decreasing it tenfold every 3 epochs. We select the best models to compute test metrics using the validation MRR value as criterion. Batch sizes are defined to fulfill the available GPU memory. In most experiments, we used batches of 85 for CodeBERT models, and 70 for CodeT5, requiring ≈ 24GB of GPU memory.

Those settings differ from CodeBERT and GraphCodeBERT papers in the following aspects: they used batches of 32 and linear learning rate decay schedule applied in all iterations. CodeBERT used initial learning rate of 10^{-5}. Such models are considered our main baselines and we adopt the reported metrics from the respective papers.

For the contrastive loss experiments, we set the margin to 0.2, and the exponential grow resistance to 0 (using hard-contrastives only). We follow CodeBERT experiments and set the default softmax temperature τ to 1. We present the impact of several of those hyper-parameter in ablation studies depicted in Sections V.

V. Experimental Analysis

The main results are obtained after running all architectures for all the different languages available on the dataset. Due to time and computational cost constraints, the ablation studies were performed with fewer programming languages. As it will be discussed in the robustness results, the patterns found do not suffer significant changes over different scenarios.

A. Results

Table IV depicts results for all baseline models in 6 different programming languages. Note that the models we trained consistently outperform the results reported in the original papers. That is evidence that those models were trained sub-optimally. We were able to achieve far better results simply by properly adjusting the batch size, learning rate scheduler, and loss function settings. For the CodeBERT model trained with the cross-entropy loss and batch of 85 instances, we achieved an impressive absolute overall improvement of almost 10%, which is far superior to the gains generated by GraphCodeBert. Also note that we provide code search results for CodeT5 that were not presented in their paper. Those settings seem to be rather important for languages such as Java and PHP, where the newly-trained models provide the largest improvements.

TABLE IV
CODE SEARCH RESULTS IN TERMS OF MRR.

Model	Ruby	Javascript	Go	Python	Java	Php	Overall
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	0.628	0.693
GraphCodeBERT	0.703	0.644	0.897	0.692	0.691	0.649	0.713
CodeT5 [Ours]	0.676	0.679	0.879	0.734	0.761	0.835	0.761
CodeBERT-Contrastive[Ours]	0.727	0.700	0.878	0.675	0.768	0.817	0.761
CodeBERT [Ours]	0.726	0.704	0.892	0.743	0.782	0.834	0.780
GraphCodeBERT [Ours]	0.757	0.728	0.901	0.763	0.781	0.848	0.796

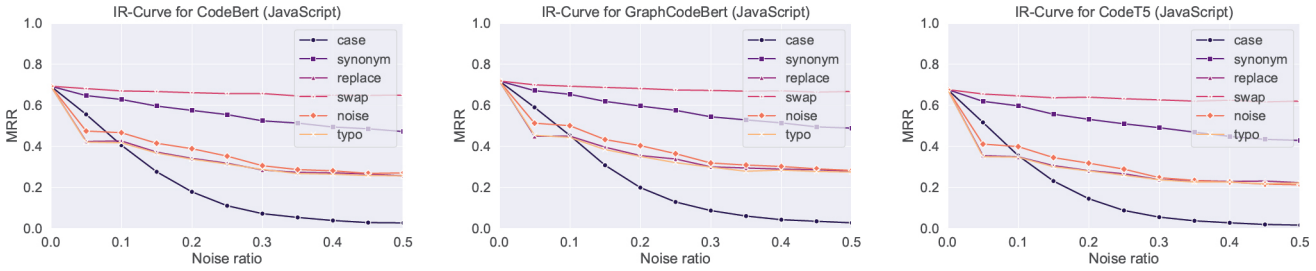

Fig. 1. Input Robustness curves.

B. Robustness Results

Next, we evaluate the models following the input query transformations proposed in the COBE benchmark. By analyzing the results demonstrated in Figure 1, it is possible to see that changes in the **case** of the input characters have the most drastic impact in the performance for all cases. That phenomenon tells us about how code-retrieval models struggle to understand the semantics of the words and the context where the case is relevant or not. Surprisingly, **case** is a much more complex noise factor to the model than transformations that effectively add or change characters.

Differently from the case transformation, we observe that **word-based** transformations are those that cause the smallest impact in terms of MRR. We can see that word order has no relevant impact in query understanding by all models since **swap** barely affects performance. On the other hand, **synonym** is responsible for a small but relevant decay in the results. That happens because even though we are replacing words with others that contain the same semantic function, in many contexts the synonym may not fit, leading the query to a slightly different meaning.

All the character-based transformations have a more significant impact on the quality of the retrieved values. **Noise**, **typo**, and **replace** show similar behavior, causing first an abrupt decrease of performance for small values of noise ratio and then a linear decrease. The small difference between **noise** and the other two is mainly due to the policy followed by each: **noise** adds a character while **replacing** and **typo** replace already existent characters, making it more likely that words will not be completely corrupted for small values of noise ratio in the **noise** transformation.

Despite the differences in MRR, looking only at the behavior from each transformation and their effect in performance, we can see that there is no language or transformer architecture that can increase robustness against modifications over input queries. The patterns also do not change much when modifying the seeds, even with strong differences in the similarity matrices as will be shown in the following ablation studies. Table V shows the IR-AUC results for all transformations in the benchmark.

C. Impact of the loss function

An interesting effect that we discover during the experiments is that cross entropy seems to perform poorly when vectors are normalized. In fact, all models underperformed in that scenario. This same effect did not repeat when using contrastive loss, which performed well in both scenarios, either normalizing vectors or not. Such an effect happens due to the fact that applying a softmax normalization in vectors constrained to $(-1, 1)$ — cosine similarity interval — is redundant. Both operators normalize vectors, and by constraining the input range before softmax, the re-normalization is less effective and will operate in a far more constrained range. This effect can be alleviated by tuning the temperature to sharpen the similarity matrix values differences. Nonetheless,

TABLE V
COBE IR-AUC BENCHMARK RESULTS.

Model	Case	Noise	Question	Replace	Swap	Synonym	Typo	Overall
CodeBERT	0.266	0.415	0.784	0.387	0.765	0.672	0.382	0.524
CodeT5	0.235	0.379	0.756	0.358	0.733	0.630	0.352	0.492
GraphCodeBert	0.279	0.437	0.782	0.408	0.776	0.684	0.402	0.538

contrastive loss seems to be more robust to that, and works for out-of-the-shelf hyper-parameters and normalization regimes.

We observe that each loss function generates very different first order statistics as presented in Figure 2. Interestingly, both presented similar cyclic patterns, which seem to be frequent for contrastive search models. We have found that such effect is most likely to happen in larger datasets, although cross entropy usually tends to converge to lower similarity values.

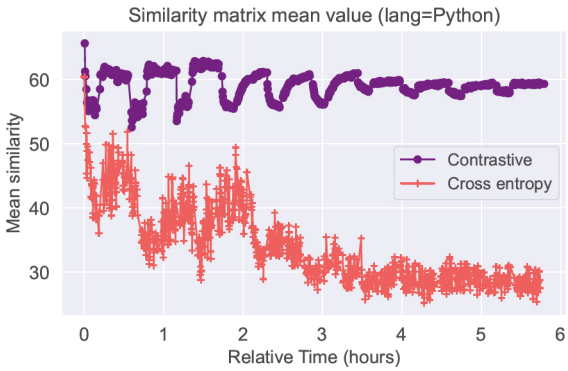

Fig. 2. Mean of the elements in the similarity matrix during training. Values for cross entropy and contrastive loss ($\mathcal{J}_{MAX}$).

D. Batch size importance

Figure 3 shows that the batch size is rather important for achieving top results when using contrastive loss functions. The baseline is a CodeBert model trained with a batch of 32 instances. We compare it to a model trained with 85 instances. The average relative gain in MRR is 2%, though for Python such improvement was near 9%. The average gain is larger than the one obtained by using AST (Asymmetric Syntactic Tree) in GraphCodeBert when compared to the *vanilla* CodeBert.

Such results are not totally unexpected, given that with larger batch sizes during training there is a larger probability of sampling harder samples that may generate more informative gradients. Indeed, with larger batch sizes it is far more likely that the global hard contrastive instance be sampled.

E. Different seeds, different rankings

Considering the fact that all models are trained with somewhat contrastive loss functions, we hypothesized that the order of contrastive samples could largely affect the resulting features — and by consequence the ranked code visible by the future user of the system. We then execute several experiments

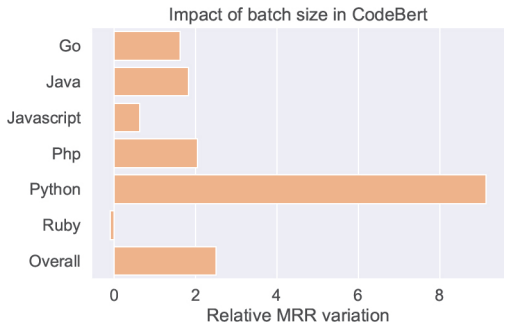

Fig. 3. Relative MRR improvement when training CodeBert with a batch size of 85 over a batch size of 32.

to show how different seeds could affect the learning process, not only in generalization performance and robustness aspects but also in the generated rankings. To test the impact of using different random initialization, we select a contrastive GraphCodeBert and execute the very same architecture on Ruby data ten times only varying the random seed. After training, we execute our COBE benchmark and also compare the rankings generated by each model.

Regarding MRR results, we see in Figure 4 that the models have a certain variability but follow a similar global pattern. During some iterations, the deviation can be as high as 5% in absolute MRR values. When analyzing the robustness scores (IR-AUC) for the different seeds, we start to see some troublesome results. Figure 4 also shows the relative IR-AUC variation of all seeds per transformation type in COBE. With word-based transformations, the distinction across seeds is notably smaller when compared with character-based transformations, where the difference across seeds can reach 14%. This analysis shows that code search models are greatly affected by underspecification, and it is not possible to detect that lack of robustness when only using the standard test set values, showcasing the importance of the proposed benchmark.

In Figure 5 we show that the same models trained with different seeds end up generating very different rankings. To generate that image, we computed all queries in test set considering the top-20 code instances retrieved sorted by similarity. It shows that the Spearman Ranking correlation matrix among all provided rankings vary between 0.14 and 0.16, which indicates a very weak correlation among models. In addition, when we compute the ratio of items that were retrieved together (discarding ranking order), we observe that often only half of the instances are the same, while all the

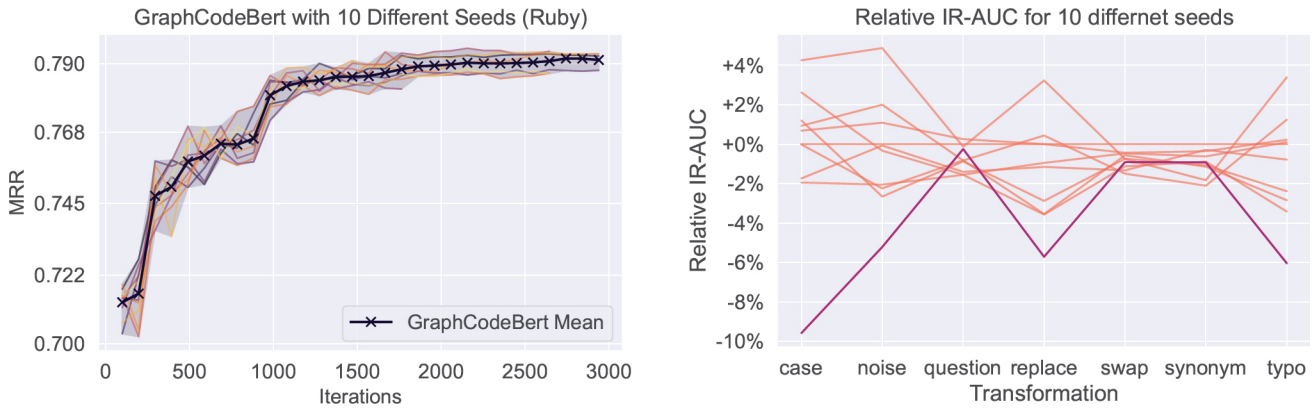

Fig. 4. Left: Validation set MRR values during training. Right: Robustness performance for all runs considering the best model.

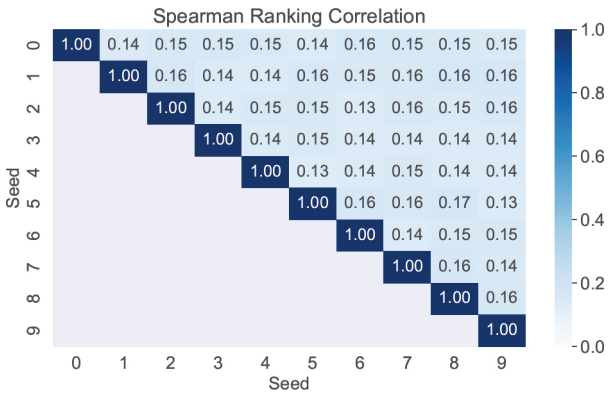

Fig. 5. Ranking correlation between different seed runs.

remaining ones are different. That particular result lead us to the conclusion that even though the models were guided to the same objective, using the same hyper-parameters and configurations, the initialization and order of the sampled batches have a drastic impact over the learning process. This kind of effect imposes credibility challenges for deploying code-retrieval systems in real-world scenarios, and we believe that novel mitigation strategies should be studied for addressing this problem.

VI. Qualitative Analysis

To have a richer understanding of the benchmark behavior, we perform a simple qualitative analysis, which allows us to verify the changes caused by the benchmark transformations over a single instance. For this analysis, we create a toy instance where the code is a simple Python method that reads a json file and the comment is a straightforward description of the function, both presented in Listing 1.

```python
# Source-Code
def read_json(filepath):
    with open(filepath, 'r') as json_file:
        json_list = json.load(json_file)

    return json_list

# Description
# read a json file from a filepath
```

Listing 1. Toy example for the qualitative analysis.

We encode both source code and its respective description with the Python GraphCodeBert model, and we then calculate the similarity between both resulting vectors. Our intention here is to bring a more clear interpretation of the model when functioning over different transformations, i.e., to verify their impact in the computed similarity. We present the similarity between code and descriptions for different ratios of the **noise** transformation in Tables VI.

We can clearly see the overall trend of similarity decreasing as the noise ratio increases. With this particular example, we can notice the lack of robustness of the current state-of-the-art code-retrieval systems, in which simple addition of character noise or case modifications strongly affect the computed similarity between source code and descriptions.

VII. Conclusion

In this paper, we introduced the COBE Benchmark, a complete framework to evaluate the robustness of code retrieval models when dealing with modifications and noise addition in

TABLE VI
Example of COBE in a toy instance for **noise** transformation.

Noise Ratio	Text	Similarity
0.0	read a json file from a filepath	81.60
0.1	riead a json fpile from a filepath	72.00
0.2	yread a json file fGrom a filepath	72.99
0.3	r#ead a json ofile from a filepath	71.52
0.4	read a json gfile 2from a filepath	69.43
0.5	#rea6d a j$soan file from a filepath	57.39

the input queries. We performed a thorough set of experiments to evaluate the current state-of-the-art approaches for the task of Natural Language Code Search (NLCS), under the assumption that the current results do not realistically convey their lack of robustness. Our experiments shows that all models struggle with small character changes, and can suffer from extremely poor performance when dealing with higher ratios of noise, even if the semantic meaning of the code description remains unchanged.

After several ablation studies we confirmed that the current models severely lack in terms of robustness. In fact, we show that the SOTA models for NLCS suffer from underspecification, which means that changes in the random initialization can generate vastly different models, whose snippet rankings are uncorrelated to each other.

Finally, we also show that very simple modifications when training NLCS models can drastically improve results, such as using different loss functions or increasing the batch sizes. We intend to continue investigating the robustness of retrieval models, and in future work we will focus on image-text alignment, in order to verify whether the underspecification problem also affects multimodal foundation models that align images with textual descriptions.

REFERENCES

[1] Maurício Aniche, Erick Maziero, Rafael Durelli, and Vinicius Durelli. The effectiveness of supervised machine learning algorithms in predicting software refactoring, 2020.

[2] Tom et al. Brown. Language models are few-shot learners. In *NeuRIPS*, pages 1877–1901, 2020.

[3] Mark Chen, Alec Radford, Rewon Child, Jeffrey Wu, Heewoo Jun, David Luan, and Ilya Sutskever. Generative pretraining from pixels. In *Proceedings of the 37th International Conference on Machine Learning (ICML 2020)*, pages 1691–1703, 13–18 Jul 2020.

[4] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 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 (NAACL-HLT 2019)*, pages 4171–4186, 2019.

[5] Elizabeth Dinella, Shuvendu K. Lahiri, Todd Mytkowicz, and Gabriel Ryan. Neural unit test suggestions, 2021.

[6] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *ICLR 2021*, 2021.

[7] Alexander D'Amour et al. Underspecification presents challenges for credibility in modern machine learning, 2020.

[8] Mark Chen et al. Evaluating large language models trained on code, 2021.

[9] Rishi Bommasani et al. On the opportunities and risks of foundation models, 2021.

[10] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[11] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

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

[13] Triet H. M. Le, Hao Chen, and Muhammad Ali Babar. Deep learning for source code modeling and generation: Models, applications, and challenges. *ACM Comput. Surv.*, 53(3), jun 2020.

[14] Alexander LeClair, Siyuan Jiang, and Collin McMillan. A neural model for generating natural language summaries of program subroutines. In *Proceedings of the 41st International Conference on Software Engineering (ICSE 2019)*, page 795–806, 2019.

[15] Andy T. Liu, Shu wen Yang, Po-Han Chi, Po-Chun Hsu, and Hung yi Lee. Mockingjay: Unsupervised speech representation learning with deep bidirectional transformer encoders. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP 2020)*, pages 6419–6423, 2020.

[16] Edward Ma. Nlp augmentation. https://github.com/makcedward/nlpaug, 2019.

[17] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision. In *ICML 2021*, volume 139, pages 8748–8763, 2021.

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

[19] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *JMLR*, 21(140):1–67, 2020.

[20] Alexander Rives, Joshua Meier, Tom Sercu, Siddharth Goyal, Zeming Lin, Jason Liu, Demi Guo, Myle Ott, C. Lawrence Zitnick, Jerry Ma, and Rob Fergus. Biological structure and function emerge from scaling unsupervised learning to 250 million protein sequences. *Proceedings of the National Academy of Sciences*, 118(15), 2021.

[21] Daniel Rothchild, Alex Tamkin, Julie Yu, Ujval Misra, and Joseph Gonzalez. C5T5: controllable generation of organic molecules with transformers. *CoRR*, abs/2108.10307, 2021.

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

[23] Zhidong Shen and Si Chen. A survey of automatic software vulnerability detection, program repair, and defect prediction techniques. *Secur. Commun. Networks*, 2020:8858010:1–8858010:16, 2020.

[24] Jianhang Shuai, Ling Xu, Chao Liu, Meng Yan, Xin Xia, and Yan Lei. *Improving Code Search with Co-Attentive Representation Learning*, page 196–207. 2020.

[25] Sebastian Thrun. *Lifelong Learning Algorithms*, pages 181–209. Springer US, Boston, MA, 1998.

[26] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30, 2017.

[27] 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. *34th IEEE/ACM International Conference on Automated Software Engineering (ASE 2019)*, pages 13–25, 2019.

[28] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

[29] Jonatas Wehrmann, Camila Kolling, and Rodrigo C Barros. Adaptive cross-modal embeddings for image-text alignment. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 12313–12320, 2020.

[30] Jonatas Wehrmann, Douglas M Souza, Mauricio A Lopes, and Rodrigo C Barros. Language-agnostic visual-semantic embeddings. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 5804–5813, 2019.

[31] Pengcheng Yin, Graham Neubig, Wen tau Yih, and Sebastian Riedel. TaBERT: Pretraining for joint understanding of textual and tabular data. In *Annual Conference of the Association for Computational Linguistics (ACL 2020)*, 2020.