GenCodeSearchNet: A Benchmark Test Suite for Evaluating Generalization in Programming Language Understanding

Andor Diera
Ulm University,
Germany
andor.diera@uni-ulm.de

Abdelhalim Dahou
GESIS - Institute for Social Sciences,
Germany
abdelhalim.dahou@gesis.org

Lukas Galke
Max Planck Institute for Psycholinguistics,
Netherlands
lukas.galke@mpi.nl

Fabian Karl
Ulm University,
Germany
fabian.karl@uni-ulm.de

Florian Sihler
Ulm University,
Germany
florian.sihler@uni-ulm.de

Ansgar Scherp
Ulm University,
Germany
ansgar.scherp@uni-ulm.de

Abstract

Language models can serve as a valuable tool for software developers to increase productivity. Large generative models can be used for code generation and code completion, while smaller encoder-only models are capable of performing code search tasks using natural language queries. These capabilities are heavily influenced by the quality and diversity of the available training data. Source code datasets used for training usually focus on the most popular languages and testing is mostly conducted on the same distributions, often overlooking low-resource programming languages. Motivated by the NLP generalization taxonomy proposed by Hupkes et. al., we propose a new benchmark dataset called GenCodeSearchNet (GeCS) which builds upon existing natural language code search datasets to systemically evaluate the programming language understanding generalization capabilities of language models. As part of the full dataset, we introduce a new, manually curated subset StatCodeSearch that focuses on R, a popular but so far underrepresented programming language that is often used by researchers outside the field of computer science. For evaluation and comparison, we collect several baseline results using fine-tuned BERT-style models and GPT-style large language models in a zero-shot setting.

1 Introduction

Language models have found their use in various tasks dealing with source code, ranging from code search to code summarization, code completion, and code translation (Lu et al., 2021). With the release of Codex (Chen et al., 2021) and ChatGPT (Ouyang et al., 2022) large language models (LLMs) became popular and widely used for AI-assisted coding. Still, as of August 2023, code completion and code-related question answering with

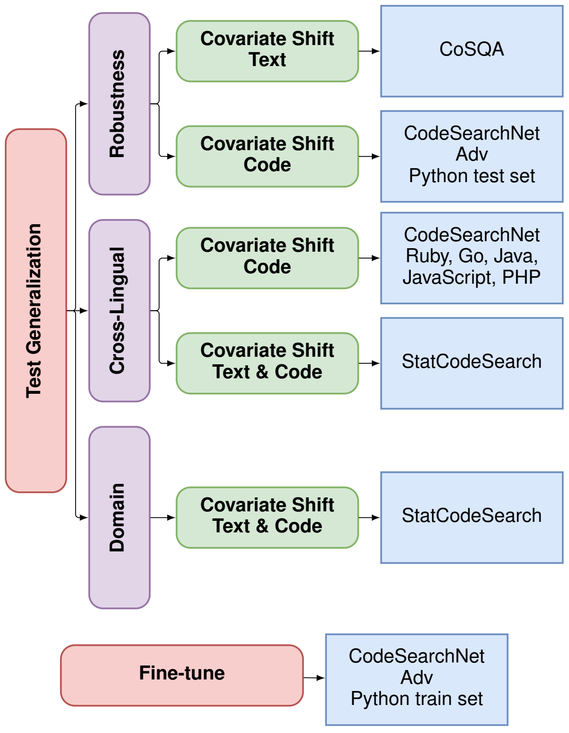
Test Generalization
Robustness
Cross-Lingual
Domain
Covariate Shift Text
Covariate Shift Code
Covariate Shift Code
Covariate Shift Text & Code
Covariate Shift Text & Code
CoSQA
CodeSearchNet Adv Python test set
CodeSearchNet Ruby, Go, Java, JavaScript, PHP
StatCodeSearch
StatCodeSearch
Fine-tune
CodeSearchNet Adv Python train set

Figure 1: Overview of the benchmark composition w.r.t. the generalization taxonomy of Hupkes et al. (2023)

LLMs is far from reliable (Kabir et al., 2023). An alternative to using general purpose LLMs for code completion is code search (i.e., finding relevant source code based on a natural language query) in curated datasets (Cambronero et al., 2019; Husain et al., 2019) which provides a more transparent aid for AI-assisted coding. However, so far it is unclear how well code search models generalize across different programming languages, different domains, and how robust they are against distribution shifts.

Although both natural language queries and source code are represented as text, one cannot safely assume to have an overlap in the words/characters used, since function and variable names do not necessarily consist of words. Still, classical string-based matching between a query and documents (Manning, 2009) is used in many information retrieval systems and practical applications (Lin et al., 2022). When documents consist of code, natural language queries will have little to no matching words, resulting in a lexical gap. Due to this lexical gap, the task of finding code snippets based on natural language queries is difficult and requires specific bimodal language models capable of processing both natural and programming languages (Feng et al., 2020; Guo et al., 2020; Wang et al., 2021, 2023).

However, this bimodal training approach is likely limited to the programming languages on which the models were trained. It is unknown how such models would generalize to programming languages that were not part of the training data or have only little representation in the dataset, i. e., a low-resource programming language. Evaluating the models on programming languages that were not part of the training data (i. e., a distribution shift occurs) would shed new light on the generalization capabilities of hybrid models for code and text – which is the aim of this work.

Moreover, there are low-resource programming languages such as R that in terms of quantity are underrepresented on popular code-sharing platforms like GitHub. However, it is the de facto programming language in many research fields relying on statistical analysis, such as economics, statistics, social sciences, and psychology. Thus, the coding conventions and style in these fields also differ from the code corpora used in existing benchmark datasets (Husain et al., 2019; Lu et al., 2021). This produces a blind spot on the current methods for code search as they are usually only tested on datasets of well-curated source code in the most popular programming languages.

We propose a new benchmark dataset called GenCodeSearchNet (GeCS) which combines a new, manually curated dataset StatCodeSearch with existing code search datasets. StatCodeSearch consists of code-comment pairs extracted from R scripts written for statistical analysis. We further propose an evaluation protocol for the benchmark that allows future researchers to systemically test the language models' generalization capabilities for programming language understanding. The evaluation setup for our dataset is illustrated in Figure 1 and consists of three generalization tests for robustness, cross-lingual, and domain generalization. We provide a detailed description of this benchmark in Section 3. In summary, the contributions of this work are three-fold:

- We create a benchmark for *programming language understanding* named GenCodeSearchNet that tests text-code matching and ranking, organized along different types of out-of-distribution generalization. The composition of the benchmark is described in Section 3 and its evaluation protocol in Section 4.
- To facilitate the new benchmark, we introduce a new, manually-curated dataset named StatCodeSearch, consisting of 1,070 text-code pairs from statistical research code written in R, which is described in Section 3.2.
- Initial baselines for this new benchmark are introduced in Section 5. We provide results for RoBERTA, CodeBERT, CodeT5+, and GPT-based LLMs in Section 5.2.

2 Related Work

Code Search Code search is an established research field with various tools and solutions available. Below, we briefly summarize existing classical works on source code search, followed by works on semantic code search based on neural networks, particularly pre-trained language models.

An established classical tool for source code search is Oracle OpenGrok[1] based on the popular full-text index Lucene. As a result, OpenGrok enables textual searching of code for strings based on Google-like search queries. Similar systems for textual searching on code are searchcode[2] and Sourcegraph[3]. The ANNE (Vinayakarao et al., 2017) system extends the purely textual search to source code by mapping natural language queries to syntactic keywords of programming languages. Search engines supporting structured queries include Aroma (Luan et al., 2019), which takes an incomplete code fragment as a query (called a snippet) and suggests concise code snippets from the code database. A similar approach is also taken by Mukherjee et al. (Mukherjee et al., 2020).

Neural networks have also been successfully applied for code search allowing semantics-based natural language code search. Just as in most fields

[1]https://github.com/oracle/opengrok
[2]https://searchcode.com/
[3]https://about.sourcegraph.com/

of NLP, the best-performing models for semantic code search are transformer-based language models. These models are pretrained on both natural language and programming language corpora and often use a contrastive loss to better align text and code representations (Li et al., 2022b; Wang et al., 2023; Neelakantan et al., 2022). Prominent examples of encoder-only bimodal language models include CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2020), and CodeRetriever (Li et al., 2022a). Encoder-decoder models designed to handle a wide range of programming language tasks such as CodeT5 (Wang et al., 2021) and CodeT5+ (Wang et al., 2023) also perform strongly on the task of semantic code search. Lastly, decoder-only models (Brown et al., 2020) can be also employed for code search, albeit they either require careful prompting (Ouyang et al., 2022) or extensive fine-tuning (Neelakantan et al., 2022).

Generalization in Programming Language Understanding Generalization, or the ability of a model to perform well on data not seen during training, is sought after in all domains of machine learning (Goodfellow et al., 2016). However, generalization can refer to a wide range of different scenarios in NLP. To tackle this lack of agreement and systematic testing, Hupkes et al. (2023) proposed a taxonomy for characterizing generalization research in NLP. Their taxonomy consists of five axes to classify the motivation, generalization type, data shift type, source, and locus of the shift.

Even though neural networks capable of handling both natural and programming languages are usually called bimodal models (Allamanis et al., 2015; Feng et al., 2020; Wang et al., 2023), the representations of these two input modalities share a lot of commonalities, including predictable statistical properties (Hindle et al., 2012). Therefore, we argue that the NLP generalization taxonomy proposed by Hupkes et al. (2023) also offers valuable insights for research in programming language understanding.

Generalization research on code-related tasks is still in short supply. The HumanEval dataset (Chen et al., 2021) used for benchmarking generative models only contains Python source code, while CodeXGLUE (Lu et al., 2021), the comprehensive programming language understanding benchmark suite is designed to evaluate on in-distribution test data. CodeS (Hu et al., 2023) offers an extensive dataset for evaluating against different types of out-of-distribution samples, but only uses two languages (Python and Java) with the singular downstream task of code classification. To the best of our knowledge, the only large-scale benchmarks for evaluating generalization of code-related tasks are CrossCodeBench (Niu et al., 2023) and XLCoST (Zhu et al., 2022). While XLCoST focuses solely on cross-lingual generalization, CrossCodeBench only includes tasks that are formulated in a text-to-text form, leaving out retrieval tasks, such as code search. In contrast to the aforementioned works, our proposed benchmark focuses on the task of natural language code search and offers evaluations against multiple types of distribution shifts.

3 Composition of GenCodeSearchNet

In this section, we describe the datasets used in our benchmark suite. The datasets are chosen and created based on the criterion to evaluate different types of generalization in programming language processing to foster further research in the field.

The GeCS dataset includes one fine-tuning set and eight test sets. It contains three previously proposed datasets, namely CodeSearchNet (Husain et al., 2019), CodeSearchNet AdvTest (Lu et al., 2021), and CoSQA (Huang et al., 2021). We add a novel set that focuses on statistical tests in the programming language R, named StatCodeSearch. Each dataset contains a natural language description (either a code comment or a search engine query) and a source code snippet. The test suite is designed to study generalization from a practical perspective, i. e., to assess programming language understanding in various evaluation scenarios. The source of datashifts between the different test sets is considered to be naturally occurring, with the locus of the shift appearing between the pretraining/finetuning data and test data. The types of generalization covered by the test suite include robustness to covariate shifts and generalization across programming languages and programming domains.

A detailed breakdown of the experimental design in the generalization framework proposed by Hupkes et al. (2023) can be seen in Figure 1. The main characteristics of the different subsets can be found in Table 1. The average number of tokens have been calculated using the pretrained RoBERTa tokenizer sourced from HuggingFace (Wolf et al., 2019). Furthermore, we also measure the covariate shift in texts and codes using the total variation dis-

Table 1: Statistics of the different subsets used in the GenCodeSearchNet test suite. The numbers shown include only the positive (matching) examples

Subset	**# text-code pairs**	**avg # text tokens**	**avg # code tokens**	**total variation distance text**	**total variation distance code**
Fine-tuning set					
CodeSearchNet AdvTest train	251 820	15.97	166.94	0.0	0.0
Test sets					
CodeSearchNet AdvTest test	19 210	16.08	177.64	0.1268	0.5372
CodeSearchNet Go	14 291	27.46	159.08	0.3281	0.6110
CodeSearchNet Java	26 909	29.07	179.63	0.3126	0.5972
CodeSearchNet JavaScript	6 483	21.93	240.21	0.2892	0.5479
CodeSearchNet Ruby	2 279	23.88	138.09	0.2962	0.5610
CodeSearchNet PHP	29 391	14.73	189.00	0.2858	0.5819
CoSQA	10 293	10.42	55.93	0.5322	0.4453
StatCodeSearch	1 070	24.55	134.02	0.5386	0.8032
Combined Test set	109 926	21.01	159.20	0.3386	0.5855

tance (Goldenberg and Webb, 2019). We calculate the total variation distance to the CodeSearchNet Adv train set (used for fine-tuning) on the tokenized samples.

3.1 Existing Datasets

The **CodeSearchNet** dataset (Husain et al., 2019) was introduced as a semantic code search evaluation tool. Since then it has been a staple benchmark dataset for studying the code search capabilities of machine learning models. It encompasses code-comment pairs from six different programming languages: Python, Go, Java, JavaScript, Ruby, and PHP. The full corpus includes 6 million functions scraped from GitHub, with 2 million of those including associated function documentation. Functions less than three lines and documentation shorter than three tokens were removed from the scraped corpus. Duplicate or near duplicate functions were also discarded to control for auto-generated code snippets and copy & paste between GitHub users. For the GeCS test suite, we collected the test sets from the HuggingFace Hub (Lhoest et al., 2021) and discarded the Python subset (since it is included later in the CodeSearchNet AdvTest dataset). We employed no further preprocessing and formatted the data into JSONL files. For each sample, we defined three fields: the *input* field contains the code comment and code snippet separated by a '[CODESPLIT]' token, the *target* field contains the index of the binary labels ('no_match','match'), which are found in the *target_options* field. This dataset is used to measure cross-lingual generalization.

The **CodeSearchNet AdvTest** dataset was developed for the CodeXGLUE benchmark dataset (Lu et al., 2021) by applying further preprocessing steps on the Python subset of the CodeSearchNet dataset. First, all code snippets that could not be parsed into an abstract syntax tree were removed, then documentations with more than 256 tokens were removed alongside samples that contained special tokens such as *"http://"* or *"<img... >"*. Finally, the functions and variables in the code snippets of the test set have been normalized by renaming them to *func* and arg_i, respectively. This normalization of the test set makes it a good fit to test robustness against a covariate shift in the source code. For the inclusion in the GeCS test suite, we sourced both the train and test set from the official CodeXGLUE repository[4]. We employed no further preprocessing and applied the same formatting as described above.

The Code Search And Question Answering (**CoSQA**) corpus (Huang et al., 2021) uses the Python subset of the CodeSearchNet dataset and matches the code snippets with real-world search queries from the Microsoft Bing search engine. The search logs were carefully filtered to only include queries that incorporate the keyword *python* and have none of the predefined keywords that relate to search intents other than code search (e. g., debugging, conceptual queries, tool usage). After this initial rule-based filtering, a fine-tuned CodeBERT encoder (Feng et al., 2020) was used to measure cosine similarity between candidate queries and code snippets. Each code snippet was then matched with the query of the highest similarity. Pairs with a similarity value of less than 0.5 were discarded. Finally, a number of human annotators were instructed to label each code-query pair whether the code snippet answers the query or not. For the GeCS dataset, we use the training set re-

[4]https://github.com/microsoft/CodeXGLUE

trieved from the official CodeXGLUE repository and discard query-code pairs that are labeled as non-matching (we create our own negative samples as described in Section 4). We apply no further preprocessing and use the same formatting as described before.

3.2 New Dataset: StatCodeSearch

The StatCodeSearch dataset is a benchmark test set consisting of code comment pairs extracted from R programming language scripts authored mostly by researchers. The dataset is sourced from the Open Science Framework (OSF)[5]. It includes text and code samples from R projects that pertain to the fields of social science and psychology with a focus on the statistical analysis of research data. These projects are often linked with research articles published in journals such as *Political Communication*, *Behavior Research Methods*, and *Cognitive Science*. R scripts in these domains seldom use branching or explicit looping constructs, as most logic is handled by higher-order functions and R's implicit vectorization. Furthermore, they heavily rely on functions of loaded libraries (Sihler, 2023).

The initial scraping of the OSF website resulted in 11,775 R projects. After discarding projects that did not have any specific permissive software license, the dataset was narrowed down to 2,832 projects, from which the code-comment pairs of the final dataset were extracted. The creation of code-comment pairs involved a three-step procedure. First, we implemented a rule-based extraction, which included the following steps. We removed empty lines and leading spaces from each line. We discarded lines identified as library loading. We detected lines commencing with "#" that include more than one word. If multiple subsequent comment lines were found, we concatenated them into one text item. We categorized lines without the leading "#" symbol as associated code blocks to the preceding text item.

After the extraction of code-comment pairs through these rules, an additional post-processing step was applied where we discarded common comment trailing symbols from the text items such as "#", "-", and "=". This first step resulted in 40,041 pairs of code and comments. In order to filter out irrelevant comments, in the second step we employed GPT 3.5 Turbo to classify the comments into four predefined classes (the prompts used for this subtask can be found in Appendix A.3.2). These classes were defined as statistical tests, statistical modeling, visualization, and data variables. On a small subset of 400 pairs, we experimented with three prompting methods for this task: zero-shot, one-shot, and few-shot. Our pre-experiments showed zero-shot to be the most suitable approach for this filtering, as it produced fewer false-negatives than the other two approaches. This automated filtering resulted in a total of 10,137 code comment pairs. The third step involved the review and evaluation of the remaining selection by the authors. We manually filtered the remaining 10,137 code pairs and removed those with irrelevant comments, or code blocks that did not correspond to the comment. This step served as a critical quality control measure and yielded a total of 1,070 pairs of code and comments. We applied the same formatting as described in Section 3.1.

4 Evaluation Protocol

4.1 Fine-tuning

Fine-tuning is an optional step for evaluating smaller models that did not have a text-code matching objective during their pretraining phase. For a fair comparison, we suggest only using the training set of the CodeSearchNet AdvTest dataset for fine-tuning, which is based on Python, a widely-used general-purpose programming language.

4.2 Measures

We apply two measures for assessing the performance of the models.

Matching We test whether a given text-code pair is a matching pair (positive) or not (negative). We evaluate the accuracy on balanced test sets with an equal number of positive and negative examples. The non-matching examples are sampled uniformly within the respective dataset.

Ranking To evaluate the ranking in the code search task, we employ Mean Reciprocal Rank (MRR). For each query, we consider 99 distractors sampled uniformly at random. For each query, the reciprocal of the best-ranked correct answer is considered. Formally, for a set of queries Q, the reciprocal of the best-ranked correct answer at rank r_i is aggregated and averaged as $\mathrm{MRR} = \frac{1}{|Q|} \sum_{i=1,\ldots,|Q|} \frac{1}{r_i}$.

The rationale for the choice of MRR over alternative ranking metrics, such as mean average preci-

[5]https://osf.io/

sion (MAP) or normalized discounted cumulative gain (nDCG) (Manning, 2009; Lin et al., 2022), is that we have only a single relevant code snippet for each query and the ratings are binary. When there is only a single relevant document, as in CodeSearchNet, MRR and MAP coincide to the same formula. Furthermore, nDCG can reflect different degrees of relevance, but in our case, since we only have a binary assessment of the code-comment pairs, there is no benefit of using this metric.

4.3 Evaluation by Generalization Type

Our proposed benchmark groups the evaluations by the generalization types proposed by Hupkes et al. (2023). For this, we take the unweighted average of the classification and ranking scores across different datasets (see Figure 1). To evaluate **robustness**, we aggregate scores on test sets that exhibit a covariate shift in either text or code. For this, we employ CoSQA and CodeSearchNet AdvTest (as described in Section 3.1). For **cross-lingual generalization** (cross-lingual referring to "across programming languages"), we average the scores across datasets CodeSearchNet and StatCodeSearch. In this generalization type, the locus of the covariate shift is mainly in the code snippets due to the differing syntax. For **domain generalization**, the sole test set is StatCodeSearch from the domain of statistical analysis, which entails a different coding/commenting style (cf. Section 3.2).

5 Baselines

5.1 Baseline Methods

We provide the results for three main types of baseline models using two evaluation strategies. We employ the encoder-only models RoBERTa and CodeBERT, the encoder-decoder model CodeT5+, and GPT-based models GPT 3.5 Turbo and Text-embedding-ada-002. For RoBERTa and CodeBERT we employ an additional fine-tuning step on the CodeSearchNet AdvTest train set. The GPT-based models are evaluated in a zero-shot setup, while CodeT5+ is tested both in the fine-tuning and zero-shot setup.

For fine-tuning, we follow the same sampling procedure that is also in the evaluation of the matching task, i. e., single negative example for matching. To tackle the matching task, we concatenate the query and code to a singular input and train an output layer on binary classification. To tackle the ranking task with fine-tuned models, we concatenate the query with each of the 100 candidate code snippets. Then, we rank all 100 candidate pairs according to the matching score emitted by the model. In the zero-shot ranking setups we create the ranking based on the cosine similarity between text and code inputs. The choice of hyperparameters for fine-tuning can be found in Appendix A.1. Below we briefly describe the existing models that we use as initial baselines for the proposed benchmark.

RoBERTa is an encoder-only transformer model that builds upon the foundations of the BERT model (Devlin et al., 2019). RoBERTa is designed to improve upon some limitations of BERT by optimizing the pretraining process. These improvements include dynamic masking in masked language modeling, larger batch sizes, increased training data size, and a more thorough hyperparameter optimization (Liu et al., 2019). RoBERTa is a commonly used baseline model for numerous NLP tasks, including semantic code search (Feng et al., 2020). Our experiments are based on the pretrained HuggingFace implementation of the *RoBERTa-base* model (Wolf et al., 2019).

CodeBERT is a pretrained transformer model designed for both natural language and programming language tasks (Feng et al., 2020). It uses the RoBERTa-base architecture with a masked language modeling objective for bimodal (natural language and programming language input pairs) training data and replaced token detection for unimodal (only programming language input) training data. Similarly to other BERT-based models, CodeBERT performs best with task-specific fine-tuning, and is a common baseline in programming language understanding tasks such as semantic code search, code summarization, and code-clone detection (Lu et al., 2021). We use the pre-trained CodeBERT model from HuggingFace (Wolf et al., 2019) and follow the original paper's fine-tuning procedure.

CodeT5+ is an encoder-decoder transformer model suited to solve a wide range of code tasks (Wang et al., 2023). This is made possible by employing a mixture of pretraining objectives, including span denoising, contrastive learning, text-code matching, and causal language modeling on both unimodal and bimodal training data. The CodeT5+ model can be be successfully deployed in multiple settings (zero-shot, fine-tuning, and instruction-tuning) and performs very well on over 20 code-related benchmark tasks. We run our experiments with the *codet5p-110m-embedding* model variant sourced from HuggingFace. This version, denoted as *CodeT5+ (encoder only)* in our experiments, includes only the encoder layers of the bimodal model. This makes it suitable to create high quality embeddings with lower computational costs. For the matching evaluation, we extend this model with a binary classifier head (similar to the RoBERTa and CodeBERT setups) and apply fine-tuning. In the ranking evaluation, we use both the fine-tuned version and the base model in a zero-shot setup.

GPT-3.5 Turbo developed by OpenAI as a member of the GPT family (Brown et al., 2020) is specifically designed to understand and generate both natural language and code. GPT-3.5 Turbo is optimized primarily for chat applications but also excels in traditional understanding and completion tasks. The model is also a successor to the OpenAI Codex model (Chen et al., 2021) and has exhibited significant enhancements in code generation, error detection, debugging, and analysis. Like other LLMs, it is built on the transformer architecture, is pre-trained on vast amounts of text and code, and is heavily fine-tuned through human feedback (Ouyang et al., 2022). We employ GPT-3.5 for the binary classification evaluation in a zero shot setup (prompts can be found in Appendix A.3.2).

Text-embedding-ada-002 denoted as Ada 2 in our experiments, is an embedding model released by OpenAI in late 2022. Embedding models are designed to generate vectors of floating point numbers that capture the semantic meaning of the input (Neelakantan et al., 2022). The second generation of the OpenAI Ada model has been created by merging the functionalities of five distinct embedding models related to text search, text similarity, and code search into one unified interface. We employ Text-embedding-ada-002 to calculate the cosine similarity between the embeddings of the input pairs in the ranking evaluation setup.

5.2 Baseline Results

The evaluation results for the GeCS dataset are shown in Table 2. The aggregated results for each generalization type (as described in Section 4.3) are displayed in Table 3. A breakdown of results by programming language within CodeSearchNet can be found in Appendix A.2.

Matching The fine-tuned models achieve on average around 90% accuracy on the matching task, with RoBERTa producing both the lowest (84.41% on CodeSearchNet AdvTest) and highest (99.18% on CodeSearchNet Go) results. The matching results of GPT 3.5 Turbo range from 32.82% (CoSQA) to 62.71% (StatCodeSearch). The highest results across the fine-tuned models were achieved on the CoSQA and CodeSearchNet Go datasets, while the lowest values were seen in the PHP subset of CodeSearchNet. Aggregating by generalization type, we find that CodeT5+ yields the highest scores on Robustness, while CodeBERT yields the highest scores on Cross-Lingual and Domain.

Ranking In the ranking evaluation, the highest MRR ratings are achieved by the Ada 2 model, which consistently places the correct code snippet in the first two ranks on average. The zero-shot CodeT5+ models also attain similarly strong performance. Compared to the zero-shot models, fine-tuned models performed poorly on ranking, achieving the highest MRR scores on the robustness tests (CodeSearchNet AdvTest and CoSQA), while scoring below 0.1 on the cross-lingual and domain tests. The highest results across all models were obtained on the CodeSearchNet Ruby datasets, while the lowest values are seen in the StatCodeSearch dataset. Aggregating by generalization type, we find that all models yield higher scores for Robustness than for Cross-Lingual and Domain.

6 Discussion

Our newly introduced benchmark dataset provides several new insights on the generalization capabilities of pre-trained language models. Compared to existing benchmarks, such as CodeSearchNet and CodexGLUE, it focuses on evaluating different types of out-of-distribution generalization, leading to new insights about existing models.

First, we observe that encoder-only models completely fail at ranking when tested out of distribution (on datasets for which they were not specifically fine-tuned). In general, we find that ranking performance suffers substantially from fine-tuning, suggesting that the models are overfitting to the fine-tuning set, resulting in limited generalization capabilities in all three investigated generalization types.

Table 2: Baseline Results on GenCodeSearchNet

Model	CodeSearchNet Avg		CodeSearchNet AdvTest		CoSQA		StatCodeSearch	
	Acc	MRR	Acc	MRR	Acc	MRR	Acc	MRR
Fine-tuned Models								
RoBERTa	0.9263	0.1054	0.8441	0.3853	0.9596	0.0441	0.8958	0.0557
CodeBERT	0.9056	0.0907	0.8862	0.4191	0.9758	0.1087	0.9607	0.0251
CodeT5+ (encoder only)	0.8734	0.0616	0.9002	0.2767	0.9773	0.0482	0.9056	0.0582
Zero-shot Models								
CodeT5+ (encoder only)	-	0.8198	-	0.8547	-	0.7972	-	0.6311
GPT 3.5 Turbo	0.5882	-	0.5687	-	0.3282	-	0.6271	-
Ada 2	-	0.8852	-	0.8264	-	0.9439	-	0.7945

Table 3: Aggregated Performance for Each Generalization Type

Model	Robustness		Cross-Lingual		Domain		Combined	
	Acc	MRR	Acc	MRR	Acc	MRR	Acc	MRR
RoBERTa	0.9018	0.2147	0.9110	0.0322	0.8958	0.0557	0.9028	0.1008
CodeBERT	0.9310	0.2639	0.9170	0.0579	0.9607	0.0251	0.9362	0.1236
CodeT5+ (encoder only) FT	0.9387	0.1156	0.8895	0.0599	0.9056	0.0582	0.9112	0.0779
CodeT5+ (encoder only) ZS	-	0.8259	-	0.7254	-	0.6311	-	0.7274
GPT 3.5 Turbo	0.4485	-	0.6076	-	0.6271	-	0.5610	-
Ada 2	-	0.8851	-	0.8398		0.7945	-	0.8398

On the other hand, large-scale pre-trained models, especially Ada 2, excel at ranking on all datasets. The dataset on which such zero-shot models yield the lowest performance is the newly introduced StatCodeSearch. A possible explanation is that the other datasets originate from GitHub and likely suffer from contamination, i.e., their test data could be part of the training data of the large language models (Golchin and Surdeanu, 2023).

The low performance (hardly better than chance) of GPT-3.5 Turbo in zero-shot matching is surprising. We cannot exclude that the performance could be increased by providing a few examples in the prompt. We opted for testing its zero-shot capabilities for a fair comparison with the other LLMs, leaving room for future work with more refined prompting strategies.

Reflecting Hupkes et al. (2023)'s generalization framework for code-related tasks has allowed us to better understand how the generalization type affects language model performance on tasks involving both natural and programming language understanding: Overall our results suggest that fine-tuned encoder-only models are strong at matching even in out-of-distribution test sets, while large-scale embedding models are strong at zero-shot ranking. Bimodal language models have been shown to achieve high performances on in-distribution natural language code search (Feng et al., 2020; Wang et al., 2023). Their results on our benchmark indicate shortcomings of existing models when tested against out-of-distribution data. We hope our newly introduced benchmark can facilitate the development of bimodal language models that generalize well beyond the training or fine-tuning distribution.

7 Conclusion

We have introduced a new benchmark called GenCodeSearchNet (GeCS) for testing the generalization capabilities of language models. The benchmark specifically aims at scrutinizing different types of generalization (robustness, cross-lingual, and domain), and evaluates matching and ranking for each type. To test generalization with covariate shifts in both textual descriptions and code snippets, we provide a new, manually curated dataset StatCodeSearch with R code comment pairs harvested from OSF. As initial baselines, we have evaluated RoBERTa, CodeBERT, CodeT5+, GPT 3.5 Turbo, and Ada 2. The baseline results of our benchmark reveal that models that are good at matching are not necessarily good at ranking and vice-versa. Hence, we hope that the new benchmark spurs the development of programming language-agnostic models that are good at both ranking and matching. Moving forward, one could extend GeCS with other low-resource programming languages to further facilitate the systematic evaluation of pre-trained language models against different aspects of generalization. We make our experiments and baseline models available on GitHub[6]. The final dataset is available on Zenodo[7].

[6]https://github.com/drndr/gencodesearchnet
[7]https://doi.org/10.5281/zenodo.8310891

Acknowledgements

This research is co-funded by the CodeInspector project (No. 504226141) of the DFG, German Research Foundation. The authors also acknowledge support by the state of Baden-Württemberg through bwHPC computing infrastructure. We would also like to thank Saurav Karmakar, Marcel Hoffmann, and Nicolas Lell for their insightful comments and reviews on our work.

Limitations

As with most benchmark collections, there is a risk that part of the test data has been present in the pre-training data of a large language model. As the new dataset StatCodeSeach is not based on code harvested from GitHub but from OSF, we assume that current language models did not have access to it in their pre-training data. However, due to the intransparency of the training set of corporate language models, we cannot fully exclude it.

We did not apply Ada 2 to the matching task. Although such embedding models can be also adapted for matching tasks, doing so in a zero-shot fashion is not straightforward as it would require determining a threshold on the (cosine) similarity. We also did not apply GPT-3.5 Turbo to the ranking task because of context size limitations and incurred costs for, e.g., running pair-wise ranking.

Our newly created dataset StatCodeSearch consists only of a single programming language, which is R. There are other tools and programming languages that are used by researchers for statistical analysis. For a more extensive dataset in the domain of statistical research code, StatCodeSearch could be extended with code snippets from SPSS, STATA, SAS, and Python.

Finally, it may seem counterintuitive that we have consulted GPT-3.5 Turbo for filtering, given its performance as a baseline for the matching is subpar. However, in the filtering step, the assessment of the pre-experiments showed that it was useful for discarding comments that were not suitable for the dataset, i.e., producing low numbers of false negatives. After this semi-automated pre-filtering, we manually filtered out the remaining code-comment pairs that were not suited for the dataset.

Ethical Considerations

Large language models come with ethical concerns regarding ethical bias, fairness, and transparency. The purpose of the paper is to introduce a benchmark that aims at increasing our understanding of large language models' capabilities in the application of code search. Therefore, we do not see any new risks introduced by our paper.

Our dataset is derived solely from source code files released under public licenses enabling re-distribution. As described in Husain et al. (2019) the CodeSearchNet dataset was filtered to include GitHub projects only with licenses explicitly permitting re-distribution. The CoSQA and CodeSearchNet AdvTest sets are released under the Computational Use of Data Agreement (C-UDA) on the official CodeXGLUE repository[8]. The newly created dataset StatCodeSearch consists of public content scraped from the Open Science Framework (OSF). All OSF content marked "Public" is available for commercial and non-commercial use according to the terms of use[9]. Additionally we filtered out projects that did not include public licenses explicitly permitting modification and re-distribution. The source code files in the StatCodeSearch dataset include the public licenses *Apache License, MIT License, CC-By Attribution 4.0, BSD 3-Clause, CC0 1.0 Universal, GNU General Public License, GNU Lesser General Public License*, and *Mozilla Public License*.

[8] https://github.com/microsoft/Computational-Use-of-Data-Agreement

[9] https://github.com/CenterForOpenScience/cos.io/blob/master/TERMS_OF_USE.md

References

Miltos Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. 2015. Bimodal modelling of source code and natural language. In *International conference on machine learning*, pages 2123–2132. PMLR.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

José Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When deep learning met code search. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, pages 964–974. ACM.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Jacob Devlin, Chang Ming-Wei, Lee Kenton, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of NAACL-HLT*, pages 4171–4186.

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

Shahriar Golchin and Mihai Surdeanu. 2023. Time travel in llms: Tracing data contamination in large language models. *arXiv preprint arXiv:2308.08493*.

Igor Goldenberg and Geoffrey I Webb. 2019. Survey of distance measures for quantifying concept drift and shift in numeric data. *Knowledge and Information Systems*, 60(2):591–615.

Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. 2016. *Deep Learning*. Adaptive Computation and Machine Learning. MIT Press.

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

Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, page 837–847. IEEE Press.

Qiang Hu, Yuejun Guo, Xiaofei Xie, Maxime Cordy, Mike Papadakis, Lei Ma, and Yves Le Traon. 2023. Codes: towards code model generalization under distribution shift. In *International Conference on Software Engineering (ICSE): New Ideas and Emerging Results (NIER)*.

Junjie Huang, Duyu Tang, Linjun Shou, Ming Gong, Ke Xu, Daxin Jiang, Ming Zhou, and Nan Duan. 2021. Cosqa: 20,000+ web queries for code search and question answering. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 5690–5700.

Dieuwke Hupkes, Mario Giulianelli, Verna Dankers, Mikel Artetxe, Yanai Elazar, Tiago Pimentel, Christos Christodoulopoulos, Karim Lasri, Naomi Saphra, Arabella Sinclair, et al. 2023. A taxonomy and review of generalization research in NLP. *Nature Machine Intelligence*, 5(10):1161–1174.

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

Samia Kabir, David N Udo-Imeh, Bonan Kou, and Tianyi Zhang. 2023. Who answers it better? an in-depth analysis of chatgpt and stack overflow answers to software engineering questions. *arXiv preprint arXiv:2308.02312*.

Quentin Lhoest, Albert Villanova del Moral, Yacine Jernite, Abhishek Thakur, Patrick von Platen, Suraj Patil, Julien Chaumond, Mariama Drame, Julien Plu, Lewis Tunstall, et al. 2021. Datasets: A community library for natural language processing. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 175–184.

Xiaonan Li, Yeyun Gong, Yelong Shen, Xipeng Qiu, Hang Zhang, Bolun Yao, Weizhen Qi, Daxin Jiang, Weizhu Chen, and Nan Duan. 2022a. CodeRetriever: A large scale contrastive pre-training method for code search. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 2898–2910, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Xiaonan Li, Yeyun Gong, Yelong Shen, Xipeng Qiu, Hang Zhang, Bolun Yao, Weizhen Qi, Daxin Jiang, Weizhu Chen, and Nan Duan. 2022b. Coderetriever: Unimodal and bimodal contrastive learning. *arXiv preprint arXiv:2201.10866*.

Jimmy Lin, Rodrigo Nogueira, and Andrew Yates. 2022. *Pretrained transformers for text ranking: Bert and beyond*. Springer Nature.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019.

RoBERTa: A robustly optimized BERT pretraining approach. *arXiv preprint arXiv:1907.11692*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. 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)*.

Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. 2019. Aroma: code recommendation via structural code search. *Proc. ACM Program. Lang.*, 3(OOPSLA):152:1–152:28.

Christopher D Manning. 2009. *An introduction to information retrieval*. Cambridge university press.

Rohan Mukherjee, Chris Jermaine, and Swarat Chaudhuri. 2020. Searching a database of source codes using contextualized code search. *Proc. VLDB Endow.*, 13(10):1765–1778.

Arvind Neelakantan, Tao Xu, Raul Puri, Alec Radford, Jesse Michael Han, Jerry Tworek, Qiming Yuan, Nikolas Tezak, Jong Wook Kim, Chris Hallacy, et al. 2022. Text and code embeddings by contrastive pre-training. *arXiv preprint arXiv:2201.10005*.

Changan Niu, Chuanyi Li, Vincent Ng, and Bin Luo. 2023. Crosscodebench: Benchmarking cross-task generalization of source code models. *arXiv preprint arXiv:2302.04030*.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744.

Florian Sihler. 2023. Constructing a static program slicer for R programs. Master's thesis, Ulm University.

Venkatesh Vinayakarao, Anita Sarma, Rahul Purandare, Shuktika Jain, and Saumya Jain. 2017. ANNE: improving source code search using entity retrieval approach. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining, WSDM 2017, Cambridge, United Kingdom, February 6-10, 2017*, pages 211–220. ACM.

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

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

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2019. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*.

Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K Reddy. 2022. Xlcost: A benchmark dataset for cross-lingual code intelligence. *arXiv preprint arXiv:2206.08474*.

A Appendix

A.1 Hyperparameter choice

For RoBERTa, we adopt best practices with a learning rate of 2×10^{-5}, 5 epochs, batch size of 32, weight decay of 0.01, and sequence length of 512. For CodeBERT, we follow the original paper's parameters, while adjusting the sequence length to 512 tokens for consistent comparisons. This involves a learning rate of 1×10^{-5}, a batch size of 32, and 8 epochs. Similarly, CodeT5+ adheres to its original parameters, but with a sequence length of 512 tokens and a batch size of 32 to ensure fairness.

Table 4: Hyperparameter settings for our language models

Hyperparameter	RoBERTa	CodeBERT	CodeT5+
Learning rate	2×10^{-5}	1×10^{-5}	2×10^{-5}
Epochs	5	8	10
Batch size	32	32	32
Weight decay	0.01	0.01	0.01
Sequence length	512	512	512

A.2 Breakdown of CodeSearchNet Results

Table 5 shows the breakdown of the CodeSearchNet by programming language. The numbers for in-distribution fine-tuning (fine-tuning for each individual language before testing) were taken from the original CodeBERT paper (Feng et al., 2020), where for each sample 999 distractors were chosen. Although this makes the comparison to our numbers (with only 99 distractors) unfair, we can still observe a substantial decrease in performance when a model is fine-tuned on out-of-distribution data.

A.3 GPT 3.5 Turbo Prompts

In the below sections, we report the prompts used for categorization and matching tasks for each of the test sets. GPT-3.5 was employed with default parameters, except for limit of 10 output tokens.

Table 5: Breakdown of the CodeSearchNet Results by Programming Language

Model	**Ruby**		**Go**		**PHP**		**Java**		**JavaScript**	
	Acc	MRR	Acc	MRR	Acc	MRR	Acc	MRR	Acc	MRR
Fine-tuned Models iid										
RoBERTa (Feng et al., 2020)	-	0.6245	-	0.6809	-	0.6576	-	0.6659	-	0.6060
CodeBERT (Feng et al., 2020)	-	0.6926	-	0.8400	-	0.7062	-	0.7484	-	0.7059
Fine-tuned Models ood										
RoBERTa	92.71	0.1551	99.18	0.0469	89.73	0.0917	90.69	0.1228	90.85	0.1109
CodeBERT	89.71	0.1451	94.55	0.0668	87.21	0.0742	91.24	0.0822	90.10	0.0850
CodeT5+ (encoder only)	87.47	0.1003	91.86	0.0158	85.03	0.0568	86.04	0.0594	86.30	0.0756
Zero-shot Models										
CodeT5+ (encoder only)	-	0.8398	-	0.8467	-	0.8000	-	0.7939	-	0.8185
GPT 3.5 Turbo	58.70	-	59.50	-	56.41	-	61.30	-	58.19	-
Ada 2	-	0.8942	-	0.9093	-	0.8684	-	0.8761	-	0.8782

A.3.1 Zero-shot code comment categorization

Task: Classify the code comment {Input} based on the categories provided below. If the comment doesn't fit into any of these categories, label it as 'No Relevant Class'. Categories: [Statistical Test], [Statistical Modeling], [Data Variable], [Visualization]

A.3.2 Matching

StatCodeSearch, CodeSearchNet and CodeSearchNet AdvTest

Given a code comment and a {Add the programming language name} programming language code snippet, determine if the comment accurately represents the code's function. Respond with 'True' if the code matches the comment and 'False' if it does not. The input format is defined as "comment" "[CODESPLIT]" "code". {Input}

CoSQA

Given a search query and a Python programming language code snippet, determine if the query accurately represents the code's function. Respond with 'True' if the code matches the query and 'False' if it does not. The input format is defined as "query" "[CODESPLIT]" "code". {Input}

A.4 GenBench Evaluation Card

In Table 6, we provide the evaluation card proposed by (Hupkes et al., 2023) for our experimental setups showcasing the different aspects of generalization our dataset studies.

Motivation			
Practical	*Cognitive*	*Intrinsic*	*Fairness*
☐ △ ◯			

Generalisation type					
Compositional	*Structural*	*Cross Task*	*Cross Language*	*Cross Domain*	*Robustness*
			☐ △	△	◯

Shift type			
Covariate	*Label*	*Full*	*Assumed*
☐ △ ◯			

Shift source			
Naturally occuring	*Partitioned natural*	*Generated shift*	*Fully generated*
☐ △ ◯			

Shift locus			
Train–test	*Finetune train–test*	*Pretrain–train*	*Pretrain–test*
	☐ △ ◯		☐ △ ◯

Table 6: GenBench Evaluation Card:
☐CodeSearchNet
△StatCodeSearch
◯CodeSearchNet Adv & CoSQA