

**MASTER OF SCIENCE
IN
COMPUTER SCIENCE AND APPLICATION**

**DEVELOPMENT OF A CODE SEARCH ENGINE USING
NATURAL LANGUAGE PROCESSING TECHNIQUE**

by

Mohammad Abdullah Matin Khan

Supervisor

Dr. Md. Moniruzzaman

Assistant Professor

Department of Computer Science and Engineering

Islamic University of Technology



Department of Computer Science and Engineering

Islamic University of Technology (IUT)

Board Bazar, Gazipur-1704, Bangladesh.

December, 2023.

CERTIFICATE OF APPROVAL

The thesis titled “**Development of a Code Search Engine Using Natural Language Processing Technique**” submitted by Mr. Mohammad Abdullah Matin Khan, Student ID No. 161042003 of Academic Year 2016-17 has been found as satisfactory and accepted as partial fulfillment of the requirement for the degree Master of Engineering in Computer Science and Application on December 20, 2023.

Board of Examiners:



Dr. Md. Moniruzzaman
Assistant Professor
Department of Computer Science and Engineering,
Islamic University of Technology (IUT), Gazipur.

Chairman
(Supervisor)



Dr. Abu Raihan Mostofa Kamal
Professor and Head
Department of Computer Science and Engineering,
Islamic University of Technology (IUT), Gazipur.

Member
(Ex-Officio)



Dr. Md. Hasanul Kabir
Professor
Department of Computer Science and Engineering,
Islamic University of Technology (IUT), Gazipur.

Member
(Internal)



Dr. Md. Mamun-Or-Rashid
Professor
Department of Computer Science and Engineering,
University of Dhaka, Bangladesh.

Member
(External)

Declaration of Candidate

This is to certify that the work presented in this thesis is the outcome of the analysis and experiments carried out by **Mohammad Abdullah Matin Khan** under the supervision of **Dr. Md. Moniruzzaman**, Assistant Professor, Department of Computer Science and Engineering (CSE), Islamic University of Technology (IUT), Dhaka, Bangladesh. It is also declared that neither this thesis nor any part of it has been submitted anywhere else for any degree or diploma. Information derived from the published and unpublished work of others have been acknowledged in the text and a list of references is given.



Dr. Md. Moniruzzaman
Assistant Professor
Department of CSE
Islamic University of Technology (IUT)
Date: December 20, 2023.



Mohammad Abdullah Matin Khan
Student No.: 161042003
Date: December 20, 2023.

Abstract

The advent of large-scale pre-trained language models has revolutionized the field of natural language processing, enabling significant advancements in various applications, including code retrieval systems. This report presents a novel approach to code retrieval using the Dense Passage Retrieval (DPR) technique that captures the functional similarity between codes as a measure of relevance. DPR is a state-of-the-art method that combines the power of pre-trained language models with dense vector representations for efficient and accurate information retrieval. The objective of this research project is to develop a large scale multimodal, multilingual dataset and leverage the DPR framework to build a code retrieval system capable of retrieving functionally relevant codes given a source code or natural language description of the code in query. To accomplish this, the study first establishes a comprehensive dataset XCODEEVAL comprising large number of source codes downloaded from competitive programming platforms. The dataset is used to train a DPR model, employing a training process that involves large scale pre-trained masked language models called *CodeBERT*, *Starencoder* to learn contextual representations of codes that will facilitate the retrieval of similar codes given a query code. Experimental evaluation is conducted to assess the effectiveness of the proposed code retrieval system. The evaluation includes metrics such as accuracy@ k . The results demonstrate that the DPR-based code retrieval system achieves notable performance gains compared to traditional information retrieval methods. The system effectively retrieves relevant code snippets for a wide range of code queries, highlighting its potential in facilitating retrieval augmented generation models, code reuse, software development, and programming education. Furthermore, the report investigates the impact of different factors, such as multilingual accuracy and batch size on the retrieval performance. Additionally, it explores the limitations and challenges associated with the proposed system, including the scalability of training and deployment, as well as potential biases in the training data. In conclusion, this report presents a comprehensive study on building a code retrieval system using the DPR framework. The experiments for code-code retrieval suggest that albeit retrieval performance after training the base models gets boosted in all cases, monolingual retrieval with functional similarity is very accurate (>80% for accuracy@100) and the multilingual retrieval is bit poor (>56% for accuracy@100). For NL-code retrieval above 80% accuracy is observed for all languages except *D*. The results demonstrate the effectiveness of DPR in leveraging pre-trained language models to improve code retrieval performance. The findings of this research contribute to the advancement of code search and retrieval techniques, opening up new possibilities for efficient code reuse and software development practices.

Acknowledgements

I would like to offer my heartfelt gratitude and appreciation to everyone who helped me finish my MSc thesis. Throughout this process, their advice, support, and encouragement have been priceless.

First and foremost, I am grateful to Almighty Allah for giving me the courage, endurance, and drive to embark on and successfully accomplish this research. I could not have conquered the obstacles and reached this critical milestone in my academic career without His blessings and assistance.

My deepest gratitude goes to my parents for their enduring love, consistent encouragement, and unflinching support throughout my academic endeavors. Their faith in me and their sacrifices have shaped my intellectual and personal development. Their unfailing support has provided me with ongoing drive and inspiration.

I'd like to thank my thesis supervisor, Dr. Md. Moniruzzaman, Assistant Professor, for his great advice, competence, and patience during the whole thesis process. His unique ideas, critical critiques, and consistent support have helped shape the direction and quality of my study. I am grateful for his guidance and encouragement to do my best.

I would also want to thank the department's faculty members for their devotion to education and commitment to fostering students' intellectual progress. Their vast knowledge, passion, and eagerness to share their knowledge have tremendously enhanced my studying experience. I am grateful for their assistance and the useful insights they have offered me during my academic path.

I would like to acknowledge the support from Dr. M Saiful Bari who recently completed his doctorate from Nanyang Technological University (NTU), who is also an alumni of IUT for providing computing resources and insights for the development and publishing of XCODEEVAL.

Finally, I'd want to convey my gratitude to my friends and classmates for their unfailing support, encouragement, and understanding. Their company and camaraderie made the trip more fun and memorable. Their contributions, no matter how large or small, have had a substantial impact on my research and personal development.

Dedicated to my parents

Contents

Certificate	ii
Declaration	iii
Abstract	iv
Acknowledgements	v
List of Tables	x
List of Figures	xi
1 Introduction	1
1.1 Motivation	3
1.2 Problem Statement	6
1.3 Project Contributions	6
1.4 Outline	6
2 Related Works	8
2.1 Retrieval Systems	8
2.2 Code Understanding	10
2.3 Code Generation	11
2.4 Code Datasets	12
3 System Architecture	14
3.1 Web scraper: building dataset	15
3.2 Indexer	17
3.3 Ranker	18
3.4 Server	19
4 Executability	21
4.1 Functional Similarity	22
4.2 ExecEval	23
4.2.1 Architecture	23

4.2.2	Execution Outcome	24
5	Dataset Preparation	26
5.1	Data Representation and Relations	26
5.2	Overview of Scraping Architecture	28
5.3	Statistics of Raw Data	30
5.4	Data Cleanup and Processing	52
5.4.1	Validation-Test Split Creation	54
5.4.2	Data Selection for Validation and Test	55
5.4.3	Control Variable Selection	57
5.4.3.1	Search Techniques	58
5.4.4	Results	58
5.5	Tasks in xCODEEVALwith Statistics	60
5.5.1	Classification Tasks	60
5.5.1.1	Tag Classification	60
5.5.1.2	Code Compilation	62
5.5.2	Generative Tasks	62
5.5.2.1	Program Synthesis	62
5.5.2.2	Automatic Program Repair (APR)	63
5.5.2.3	Code Translation	64
5.5.3	Code Retrieval	64
5.5.3.1	NL-Code Retrieval	65
5.5.3.2	Code-Code Retrieval	65
6	Model Training and Results	68
6.1	Model Training	68
6.1.1	Model Architecture	68
6.1.2	Training Dataset	69
6.1.3	Data Preparation	70
6.2	Indexing the Corpus	72
6.3	Model Evaluation	72
7	Code Search Engine	80
7.1	User Interface	81
7.1.1	Query Input	81
7.1.2	Retrieval Metrics	85
7.1.3	Retrieved Results Display	86

7.2	Backend	88
7.2.1	Initialization	89
7.2.2	Retrieval	89
7.2.3	Benchmark	89
8	Conclusion	90
8.1	Executability	90
8.2	xCODEEVALand ExecEval	91
8.3	DPR Model	92
8.4	Code Search Engine	93
8.5	Future Works	93
	Bibliography	95
A	Resources	108

List of Tables

1.1	Comparison of different execution based evaluation benchmarks.	5
2.1	Comparison between xCODEEVAL and other benchmarks and datasets. . .	13
4.1	Supported languages and their versions in ExecEval.	24
5.1	Statistics of basic quantities of the downloaded raw data.	31
5.2	Comparison of uniformity of data selection between circulation (proposed) and random selection.	59
5.3	Size of the datasets for each task and the evaluation metrics.	60
5.4	Dataset statistics per language and task.	61
5.5	Statistics of retrieval datasets for both <i>nl-code</i> and <i>code-code</i>	66
6.1	List of major hyperparameters used in training.	71
6.2	Summary of performance of both <i>CodeBERT</i> and <i>Starencoder</i> in both tasks.	73
7.1	Request/second benchmark of retrieval backend.	89
A.1	Links to resources made public.	108

List of Figures

3.1	Functional block diagram of the search engine.	15
3.2	Principle of database indexing.	16
5.1	An example of a problem along with a correct submission, and a wrong submission.	27
5.2	Minimal ER diagram of Codeforces database.	28
5.3	Architecture of the scraper.	29
5.4	Distribution of number of submissions across all language compiler versions.	32
5.5	Distribution of number of submissions across all major languages.	33
5.6	Distribution of number of pure submissions across all major language compiler versions.	34
5.7	Distribution of number of mixed submissions across all tags.	36
5.8	Distribution of number of pure submissions across all tags.	37
5.9	Distribution of number of problems across all tags.	38
5.10	Distribution of number of problems across all difficulty rating.	40
5.11	Distribution of number of mixed submissions across all difficulty rating.	41
5.12	Distribution of number of submissions across all verdicts.	43
5.13	Distribution of number of submissions across all verdicts for C++.	44
5.14	Distribution of number of timeline across timeline length.	45
5.15	Distribution of number of mixed submissions across number of token.	47
5.16	Cumulative distribution of number of mixed submissions across number of token.	48
5.17	Distribution of number of translation pairs across pairs of languages.	50
5.18	Distribution of number of translation pairs across pairs of languages.	51
5.19	Flow network of for validation-test dataset creation.	56
5.20	Tag distribution in <i>tag classification</i> task in XCODEEVAL.	61
5.21	Distribution of samples in <i>code translation</i> task.	64
5.22	A code retrieval example.	65

6.1	Comparison of performance between batch sizes.	73
6.2	Comparison of performance between different epochs.	74
6.3	Comparison of top-1 accuracy across all language pairs for <i>CodeBERT</i> . . .	75
6.4	Comparison of top-10 accuracy across all language pairs for <i>CodeBERT</i> . . .	76
6.5	Comparison of top-100 accuracy across all language pairs for <i>CodeBERT</i> . .	77
6.6	Comparison of top-100 accuracy across all language pairs for <i>Starencoder</i> . .	78
7.1	The query input section of UI.	81
7.2	The query input filled up with an example Rust code.	82
7.3	Inputs to generate a metric report on the retrieved results.	83
7.4	Inputs with example data to generate a metric report on the retrieved results.	84
7.5	Retrieval metrics in UI.	84
7.6	Retrieved result metadata display without enabling search metrics.	84
7.8	Retrieved code displayed upon clicking on the result metadata display. . . .	85
7.7	Retrieved result metadata display with search metrics enabled.	85
7.9	Lighthouse score of the UI,	87

Chapter 1

Introduction

In the rapidly evolving landscape of software development, the importance of efficient and effective code search cannot be overstated. The code search engine, at its core, serves as a versatile tool designed to locate pertinent code snippets based on user queries. What defines "relevance" in this context can vary widely, ranging from syntactical similarity to semantic equivalence, depending on the specific needs of the user [13, 77, 39]. The engine operates within a vast database housing an extensive collection of code, forming the backbone from which search and retrieval operations draw their results. The indispensability of tool is reflected through the code search engines provided by different version control service platforms such as Github, Gitlab, Bitbucket etc where one can search for semantically similar codes from the repository. Google Code Search¹ also provide a semantic code search tool with multitude of controls (e.g. regular experssion, class name, file name, ignoring comments etc.).

The inherent generality of the code search engine opens the door to a myriad of applications that extend far beyond conventional search functionalities. One of its notable applications is its role as a plagiarism detector [50], capable of identifying instances where code similarities might indicate unauthorized reproduction. This not only aids in maintaining the integrity of software projects but also serves as a safeguard against intellectual property violations. Code search as a backbone for plagiarism detection is offered by Code Plagiarism Checker², Codequiry³, Copyleaks⁴. Additionally, the engine can be harnessed for Retrieval-Augmented Generation (RAG), offering developers a valuable resource to enhance their coding practices through intelligent suggestions and

¹<https://developers.google.com/code-search/>

²<https://www.codeplagiarismchecker.com/>

³<https://codequiry.com/>

⁴<https://copyleaks.com/codeleaks/code-plagiarism-checker>

contextually relevant examples [51, 45]. Bing Chat⁵, Google Bard⁶ are two of the most popular RAG tools available publicly.

Moreover, the code search engine proves invaluable in the realm of copyright infringement detection, acting as a vigilant guardian against unauthorized use of proprietary code. The Hive AI offer one such copyright search tool⁷. The code search engine's capabilities extend to code completion and reuse tools, empowering developers with the ability to expedite their coding processes by leveraging existing solutions (e.g. tabnine⁸). The engine also plays a pivotal role as a coding assistant, offering guidance and support to programmers, whether they are novices seeking to learn or experienced developers aiming to streamline their workflows (e.g. Github Copilot⁹). As we delve deeper into this report, we will explore the intricacies of the code search engine, examining its architecture, functionalities, and the diverse range of applications it facilitates.

Central to the functionality of any search engine lies its retrieval system, a sophisticated component that embodies the principles of information retrieval (IR). The IR techniques employed by this system form the backbone of the engine's ability to process queries and yield relevant search results from its expansive code database. Information retrieval, in the context of code search engines, encompasses the art and science of efficiently and accurately retrieving relevant code snippets based on user-defined criteria [17, 76, 8, 70, 38].

The search engine itself acts as a streamlined interface, a thin wrapper meticulously crafted to enhance the interaction between clients and the underlying retrieval system. This wrapper serves a dual purpose: first, it ensures the consistency of user interactions with the system, providing a standardized experience irrespective of the complexities within the retrieval engine. This uniformity proves paramount for users seeking reliability and predictability in their interactions with the search engine [38].

Secondly, the wrapper plays a crucial role in fortifying the system against faults and errors, rendering the overall user experience resilient and dependable. Fault tolerance is a critical aspect, especially in the dynamic and often unpredictable environment of software

⁵<https://www.bing.com/search>

⁶<https://bard.google.com/>

⁷<https://thehive.ai/apis/copyright-search>

⁸<https://www.tabnine.com/>

⁹<https://github.com/features/copilot>

development. By encapsulating the intricacies of the retrieval system, the search engine shields users from potential disruptions, ensuring a seamless and uninterrupted search process.

Beyond these fundamental functionalities, the search engine's wrapper introduces additional features to enhance ease of use, catering to the unique demands of various use cases. These features are tailored to the specific needs of developers, whether they are navigating the engine for code completion, plagiarism detection, or any of the myriad applications the code search engine supports. As we delve further into this report, we will dissect the inner workings of both the retrieval system and the search engine wrapper, unraveling the complexities that contribute to the efficiency and versatility of this indispensable tool in the realm of software development [39, 38].

1.1 Motivation

Software development relies heavily on the reuse and adaptation of existing code snippets, libraries, and frameworks. Programmers regularly face situations where they need to locate code examples, API documentation, or open-source projects to expedite their development process. In today's software development landscape, programmers often encounter this task of searching for relevant code snippets, libraries, or documentation to solve coding challenges efficiently as rather daunting [17, 38, 39, 8, 70, 50]. Currently GitHub Copilot, tabnine etc. tools exist as an extension to the preferred IDE or editor the developers use. These extensions use generative models as backend to provide the code snippets whereas sometimes a google search is preferred which by definition is a retrieval system as it queries a database for relevant documents. Traditional retrieval systems, while useful for general information retrieval, often fall short when it comes to retrieving accurate and contextually relevant code-related results as it caters to general web searches and often produce an overwhelming number of irrelevant or outdated results when used to search for code-related information. [13] This limitation hampers the productivity and efficiency of developers.

At the end of 2022, OpenAI published chatGPT in their official website which is a generative text model which can help people as search engines and even produce code snippets to help developers. That being said generative models are predicting next token/word of its generated text based on their previous tokens/words it generated. Goldstein

et al. [21], Borji [7] showed that is there is no real referencing within the document and the generated texts may well be false or incorrect. It is worth mentioning that Retrieval Augmented Generative (RAG) models are one of the most promising frontier of current NLP research. It works in the following way:

1. There are two models under the RAG model hood, namely a dense passage retriever and an autoregressive generative language model. The retriever model takes the input query (such as a question or a prompt) and encodes it into a dense vector representation.
2. The retriever model then uses maximum inner product search (MIPS) to find the top-K documents from Wikipedia that have the highest similarity with the query vector.
3. The retrieved documents are passed to the generative model as additional context, along with the input query.
4. The generative model generates an output sequence (such as an answer or a text) based on the input query and the retrieved documents.
5. The generative model can either condition on the same retrieved documents across the whole output sequence, or use different documents for each output token. This is controlled by a RAG formulation parameter. [44]

The pursuit of automating the generation of computer programs to tackle intricate challenges stands as a longstanding aspiration within the realm of Artificial Intelligence (AI) [54]. Recent years have marked a significant juncture, particularly with the burgeoning prominence of Large Language Models (LLMs), wherein remarkable strides have been witnessed in synthesizing code. Notably, this synthesized code not only remains pertinent but also boasts full functionality without necessitating further human intervention [11].

The advancements achieved in corollary domains such as program synthesis [14, 47], program repair [6], code translation [73, 74], and code retrieval [86, 65] exert a profound influence. They not only significantly enhance developer productivity [102] but also furnish invaluable support to educators [19]. These developments signify a transformative phase wherein AI innovations centered on code generation are poised to reshape the landscape of software development and educational methodologies alike.

In spite of the anticipated ubiquity of such technological advancements, it is noteworthy that their comprehensive evaluation remains a challenging endeavor. The assessment

of these advancements has, more often than not, been conducted in a disparate manner, primarily confined to a limited spectrum of programming languages, such as *Python* and *Java*. This evaluation has been further constrained by an incomplete granularity level, often confined to the realm of individual statements [29] or functions [30].

Furthermore, the existing evaluative efforts tend to be narrowly focused, concentrating on specific tasks such as program synthesis and translation. Notably, these evaluations frequently lack the requisite fine-tuning data [5] or rely on simplistic metrics, such as lexical n -gram based relevance, rather than capturing the essence of actual execution dynamics [31]. The inadequacy of such evaluation methodologies raises concerns regarding the comprehensive understanding and validation of the advancements under consideration.

In response to these challenges, we contribute to the discourse by providing a succinct analysis of the prevailing characteristics inherent in extant program evaluation test-beds. Notably, our focus encompasses unit-level evaluations, as delineated in table 1.1. The table compares the total number of unit test cases provided with the benchmarks. Here ∞ means automated unit test generation by EvoSuite. N/A refers to unit tests not openly available. For our retrieval tasks, each candidate is pre-evaluated against the test cases. More overarching dataset comparisons are presented in section 2.4. This analytical framework serves to illuminate the existing gaps and variations in the evaluation landscape, offering valuable insights for the refinement and advancement of future evaluations in the field of program generation, retrieval and related endeavors.

Table 1.1: Comparison of different execution based evaluation benchmarks.

Benchmark	Lal	Unit Test
TransCoder [73]	3	14,100
HumanEval [11]	1	1,325
HumanEval-x [83]	9	840
MBPP [5]	1	1,500
TransCoder-ST [74]	3	∞
APPS [26]	1	22,711
MBXP [4]	10	1,500
CodeContests [47]	3	27,220*
XCODEEVAL (ours)		
– Classification tasks	11	-
– Generation tasks	11	62,798
– Retrieval tasks	17	62,798

Another important aspect to consider is the development of different neural retrievers following the year 2017 after the mass adoption of transformer architecture [85]. Different Bidirectional Encoder Representations from Transformer (BERT) [16, 18, 46] based models were introduced which contributed to the neural retrievers in representing text in a meaningful way from unstructured text. This allowed for end-to-end solutions for retrieval systems as no text processing is required for it to work [22, 39, 8, 76, 36]. This directly motivates to create an appropriate dataset with annotations for executability and functionality and develop better language models with semantic understanding of code which can then be used to build retrieval systems and generative models with superior code understanding.

1.2 Problem Statement

The purpose of the project is to create a large parallel corpus of programming language data from online competitive programming websites, train a BERT based model to learn embedding with retrieval and develop a scalable code retrieval system with high degree of semantic understanding.

1.3 Project Contributions

1. Develop the frontend and backend server of a prototype code search engine.
2. Implement a scraper with lot of fail safe mechanism and download data from Codeforces. Prepared a large dataset of ~ 25 M codes from this downloaded data called XCODEEVAL which provides train, validation, and test datasets for 7 Code-Code, NL-Code tasks.
3. Train BERT based dense passage retrievers with functional similarity annotated dataset and evaluate with top- k accuracy.
4. Design a network flow based data selection technique.
5. Develop a distributed, extensible, secure solution for evaluating machine generated code with unit tests in multiple programming languages

1.4 Outline

We discuss relevant background studies in chapter 2 starting with some history on information retrieval and how neural retrievers became the state-of-the-art way to develop retrieval

systems. Furthermore we discuss why a new dataset is needed for making any significant progress in coding related tasks to be solved by language models. Chapter 3 describes different components of the code search engine, briefly mention how the components are developed with elaborate justification of the choices we made for our cause. Chapter 4 is an walk through of the execution-based evaluation we eluded in the introduction along with the discussion on *functional similarity* which is the core concept defining relevance for code retrieval. Then chapter 5 show how the dataset is prepared in details, chapter 6 show the development of neural retriever with analysis of its performance. Finally in chapter 7 provides an walk through in the code search engine that is the wrapper over the retrieval system we develop in previous chapter. Capabilities and limitations of this project as well as future works and final thoughts are explained in the chapter 8.

Chapter 2

Related Works

The prevalent works in the domain of our project are discussed as following: section 2.1 covers the study of the information retrieval; section 2.2 covers the work on neural network models in learning a latent representation of documents (source code with more metadata) which allows fine tuning the model for task specific datasets; section 2.3 covers the works on generative language models. Furthermore, we compare the available code related datasets with our own dataset (as shown in details in chapter 5) in table 2.1 and in section 2.4.

2.1 Retrieval Systems

Information retrieval (IR) is the process of finding and accessing relevant information from a collection of documents. The history of IR can be traced back to the creation of electromechanical searching devices in the late 19th and early 20th centuries, such as the Mundaneum and the Memex [79]. The first computerized IR systems emerged in the 1950s, such as the Univac and SMART [79], which used simple keyword matching and statistical techniques to rank documents. In the 1960s and 1970s, IR research focused on developing more sophisticated models of document representation and retrieval, such as the vector space model, the probabilistic model, and the Boolean model [15]. In the 1980s and 1990s, IR research expanded to deal with various types of documents and queries, such as hypertext, multimedia, natural language, and speech. In the 2000s and 2010s, IR research was influenced by the rise of the web and its challenges, such as scalability, diversity, personalization, and evaluation.

In late 2017, neural retrievers, heralded for their superior performance over traditional term-based counterparts like TF-IDF and BM25, have demonstrated efficacy across diverse domains, particularly when ample training data is available [36, 43, 37]. However, the

expensive annotation of retrieval datasets for novel tasks has spurred exploration into methodologies that enhance neural retrievers in zero-shot scenarios. Contriever, an unsupervised approach [32], pre-trains neural retrievers on unlabeled data, offering a potential solution to the cost constraints associated with dataset annotation. Another strategy involves leveraging large-scale supervised datasets, such as MS MARCO [58], to train a single retrieval system, transferring the acquired knowledge to new datasets [32, 37, 60, 12]. However, challenges arise as these models often struggle to generalize beyond their training data [27].

To overcome such limitations, a third paradigm emphasizes training specialized retrievers tailored to specific tasks, utilizing unlabeled corpora and leveraging another model’s capabilities to autonomously generate necessary training data [88]. Notably, Hidey and McKeown [27] employed task-specific templates and few-shot samples to automatically generate in-domain training queries, selecting documents from the target corpus and utilizing Fine-tuned Language Model (FLAN) as mentioned in [92]. Yet, this approach often demands the use of extensive large language models and the training of distinct retrievers, resulting in a slow and resource-intensive adaptation process.

Another noteworthy technique involves pre-training language models on source codes, leveraging large-scale code corpora to develop general-purpose dense representations applicable to various downstream tasks. This approach enhances both code understanding and natural text comprehension, as demonstrated by studies such as Zhang et al. [97] and Wang et al. [87]. The benefits extend to tasks like code summarization, documentation generation, bug fixing, and automated code review, making it a promising technique with cross-modal applicability.

In the realm of retrieval systems, the use of dense representation of text stands out as a transformative technique. This method maps text into low-dimensional continuous vectors, capturing semantic meaning and offering advantages over sparse and high-dimensional representations like bag-of-words or TF-IDF. Dense representation overcomes vocabulary mismatch challenges, facilitates efficient retrieval through *Maximum Inner Product Search* (MIPS) algorithms and specialized hardware, and supports diverse scenarios such as open-domain question answering and cross-lingual retrieval [49, 53].

2.2 Code Understanding

The CodeXGLUE benchmark, introduced by Lu et al. [52], addresses the multifaceted landscape of code understanding through three fundamental tasks: defect detection, clone detection, and code search. Defect detection is framed as a binary classification task, as demonstrated by Zhou et al. [99], who present the Devign model evaluated on prominent open-source C projects. Meanwhile, Russell et al. [75] focus on function-level vulnerability detection using open-source C/C++ repositories. To delve deeper into code semantics, Svajlenko et al. [81] propose the BigCloneBench benchmark, gauging the similarity between code pairs for clone detection, derived from validated open-source Java repositories. However, the adequacy of defect and clone detection for comprehensively evaluating code semantics understanding has been contested [89, 23], primarily due to their language-specific nature and limited coverage.

In contrast, code search encompasses semantic relevance in both code-to-code and text-to-code contexts. Notably, Husain et al. [30] and the CodeSearchNet benchmark adopt a code description or the first documentation as a text query to retrieve corresponding functions. Recognizing the limitations of existing benchmarks, CodeXGLUE’s code search considers semantic similarity for a given query code or code description [52, 30]. This expansion aligns with a surge in datasets, prompting the release of various Language Models (LM) and LLMs specifically tailored for code understanding. Inspired by the success of transformer-based pre-trained LLMs such as BERT [16], GPT [67], and T5 [68] in generic text datasets, the development of pre-trained LMs on code like CodeBERT and CodeT5 has demonstrated significant prowess in both code comprehension and generation tasks [18, 89]. This evolution underscores the pivotal role of transformer-based pre-trained models in advancing the understanding of code semantics.

The landscape of code understanding has witnessed a surge in popularity, primarily propelled by the notable success of pre-trained LLMs in various coding tasks. Particularly, decoder-only models, exemplified by works such as CodeFill [33] and CodeGen [59], along with encoder-decoder models like CodeT5 [89], UnixCoder [23], and PLBART [2], have demonstrated remarkable performances. Among the pinnacle achievers are PaLM [14] and AlphaCode [47], surpassing the coding capabilities of the average human participant in competition-level scenarios. This success has spurred researchers to push the boundaries of code generation tasks, leading to the formulation of more challenging and factually intricate objectives. These objectives manifest in two primary categories: code-to-code generation

and text-to-code generation. As the field evolves, the focus extends beyond conventional coding exercises, inspiring the development of tasks that demand a nuanced understanding of code and its diverse applications.

2.3 Code Generation

In the realm of code-to-code generation tasks, such as automatic program repair (APR) [84] and code translation [52], the conventional metric-based automatic evaluation measures like BLEU [63], CodeBLEU [71], and exact match scores fall short in effectively assessing the quality of generated code. Recognizing this limitation, Berabi et al. [6] have made a significant contribution by introducing a comprehensive JavaScript patch repair dataset derived from GitHub commits. Notably, they leverage a static analyzer, ESLint¹, to identify 52 distinct error types. Going beyond conventional evaluation metrics, their work emphasizes the importance of introducing an error removal metric that considers diverse forms of error fixes. This novel approach aims to enhance the reliability and feasibility of code generation evaluation.

Addressing the intricacies of code semantic and syntactic evaluation, there is a growing demand for execution-based evaluation methods, coupled with comprehensive test suites. A noteworthy benchmark in the domain of Java APR is Defects4J [35], which evaluates the correctness of fixes based on the ability to pass all relevant test cases and provide the desired functionality. However, a notable drawback of Defects4J is its lack of a cohesive training corpus. In response to this limitation, researchers commonly resort to constructing training datasets using GitHub’s publicly available repositories, relying on bug-specific commit messages [101]. Unfortunately, this heuristic-based approach introduces bug-irrelevant commits and unrelated code pairs, significantly impacting the quality of the collected training dataset [93]. As the field progresses, it becomes imperative to address such challenges for a more accurate and robust evaluation of code generation tasks in natural language processing research.

For text-to-code generation, the cornerstone dataset CONCODE [31] has emerged as a prominent resource for advancing our understanding of the complex interplay between natural language (NL) comments and Java code snippets. This dataset, renowned for its widespread use in research and development, is meticulously curated to encompass a diverse array of

¹<https://eslint.org>

nl comments paired with corresponding Java code snippets. The dataset’s construction involves the systematic scraping of code snippets from open-domain Java GitHub repositories, thereby ensuring a representative and varied selection of real-world programming scenarios. Notably, the inclusion of nl comments is accomplished through the judicious application of heuristics designed to extract relevant information from Javadoc, showcasing a strategic approach to capturing the nuanced relationship between human-readable explanations and machine-executable code. By leveraging the CONCODE dataset, researchers gain access to a rich and authentic repository of language and code interactions, fostering advancements in the field of natural language processing (NLP) for code understanding [31].

2.4 Code Datasets

Several innovative efforts have been made to harness the wealth of publicly available programming resources. One notable initiative is JuICe [1], which systematically gathers Jupyter notebooks from GitHub, recognizing their potential as valuable repositories of practical code implementations. Concurrently, CoNaLa [94] focuses on the aggregation of Python and Java code snippets accompanied by natural language comments from the expansive database of StackOverflow posts. Noteworthy is the commitment to enhancing the overall quality of these datasets through the involvement of professional annotators. Complementing this, the MoCoNaLa project [90] emerges as an extension of CoNaLa, aiming to broaden its scope by incorporating support for a more diverse array of natural languages. This collaborative pursuit signifies a multifaceted approach to building comprehensive and linguistically diverse code corpora, thereby advancing the understanding and modeling of programming languages in a broader linguistic context.

The inadequacy of general lexical-based evaluation metrics in assessing the accuracy of generated code is a recognized challenge. A noteworthy approach addressing this limitation is the ODEX framework, as introduced by Wang et al. in their work on code execution evaluation [91]. Unlike conventional metrics, ODEX adopts an execution-based evaluation paradigm, relying on human-written test cases derived from diverse Python libraries. This methodology has found widespread application in evaluating benchmarks within the Data Science domain, exemplified by its use in DSP [10], DS-1000 [42], and Exe-DS [29]. Additionally, its utility extends to single-language settings, as evidenced by its incorporation in general code generation benchmarks such as HumanEval [11], MBPP [5], and APPS [26]. Going beyond the confines of a single language, specialized benchmarks like multi-turn MTPB [59] and multi-language CodeContests [47] also embrace the use of test cases and

Table 2.1: Comparison between xCODEEVAL and other benchmarks and datasets.

Dataset	Train	Test	Lal	Task Type	Evaluation	Level	Genre
Django [61]	16,000	1,805	1	Program Synthesis	Lexical	Local	N/A
WikiSQL [98]	56,355	15,878	1	SQL Queries	Lexical	Modular	SQL
Miceli Barone and Sennrich [55]	109,108	2,000	1	Synthesis, Summarization	Lexical	Local	GitHub
CoNaLa [94]	2,379	500	2	Program Synthesis	Lexical	Local	Stackoverflow: QA
CONCODE [31]	100,000	2,000	1	Program Synthesis	Lexical	Modular	GitHub
Android [64]	26,600	3,546	1	Program Synthesis	Lexical	Local	Map oriented, GitHub
CodeSearchNet [30]	6,452,446	99	6	Plain Text, Retrieval	NDCG	Modular	GitHub
JuICE [1]	1,518,049	1,981	1	Notebook Cell Gen.	Lexical	Local	Prog. assignment
TransCoder [73]	721MB	1,410	3	Program Translation	Lexical	Modular	GitHub
HumanEval [11]	-	164	1	Program Synthesis	Execution	Modular	Interview Question
HumanEval-X [83]	-	820	9	Synthesis & Translation	Execution	Modular	Interview Question
MBPP [5]	-	974	1	Program Synthesis	Execution	Modular	Interview Question
CodeXGLUE [52]	2,840,000	759,000	9	10 Tasks	Lexical	Local	N/A
AVATAR [3]	5,937	1,693	2	Program Translation	Lexical	Global	Problem Solving
TFix [6]	84,846	10,504	1	Program Repair	Lexical	Local	GitHub
CCSD [51]	84,316	6,533	1	Program Summarization	Lexical	Modular	Linux Kernel
TL-CodeSum [28]	55,766	6,971	1	Program Summarization	Lexical	Modular	GitHub
CodeNet [66]	8,906,769	2,783,365	55	Classification, similarity	Lexical	Global	Problem Solving
TransCoder-ST [74]	333,542	103,488	3	Program Translation	Execution	Modular	GitHub
DSP [10]	-	1,119	1	Notebook Cell Gen.	Execution	Local	Math and Data Science
MTPB [59]	-	115	1	Multi-turn Code Gen.	Execution	Local	Problem Solving
Exe-DS [29]	119,266	534	1	Notebook Cell Gen.	Execution	Local	Data Science
DS-1000 [42]	-	1,000	1	Notebook Cell Gen.	Execution	Local	Data Science
MoCoNaLa [90]	-	896	1	Program Synthesis	Lexical	Local	StackOverflow
ARCADE [95]	-	1,082	1	Notebook Cell Gen.	Lexical	Local	Data Science
ODEX [91]	-	945	1	Program Synthesis	Execution	Local	StackOverflow
MBXP [4]	-	13,877	10	Program Synthesis	Execution	Modular	Interview Question
XLCoST [100]	496,333	45,394	7	10 Task	Lexical	Local, Global	GitHub
DeepFix [25]	37,000	7,000	1	Program Repair	Ececution	Global	Compile Error, Students
Defects4J [35]	-	835	1	Program Repair	Execution	Local, Global	N/A
APPS [26]	5,000	5,000	1	Program Synthesis	Execution	Global	Interview Question
CodeContests [47]	4,432,447	32,181	3	Program Synthesis	Execution	Global	Problem Solving
CoderEval [96]	-	460	2	Program Synthesis	Execution	Modular, Global	GitHub
Humanevalpack [57]	-	6×164	6	Program Synthesis	Execution	Modular	Interview Question
BioCoder [82]	-	2,522	2	Program Synthesis	Execution	Modular, Global	GitHub
CodeApex [20]	-	706	1	3 tasks	Execution	Modular	Online Judge platform
xCODEEVAL (ours)	19,915,150	159,464	17	7 Tasks, see table 5.3	Execution	Global	Problem Solving

exploit code execution as a pivotal component for enhanced evaluation. The incorporation of execution-based assessment methods, as exemplified by ODEX, reflects a conscientious effort within the NLP research community to establish more robust and comprehensive evaluation frameworks for generated code.

Comparison between xCODEEVAL and other benchmarks are compiled in table 2.1. For simplicity, we combine NL-code generation and code completion as Program Synthesis. Compared to others, xCODEEVAL offers the largest suite of training and test data and a more comprehensive set of test cases. Evaluation levels *Global*, *Modular*, and *Local* refer to document, function, and statements level evaluation, respectively. We elaborate more on these evaluation levels in our discussion of *executability* in chapter 4.

Chapter 3

System Architecture

Commencing our endeavor, the primary objective is to develop a sophisticated code search engine that offers a responsive interface, adept at furnishing functionally pertinent codes from an expansive database. This search engine functions as a wrapper encapsulating the core code retrieval system, strategically designed to enhance accessibility, user-friendliness, and customization to cater to specific requirements.

At the heart of our retrieval system lies information retrieval, a pivotal process that necessitates the delineation of a query for retrieval purposes and a scrupulously curated database from which results are extracted. This intricate process mandates the incorporation of two key components: a user interface facilitating query input, seamlessly handled by our frontend, and an extensive database housing an array of codes to proficiently respond to queries, carefully organized within the dataset we construct. The ensuing challenge then lies in the delicate definition of relevance and the subsequent ranking of documents in response to a given query, a task that hinges on an intermediate phase of indexing, further elucidated in section 3.2.

As expounded in chapter 5, our data acquisition strategy involves the extraction of data from Codeforces, wherein each code snippet is tagged with a distinctive *problem_id*. This identifier serves as a criterion for discerning functional similarity, as it enables the assessment of whether two codes constitute correct solutions to a shared problem. Though, this evaluative approach is merely a facet of gauging our model's performance, it is also of utmost important concept for preparing semantically meaningful dataset for training. We dedicated chapter 4 solely for the discussion on executability and functional similarity of codes.

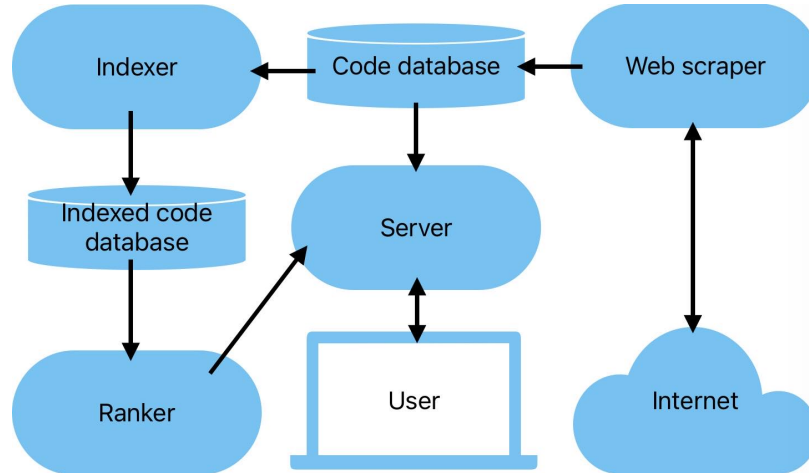


Figure 3.1: Functional block diagram of the search engine.

To evaluate the functionality of our system, we employ an encoder language model to generate real vector embeddings for individual codes, subsequently gauging their functional congruence by measuring the dot product as the inner product of said embeddings. This approach not only facilitates an exact evaluation but also defines the metric induced from the inner product, thereby establishing a robust framework for performance assessment.

In summation, the comprehensive project can be bifurcated into four distinct yet interconnected blocks, each possessing its own life cycle. Commencing with a assiduously crafted web scraper functioning as the data provider, we transition to a machine learning system responsible for updating the dense retrieval model, denoted as the **indexer**. Subsequently, a **ranker** comes into play, employing FAISS to refine the ranking process. Finally, a **server** equipped with a web interface is implemented to facilitate seamless user interaction. Figure 3.1 encapsulates this intricate orchestration, presenting a simplified yet comprehensive functional block diagram of the search engine.

3.1 Web scraper: building dataset

In the realm of retrievers, the significance of possessing an extensive retrieval corpus or database cannot be overstated. As elucidated by Google¹, the Google Search index, comprising hundreds of billions of webpages, surpasses a colossal size of 100 petabytes.

¹<https://www.google.com/search/howsearchworks/how-search-works/organizing-information>

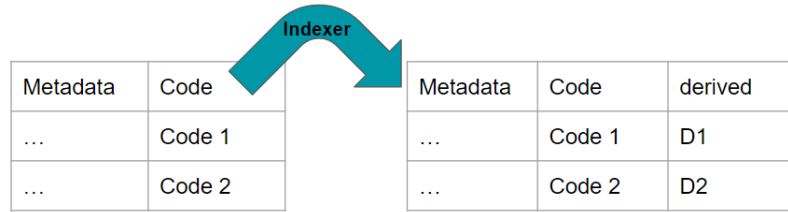


Figure 3.2: Principle of database indexing.

This colossal scale underscores the indispensability of a vast corpus for efficient indexing.

For code retrieval systems, source codes can be procured from diverse public repositories such as GitHub, GitLab, and competitive programming platforms like Codeforces, Aizu, AtCoder, and others. While general crawlers may yield noisy data due to variations in data policies and structures across websites, bespoke crawlers tailored for specific platforms can ensure the acquisition of high-quality data, complete with well-structured annotations. However, it is imperative to acknowledge that specialized crawlers necessitate vigilant maintenance owing to the dynamic nature of website layouts and data representations. Additionally, the pursuit of amassing copious amounts of data mandates scalable storage solutions and formidable computational resources. Notably, Codeforces emerges as a preeminent repository, renowned for its richness in both the quality and quantity of available data. Furthermore, the flexibility to continuously augment the database by downloading additional data aligns with the standard practices observed in the crawling and indexing pipelines integral to any search engine architecture.

In the context of this comprehensive perspective, a dedicated scraper was meticulously developed for Codeforces, resulting in the extraction of a substantial 660 gigabytes of data. Subsequent refinement processes culminated in a curated collection of raw source codes, totaling 40 gigabytes and spanning contributions from diverse contestants addressing 7,514 problems hosted on Codeforces. This concerted effort facilitated the establishment of xCODEEVAL. A detailed exploration of the scraper, data processing methodologies, and pertinent statistical analyses is presented in chapter 5. Noteworthy inclusions within the dataset, such as annotations, enable the assessment of the functional relevance of query responses generated by the retriever, thus enhancing the robustness of the research framework.

3.2 Indexer

The foundational objective of indexing within the realm of database management is the computation of derived values for individual rows or documents, culminating in a systematic enhancement of search operations within the database. This pivotal functionality is succinctly encapsulated in figure 3.2, illustrating the systematic derivation of values for each document within the database. To expound further, consider the exemplar scenario wherein a user data table incorporates a birth date field, and the objective is to ascertain the number of users falling within a specific age range. This necessitates the derivation of age values for each user, thereby facilitating subsequent counting operations within the designated age range.

In our specific context, the derivation involves the computation of real vectors for each code, which are subsequently utilized for ordering rows based on the maximum inner product. The choice to employ real vectors and seek the maximum inner product warrants justification. As delineated in chapter 2, early retrieval systems extensively relied on diverse text processing algorithms and metrics to compute various derived values, instrumental in identifying functionally analogous documents. However, inherent challenges surface with this approach, primarily encompassing the potential dissimilarity between two codes that employ disparate algorithms, variable names, or coding patterns while addressing the same problem. Albeit extant text algorithms proficiently handle variable name mangling and coding pattern variations, they falter in discerning disparate codes that solve identical algorithmic challenge without executing the actual code.

A secondary impediment lies in the multi-stage nature of such systems, an attribute that distinguishes them from the end-to-end solutions proffered by neural retrievers. As substantiated by the comparative analysis in chapter 2, neural retrievers, particularly those grounded in neural architectures, eclipse their non-neural counterparts. The advantage lies in the provision of an end-to-end solution, obviating the explicit extraction of features from codes. Consequently, the adoption of neural retrievers becomes imperative.

This necessitates the calculation of a real vector, termed an *embedding*, for each code through BERT-based models, colloquially referred to as encoders. The nomenclature aligns with their capacity to encode all pertinent attributes from the textual content into a real vector. A pertinent query arises: why opt for BERT-based models, specifically masked language models, over alternatives such as GPT or T5 models, categorized as autoregressive

language models? The answer lies in the mathematical representation inherent to the language model architectures.

BERT models, in contrast to GPT and T5, predict token probabilities based on both prefix and suffix information of the token. This bidirectional contextual representation enables BERT models to extract features that represent contextual relations more comprehensively, all achieved with a reduced number of model parameters. Consequently, the advantage of training smaller, faster models is realized, with the added benefit of BERT training being more parallelizable compared to autoregressive counterparts, thereby expediting the training process significantly [69].

The experimental endeavors undertaken in this research involved the refinement of BERT-based models, specifically CodeBERT and Starencoder. These models are designed to encode textual codes into an embedding within the vector space \mathbb{R}^{768} , constituting a dense representation of the code. The training methodology employed was premised on the notion that the vectors corresponding to similar codes would exhibit greater proximity than those representing dissimilar codes. In essence, the objective function guiding this process is identified as the *Inverse Cloze Test* (ICT), a term explicated further in chapter 6. The fundamental principle underlying the ICT framework is the optimization for increased inner product values between vectors representing similar codes, in contrast to dissimilar ones.

The adoption of a dense representation for textual or code data is a conceptually ingenious strategy, poised to enhance the discrimination between texts that may be syntactically proximate but lack semantic similarity. A comprehensive exploration of the intricacies involved in the training regimen and the consequential outcomes is presented in chapter 6, providing a nuanced understanding of the methodologies employed and the efficacy of the implemented models.

3.3 Ranker

The efficacy of a retrieval system hinges upon its ability to swiftly navigate through an extensive dataset. Within the retrieval system, the role of a ranker is paramount; it diligently searches and assesses the relevance of documents by employing efficient similarity scoring mechanisms. In alignment with our established preferences, the selection of a ranker becomes pivotal, necessitating a system adept at probing for akin embeddings within the

embedding space.

Inverse Cloze Test objective serves as our guiding principle, ensuring that embeddings corresponding to similar vectors exhibit higher inner product values compared to their dissimilar counterparts within the embedding space. Consequently, the chosen ranker must embody an algorithm characterized by efficiency in executing Maximum Inner Product Search (MIPS) [80, 24]. To this end, FAISS (Facebook AI Similarity Search) [34] emerges as a compelling choice, proficiently navigating encoded vectors (embeddings) within the indexed corpus to identify embeddings with the highest inner product values in relation to the query vector.

The development of FAISS by Johnson et al. [34] builds upon prior contributions by Shrivastava and Li [80] and Guo et al. [24], amalgamating diverse techniques and algorithms. FAISS achieves state-of-the-art efficiency by facilitating MIPS with GPUs at an unprecedented billion-document scale. This accomplishment is underpinned by the integration of innovative data structures, pre-computations, and quantization techniques. The culmination of these advancements yields a distributed k -NN (k -Nearest Neighbors) algorithm tailored for searching nearest neighbors within high-dimensional real vector spaces. Notably, FAISS elucidates the intrinsic connection between MIPS and nearest neighbors in the embedding space, thereby substantiating how k -NN resolution elegantly addresses the MIPS problem.

3.4 Server

Within our code search engine architecture, we have sedulously devised a comprehensive frontend and backend infrastructure to serve as the indispensable wrapper over our sophisticated retrieval system. This symbiotic relationship between the frontend and backend components constitutes the backbone of our endeavor, seamlessly integrating cutting-edge technologies to ensure a robust and user-friendly experience.

The backend, positioned as the linchpin connecting our FAISS ranker to the frontend web interface, plays a pivotal role in facilitating user-system interactions. This critical element is responsible for orchestrating the retrieval process, delving into the indexed corpus/database with unparalleled efficiency. Its seamless operation enables it to promptly respond to HTTP requests, appending invaluable additional metadata to the retrieved results. This

augmentation is strategically designed to enhance visualization on the frontend, offering users a refined understanding of the retrieved data.

On the frontend, we have crafted a sophisticated web interface that serves as the gateway for users to interact with the system. At its core, this interface features an intuitive user input field, allowing users to input query text seamlessly. The subsequent presentation of search results occurs in a structured list format, accompanied by thoroughly curated additional metadata. This metadata serves a dual purpose – not only does it contribute to justifying the accuracy of the retrieved results but also enriches the user’s understanding of the underlying data.

To further elevate the user experience, our web interface incorporates insightful plots that visually articulate the similarity between the query and results. These plots leverage various accuracy measurements, providing users with a multidimensional perspective on the efficacy of the search operation. The integration of such visual aids not only enhances the interpretability of results but also aligns with our commitment to delivering a transparent and comprehensible user experience.

Chapter 7 serves as an exhaustive repository of knowledge, offering a accurate and elaborate description of both the frontend and backend servers. This section provides an in-depth exploration of the intricacies inherent in our architecture, elucidating the synergies between the frontend and backend components that form the backbone of our innovative code search engine.

Chapter 4

Executability

Executability of codes is the most vital aspect of both the dataset and the retrieval model we present in this book. We define *executability of code* as the ability to measure syntactic correctness and functional correctness. Both correctness are essential measurement for any code [72]. The *syntactic correctness* for a code means it conforms to the grammar presented by the compiler version of the language and thus equivalently measures whether it compiles or not. The *functional correctness* or *algorithmic correctness* on the other hand denote whether the code solves a particular problem or not within specified time and memory complexity. Although code capabilities of LLMs were measured with lexical or syntactic algorithms, de facto executability is the only natural way to evaluate codes generated/retrieved by LLMs. We call the evaluation frameworks that measure executability the *execution-based evaluation* frameworks. We further divide the execution-based evaluation frameworks into three categories as follows:

1. **Modular:** It can measure correctness for individual functions. Such frameworks inject the function to be tested in a template code and then successful termination of the executable represents correctness. Function name is predefined in this case.
2. **Local:** Here the test code comprises of several statements which is then injected in the template code. Variable and function names are predefined in these cases too. Similar to *modular*, successful termination of the code represent correctness.
3. **Global:** this frameworks can evaluate only a complete program. There is no template code. To test correctness of any code, one needs to have the proper collection of unit tests that can rigorously measure the correctness of the code. A *unit test* represent a *input* and *expected output* pair and during evaluation correctness is implied if

providing the input to the executable produces the expected output. By a proper collection of unit tests we mean a list of unit tests that covers all sort of corner cases from algorithmic sense.

The key aspect of XCODEEVAL dataset is offering unit tests for all data in validation and test splits for the tasks. This feature makes our dataset execution base evaluation compatible. But there is no available global execution-based evaluation framework publicly available, let alone being secure or multilingual. This inspired the development of ExecEval¹. Before we dive in the discussion on ExecEval, we will define functional similarity between two codes or a natural description and a code which is paramount to the evaluation of retrieval performance on our dataset.

4.1 Functional Similarity

Mathematically speaking similarity is an equivalence relation (i.e. reflexive, symmetric, transitive) and a popular theorem states that having the equivalence classes of the relation partitions the set. According to Sajnani [78], two codes are functional clones if they implement the same functionality. This implies the codes have same time and memory complexity. This is indeed an equivalence relation on any set of codes (trivially any code belongs to its own equivalence class) [48]. From technical perspective, implementation details of the codes implementing same algorithm and hardware states during executing those codes can result in different execution time and memory consumption. This poses an unavoidable obstacle for judging functional clones.

We need a definition of relevance for code retrieval and we want the definition to be motivated by both *executability* and equivalence of functional clones. This is solved by defining two codes *functionally similar* if they are the correct solution for same *problem* (an algorithmic challenge with a natural text description). *Problem* will be formally defined in section 5.1. Informally, every code have an unique identifier denoting an algorithmic challenge and an execution outcome of whether the code is a correct or not. This is again an equivalence relation as every code belongs to only one problem. This attribute in the dataset allows the performance evaluation of retrieval based on functional similarity of codes. Analogously a natural language description and a code is defined to be functionally similar if the code is a correct solution for the description. Via the same annotated attributes for *code-code* case,

¹<https://github.com/ntunlp/ExecEval>

n-code functional similarity can be measured while evaluating any retrieval model with XCODEEVAL.

4.2 ExecEval

The latest trend in code generation is to move to a new metric called $\text{pass}@k$ backed by an execution-based evaluation framework to capture the functional correctness of the generated code instead of using *n*-gram matching algorithms capturing the syntactic similarity such as codebleu [41, 11]. Two challenges emerge in moving to a complete execution-based evaluation framework: firstly one needs to have unit tests for the generative tasks, secondly a execution framework that enables compiling and executing arbitrarily generated code in specific languages. Chen et al. [11] only implemented the support for python and function level execution. XCODEEVAL has the unit tests for validation and test splits of all generative tasks, which solves the first challenge in execution-based evaluation.

To solve the second challenge, which is an essential requirement for execution-based evaluation in different programming languages is the availability of a secure and scalable framework [9]. We developed a distributed, extensible, secured, dockerized execution engine named ExecEval. It offers the solution for evaluating programs with unit tests in multiple programming languages. With its capacity to support 44 compiler/interpreter versions in 11 different languages of XCODEEVAL, ExecEval offers a versatile and comprehensive approach to program evaluation. Table 4.1 shows the currently supported list of compilers and interpreters. The engine is distributed as a secure Docker image, ensuring the safe and efficient execution of potentially malicious generated programs such as fork bomb, arbitrary file system i/o etc. This feature makes it an ideal tool for researchers who require a trustworthy and secure environment to evaluate their code. It also provides scripts to generate $\text{pass}@k$ reports for codes generated against the generative tasks offered through XCODEEVAL. In addition, ExecEval supports easy integration of new compilers and interpreters with custom execution flags (flags can also be changed at runtime).

4.2.1 Architecture

ExecEval is hosted as a HTTP server. It exposes two APIs, one for list of supported compiler versions (GET), one for executing any code (POST). The docker container is built over ubuntu base image and the compiler versions in table 4.1 are installed during build

Table 4.1: Supported languages and their versions in ExecEval.

Language	compiler	Supported Versions
Ruby	ruby 3.0.2p107	Ruby 3
Javascript	node.js v16.17.1	Node.js v16.17.1
Go	go1.19.2	Go 1.19
C++	gcc 12.1.0	GNU C++17, GNU C++20, GNU C++11, GNU C++14, GNU C++17, GNU C++0x
	clang 14.0.0	Clang++17, Clang++20
C	gcc 12.1.0	GNU C11, GNU C
Java	java 19.0.2	Java 6, Java 7, Java 17, Java 11, Java 8
Python	PyPy 7.3.9 with GCC 10.2.1	PyPy 3.9.12, PyPy 2.7.18,
	Python 2.7.18	Python 2.7.18
	Python 3.11.0rc1	Python 3.11
C#	Mono JIT compiler version 6.12.0.182	Mono C# 6.12
PHP	PHP 8.1.2	PHP 8.1
Rust	rustc 1.67.1	Rust 2021, Rust 2018, Rust 2015
Kotlin	Kotlin 1.7.20	Kotlin 1.7.20

time. Gunicorn² takes Flask app objects and spawn workers according to user defined ‘num workers’. When a POST request to execute a code is received, a subprocess is initiated to compile the code and then another subprocess execute the executable generated in previous step. Prlimit³ and seccomp⁴ is used to restrict access within predefined boundaries to make it secure. We run the executable once for each unit test provided in the POST request. Input is provided through *stdin* and *stdout* is read and parsed to match with the expected output of the unit test. ExecEval responds with execution outcome for each of the unit tests and a solution is considered functionally correct if ExecEval reports PASSED for all of the unit tests.

4.2.2 Execution Outcome

During compiling the code and executing the executable, several scenario can occur. These scenarios are listed below along with the execution outcome that will be reported by ExecEval.

- **COMPILATION ERROR:** The program fails to compile or run due to a syntax error.
- **RUNTIME ERROR:** The program successfully compiles but fails during runtime due to native environment issues (i.e., asserts, division-by-zero, heap/stack overflow).

²<https://gunicorn.org/>

³<https://man7.org/linux/man-pages/man1/prlimit.1.html>

⁴<https://man7.org/linux/man-pages/man2/seccomp.2.html>

- `MEMORY LIMIT EXCEEDED`: The program occupies more memory than the memory limit for it during execution.
- `TIME LIMIT EXCEEDED`: The program requires more time than the limit to produce an output for an unit test.
- `WRONG ANSWER`: The program successfully compiles (or interprets) and generates an output but fails to produce a correct answer for the unit tests, potentially having logical errors.
- `PASSED`: A solution that successfully pass all the unit tests. The program will be flagged as buggy (1-5) even when it fails on a single unit test.

Chapter 5

Dataset Preparation

5.1 Data Representation and Relations

To address the limitations mentioned on section 2.4, and drive further advancements in the creation of more general-purpose LLMs for problem solving, we introduce `xCODEEVAL`, the largest executable multilingual multitask benchmark to date consisting of 25M coding examples from about 7.5K unique algorithmic problems. It covers up to 17 programming languages with the parallelism of multilingual data which can benefit both mono- and multi-lingual code intelligence applications. It features a total of 7 tasks involving code understanding, generation, translation and retrieval, and wherever appropriate it employs an execution-based evaluation protocol. A detailed documentation of the dataset will be presented in this chapter. Figure 5.1 shows an example from `xCODEEVAL`; it includes a problem description in natural language, a buggy and bug-free solution to the problem, and relevant metadata such as difficulty level, language, problem tags (e.g., brute force).

The dataset prepared from codes downloaded from Codeforces along with various metadata. Codeforces is an online competitive programming platform that hosts contests on a regular basis. Formally speaking, each contest C_i has a set of problems $P_j \in \mathcal{P}$ (set of all problems), and each problem has a set of submissions $S_k \in \mathcal{S}$ (set of all submissions). Each submission belongs to an unique user U_l of Codeforces. Also each problem is annotated with a list of algorithmic techniques here called tags $T_j \subset \mathcal{T}$, the set of all tags. Figure 5.1 shows an example of a submission and with the problem.

Figure 5.1 presents an example of a problem along with a correct submission, and a wrong submission. Each problem includes a natural language description, input, and output description, and a few sample *i/o* examples. It also includes relevant meta-information such

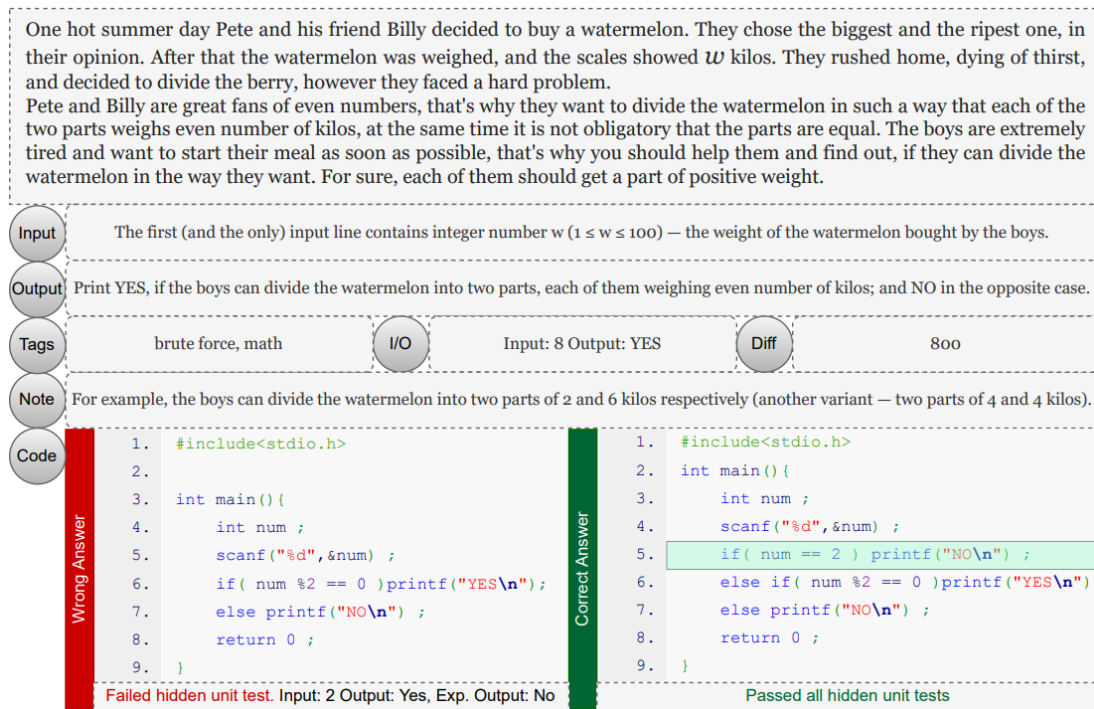


Figure 5.1: An example of a problem along with a correct submission, and a wrong submission.

as language, difficulty level (800 in the figure), problem tags (e.g., brute force, math), and a note (explanation of i/o). It also contains a number hidden unit tests (not shown in the figure) against which one can evaluate the corresponding code. For example, although the code at the left gives the correct answer to the given input, the solution is actually incorrect.

To better grasp the data representation of contest, problem, submission, and user some entities and relations between them are defined here. All the entities and attributes used by Codeforces will not be defined here rather only the ones that are using in preparing the dataset, also such information is published publicly¹. Figure 5.2 shows the ER diagram relevant to this project. The entities are defined below.

- *Problem*: A text description of an algorithmic task with i/o specification according to which a user has to write a solution code in any supported programming languages. Has a unique *problem id*, *contest id*, and a list of *tags*. Here tags come from a list of 37 algorithms and techniques. The list of *tags* for the problem represents the algorithms and techniques needed for solving the problem.

¹<https://codeforces.com/apiHelp/objects>

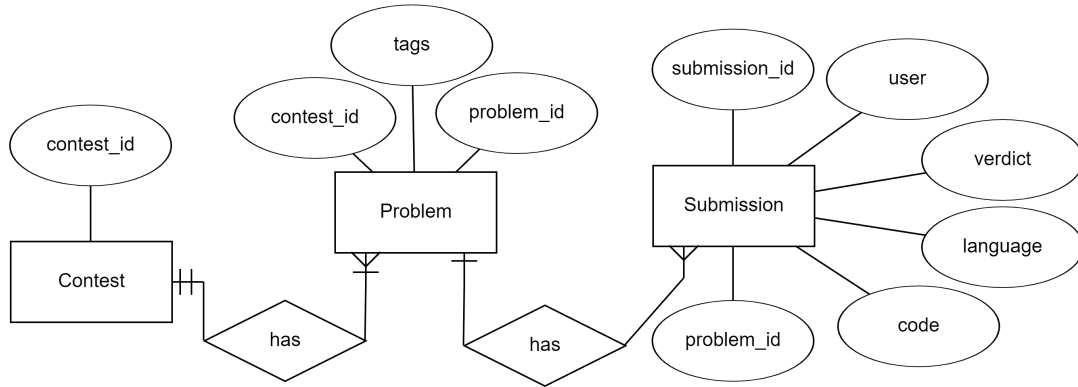


Figure 5.2: Minimal ER diagram of Codeforces database.

- *Submission*: A text *code* written in any supported programming *language* by an unique *user* as an attempt at solving a unique *problem*. It has an unique *submission id*. It also has a *verdict* justifying the code as being correct (referred to as Accepted/OK/PASSED in later parts) or incorrect (anything other than Accepted/OK/PASSED).
- *Contest*: A collection of *problems* along with their *submissions* and other statistical data. It has an unique *contest id*.

With the above entities available in structured way at Codeforces, the aim is to download the problems and submissions to prepare *nl-code*, *code-code* datasets utilizing the relations of the entities to improve quality of the downloaded data in the sense of maximizing the coverage of different attributes.

5.2 Overview of Scraping Architecture

This section provides a deep dive into the scraper developed for the download. It depends on the following technologies: Scrapy, tor, privoxy. Privoxy service upon receiving https requests and converts them to SOCKS5 request, which can then be processed by the tor network. The architecture of the scraper is adopted from scrapy framework with some modifications to improve the fail safe mechanisms implemented by the scraper.

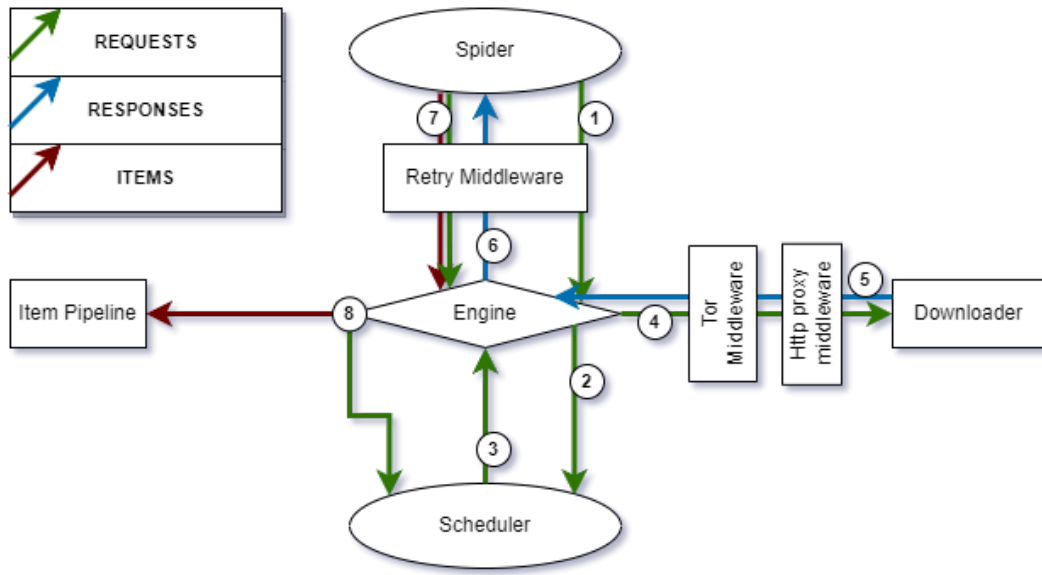


Figure 5.3: Architecture of the scraper.

The data flow of the scraper is controlled by the engine in figure 5.3. The engine uses Python's `twisted` library to run an event loop and hence allows asynchronous execution of python code, and in turn allows parallel downloads. The general control flows is as follows:

The web crawling process begins with the spider initiating a crawl request to the engine. The engine then places the request in a scheduler, a priority queue manager storing requests for crawling. Subsequently, the engine retrieves the next request from the scheduler and directs it to the tor middleware, which manages request counts, resets counters at specific intervals, and renews the tor circuit. Following the tor middleware, the request traverses a http proxy middleware, utilizing the Privoxy server URL as the http proxy, and is then forwarded to the downloader. The downloader, upon completing the page download, generates a response and forwards it to the engine. The engine, in turn, sends the response to the retry middleware for validation; unsuccessful validations prompt the request to be resent to the engine, while successful validations result in the response being sent to the spider for further processing. The spider processes the response, extracting JSON data (scraped items) and generating new requests for the engine. The engine sends items to the item pipeline, enqueues requests in the scheduler, and writes items to a disk file. This process iterates until no further requests remain in the scheduler.

In light of this architecture, the algorithm to generate sequence of requests at the spider is as follows:

1. Iterate over all contest ids (natural number upto some N and download the list of submissions metadata for the contest.
2. Partition the list of submission metadata into list of submission metadata per problem.
3. For each partition, use algorithm 1 to get the submissions download.
4. Return the requests corresponding to the newly selected submissions for download.

Algorithm 1 Algorithm to select submissions to download

- 1: **X**: List of submission metadata for any given problem, V list of bot users to ignore, n number of users to limit.
 - 2: $Y \leftarrow \emptyset$
 - 3: Let L be the programming languages available in X .
 - 4: **for all** $\text{lang} \in L$ **do**
 - 5: Let $X_{\text{lang}} = \{x \in X : x \text{ uses language 'lang'}\}$
 - 6: Let U be the users in X_{lang} who solved the problem and not in V .
 - 7: Let $c : U \rightarrow \mathbb{N}$ defined as $c(u) = \text{number of submissions used to solve the problem}$.
 - 8: Let U' be the ordered set from U with the ordering defined by $\forall a, b \in U, c(a) < c(b) \implies a < b$.
 - 9: Let F be the first n users from U' .
 - 10: $Y \leftarrow Y \cup \{x \in X_{\text{lang}} : x.\text{user} \in F\}$
 - 11: **end for**
 - 12: **return** Y
-

In this case, $N = 1760$. After downloading all valid submissions for N contests, the submissions were appended to a file in `jsonl` format. A total of $\sim 25\text{M}$ submissions are downloaded for $\sim 9\text{K}$ problems along with the hidden unit tests used to test the correctness of the code in the submission. It took 660GB on the disk to store the raw data. The next section covers the distribution of values across different attributes of the downloaded data.

5.3 Statistics of Raw Data

Here are some interesting and noteworthy statistics from the raw data. For rest of this chapter we will call *passed* submissions as pure in the sense that the code in those submissions are

Table 5.1: Statistics of basic quantities of the downloaded raw data.

Data	Value
Submissions	25,097,955
Submissions of language of interest	24,576,186
Accepted (pure) submissions	7,014,519
Problems	9,081
Problems with rating	7,821
Tags	37
Interactive problems	151
Submissions of interactive problem	418,564
Clean code size (in GB)	39.82

rigorously asserted by Codeforces as the correct solution for their respective problems, and impure otherwise. Mixed submissions will mean both pure and impure submissions are present.

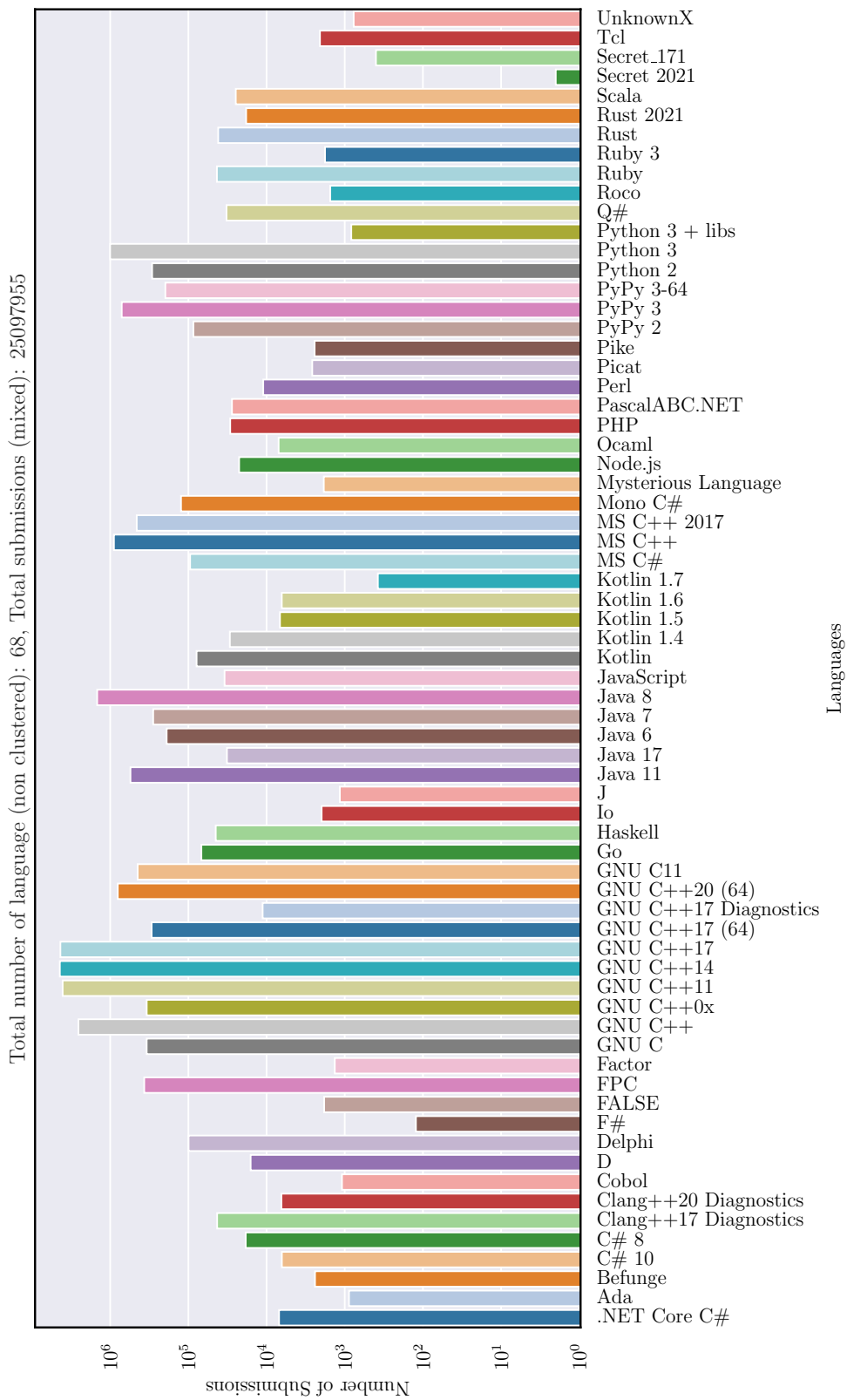


Figure 5.4: Distribution of number of submissions across all language compiler versions.

Total number of language (clustered): 17, Total submissions (pure): 7014519

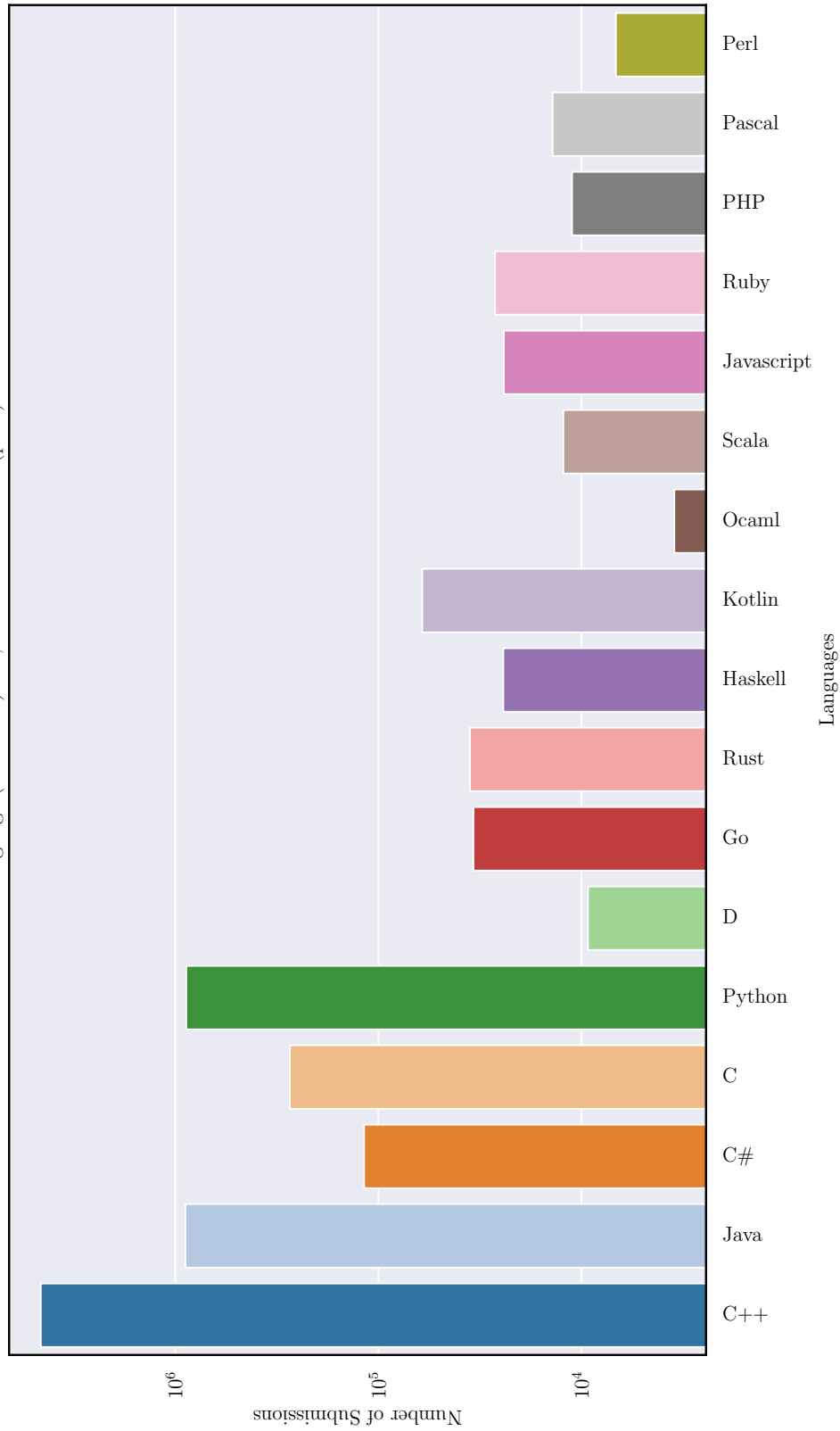


Figure 5.5: Distribution of number of submissions across all major languages.

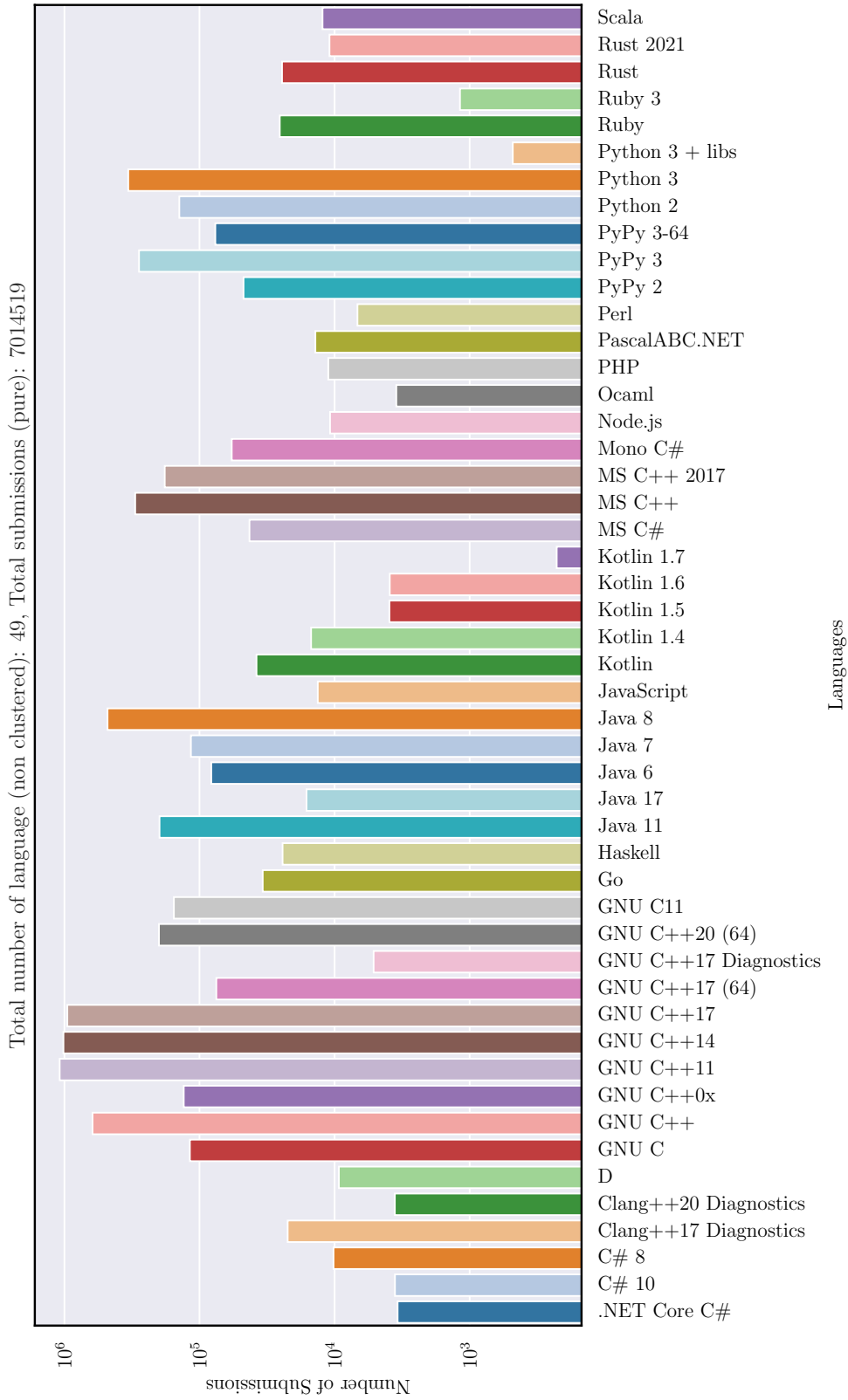


Figure 5.6: Distribution of number of pure submissions across all major language compiler versions.

In figure 5.4, a comprehensive overview of the distribution of submissions across all language compiler versions is presented. The data encompasses a total of 68 compiler version codes, providing a rich dataset comprising approximately 25 million codes. This extensive compilation lays the foundation for a thorough analysis of language diversity and usage patterns within the dataset.

Moving to figure 5.5, an insightful representation of the distribution of submissions across major programming languages is showcased. The chart lists 17 languages pivotal for generating *code-code* and *nl-code* datasets. Here, the term "major" signifies the substantial presence of submissions under these languages within the downloaded dataset. This exploration of major languages serves as a key aspect in understanding the linguistic landscape of the compiled data.

Figure 5.6 refines the analysis by providing a focused examination of the distribution of pure submissions across major language compiler versions. The figure meticulously lists 49 compiler versions belonging to the 17 major languages identified earlier. This detailed breakdown contributes to a nuanced understanding of the prevalence and distribution of submissions within specific languages.

The language coverage of the entire dataset is commendable, with a diverse array of 49 compiler versions spanning 17 major programming languages. This compilation constitutes a substantial volume of approximately 25 million mixed submissions, among which approximately 7 million are deemed pure. These findings are graphically depicted in figures 5.4 to 5.6, emphasizing the intricate interplay between language usage and submission types.

In light of these observations, there is a compelling motivation to harness the richness of this dataset for the creation of multilingual datasets. The robust representation of various languages and compiler versions makes this dataset an invaluable resource for researchers and practitioners seeking to explore and develop multilingual programming datasets.

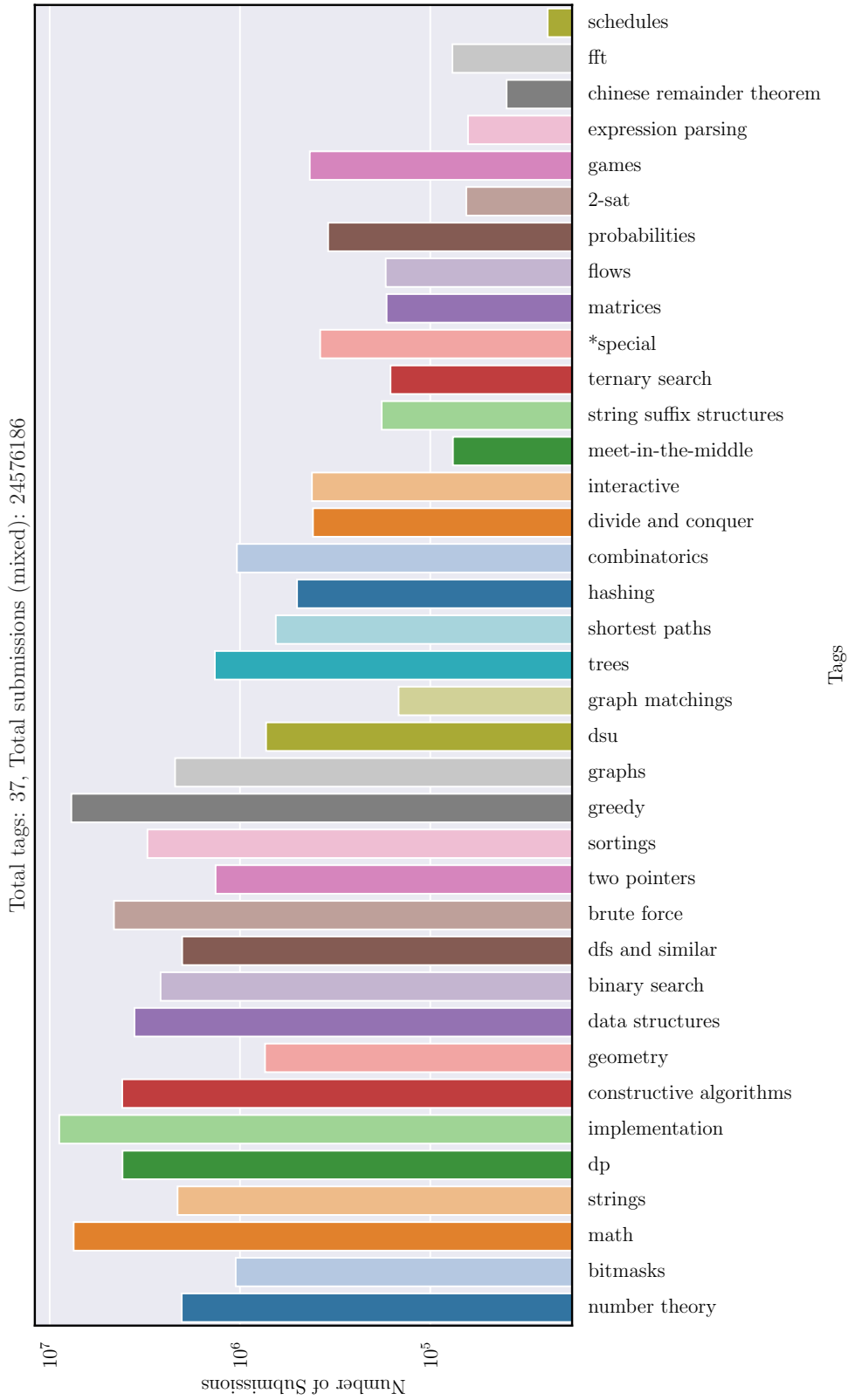


Figure 5.7: Distribution of number of mixed submissions across all tags.

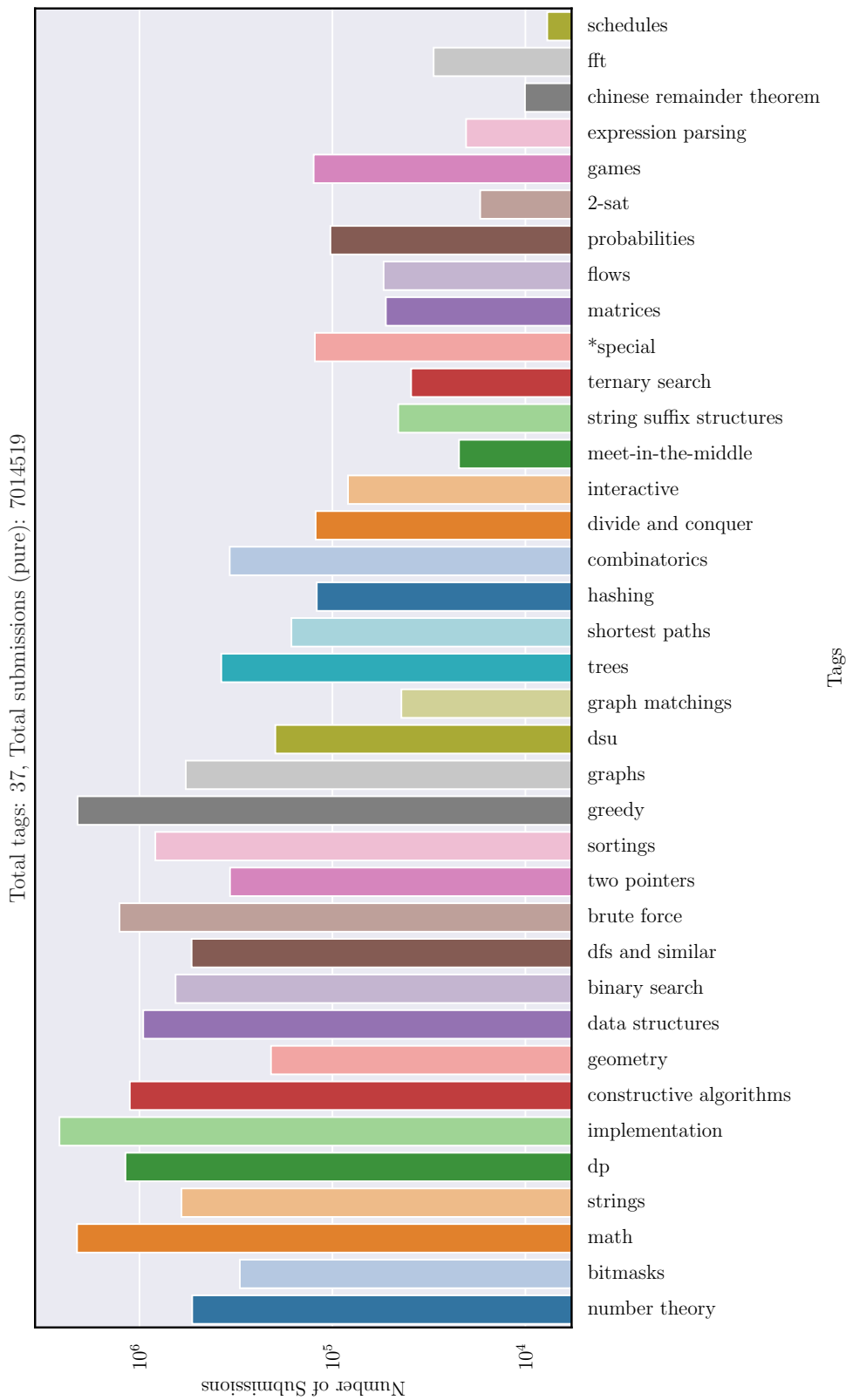


Figure 5.8: Distribution of number of pure submissions across all tags.

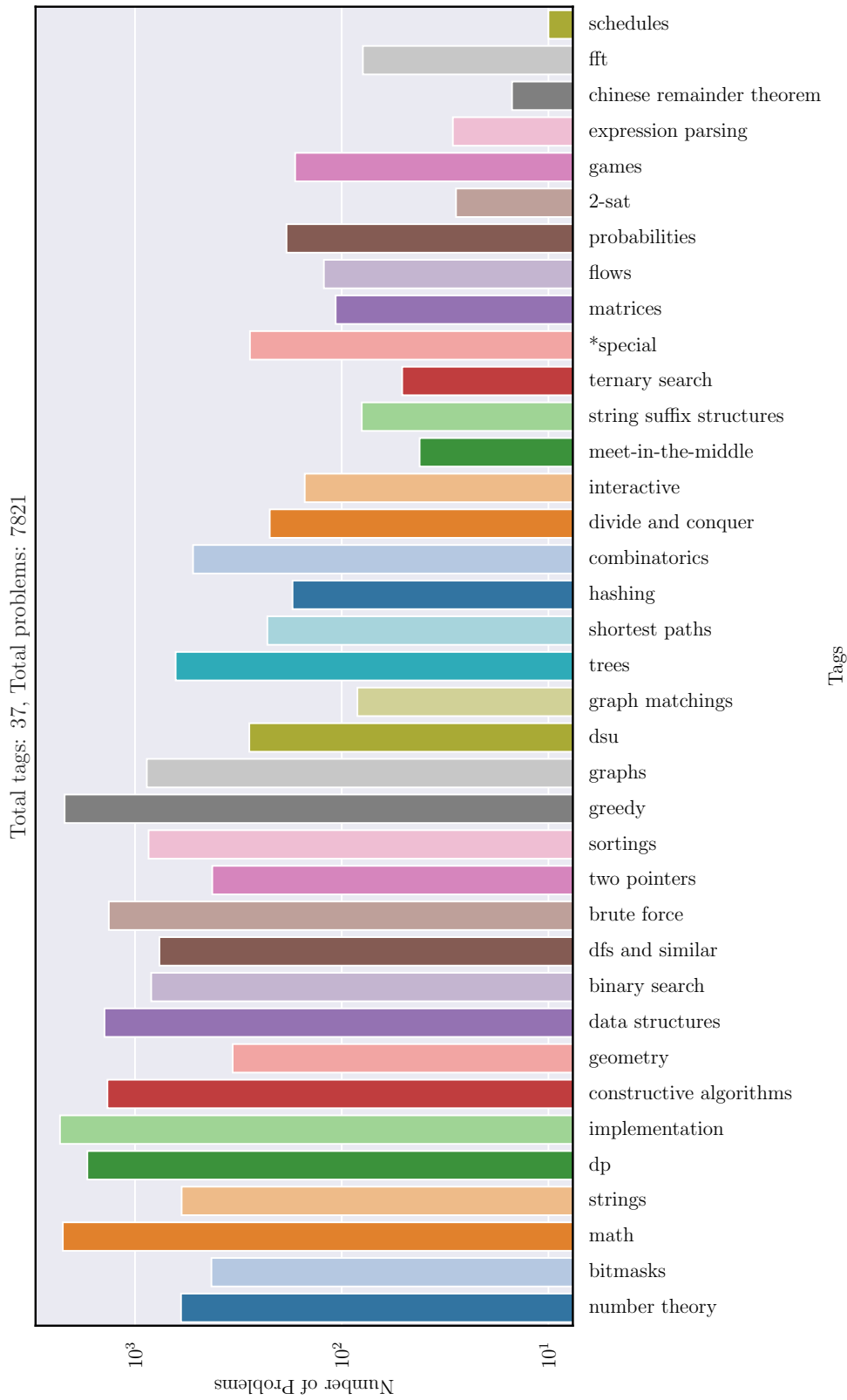


Figure 5.9: Distribution of number of problems across all tags.

In figure 5.7, a comprehensive portrayal emerges as we delve into the intricate landscape of mixed submissions, totaling approximately 25 million, distributed across a rich spectrum of 37 distinctive tags. This visual representation not only encapsulates the sheer magnitude of mixed submissions but also provides a granular view of their dispersion among the various tags T_j embedded within the dataset.

Meanwhile, figure 5.8 outlines the distribution of roughly 7 million pure submissions across the same array of 37 tags \mathcal{T} . This nuanced exploration sheds light on the prevalence of untainted contributions within each tag, offering insights into the diverse array of topics encapsulated by these pure submissions.

Furthermore, figure 5.9 offers an insightful perspective on the distribution of problems across the expansive spectrum of tags. The uniqueness of this representation lies in its consideration of the multifaceted nature of problems, counting them once for each tag to which they belong. A total of 7821 problems \mathcal{P} with associated tags are examined, providing a comprehensive view of the interplay between problems and their corresponding tags.

Upon synthesizing the information from Figures 5.7, 5.8, and 5.9, it becomes evident that the 37 tags under consideration serve as a fundamental framework for categorizing algorithmic techniques and data structures. These tags, as illustrated by the visualizations, offer a structured lens through which the vast expanse of submissions and problems can be comprehended. This becomes particularly salient in the context of typical computer science curricula, where these algorithmic techniques form the bedrock of knowledge dissemination.

In conclusion, the triad of visualizations encapsulates the intricate relationship between submissions, problems, and tags, providing a holistic view that is indispensable for researchers and practitioners alike. The narrative unfolds within the realm of fundamental computer science concepts, offering a valuable resource for those seeking a deeper understanding of the intricacies inherent in algorithmic and data structure landscapes.

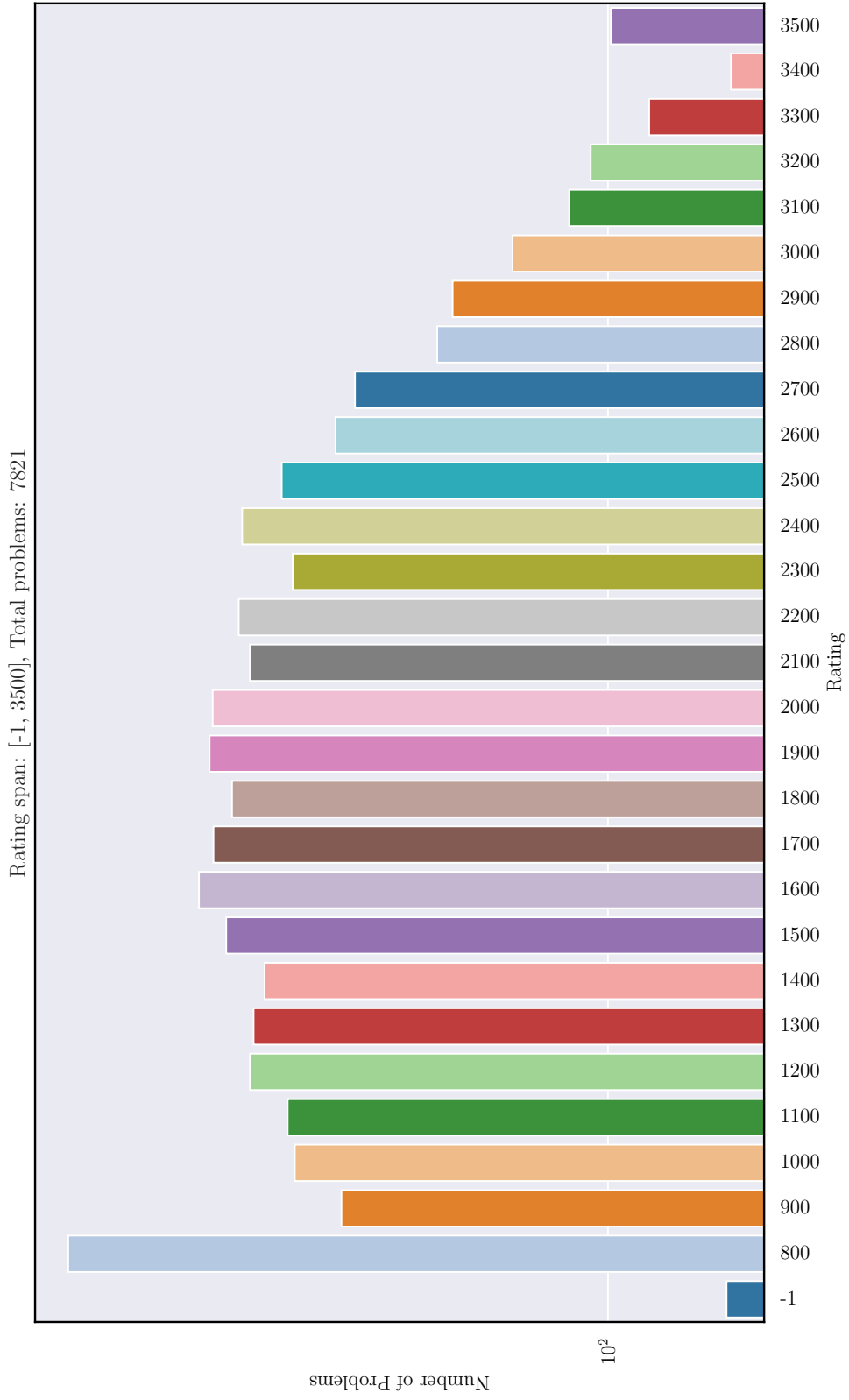


Figure 5.10: Distribution of number of problems across all difficulty rating. A difficulty rating of -1 means the rating for the problem was not assigned by Codeforces. Total problems here mean the number of problems with rating, so problems with difficulty rating of -1 is not counted.

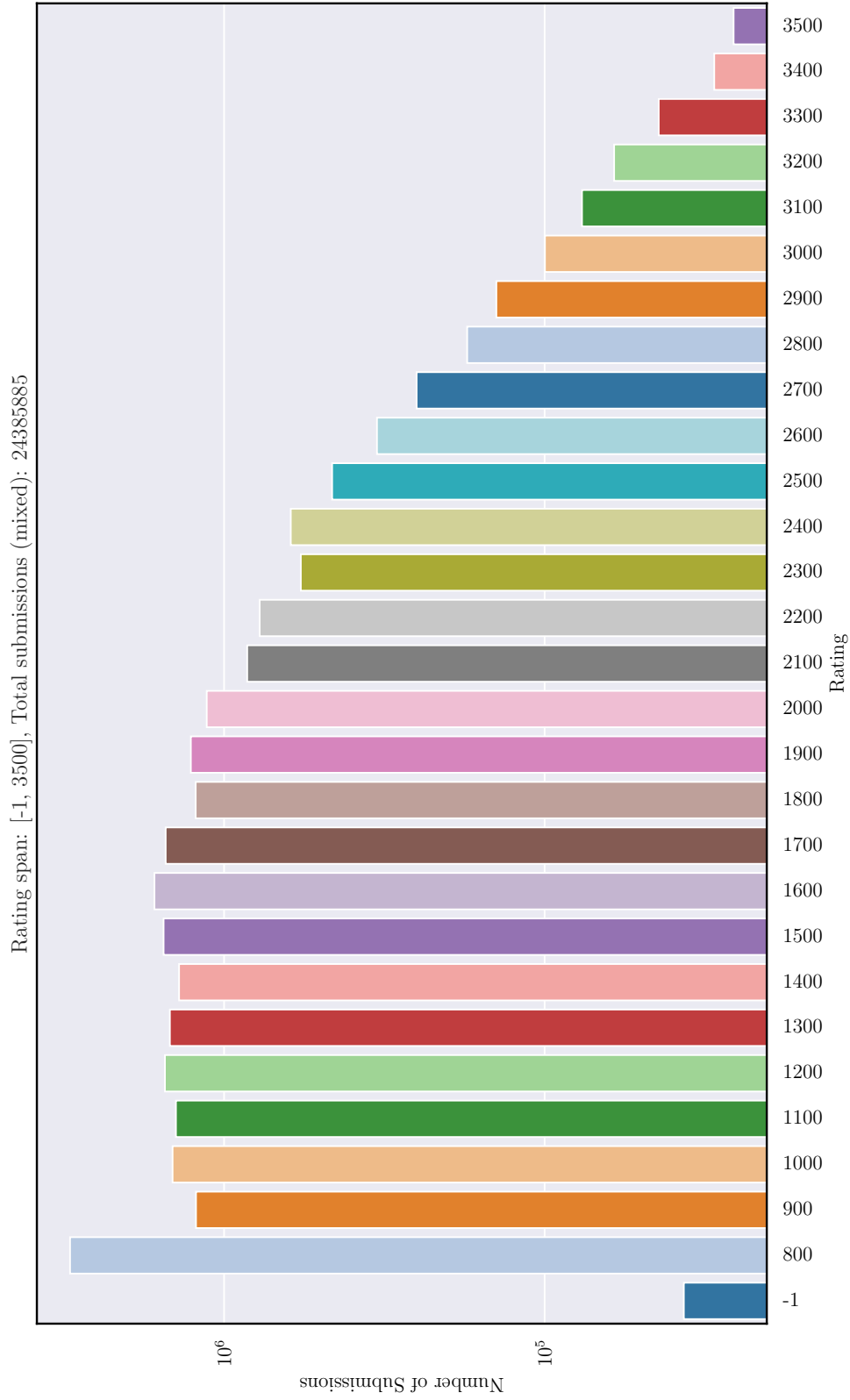


Figure 5.11: Distribution of number of mixed submissions across all difficulty rating. A difficulty rating of -1 has same meaning as discussed in figure 5.10. Total problems here mean the number of problems with rating, so problems with difficulty rating of -1 is not counted.

Delving into the intricate world of problem difficulty assessment on Codeforces, the investigation into the distribution of problems across various difficulty ratings unveils a compelling narrative. A difficulty rating of -1 denotes an absence of an assigned rating for a given problem. Ergo, the total number of problems considered in this context excludes those falling under the -1 difficulty rating, elucidating a more focused analysis in figure 5.10.

Moreover, the exploration extends beyond mere problem distribution to encompass the distribution of mixed submissions across these diverse difficulty ratings. Similar to the aforementioned scenario, problems bearing a difficulty rating of -1 are omitted from consideration in figure 5.11, ensuring a coherent evaluation.

The amalgamation of approximately 7.8K difficulty-rated problems, as illustrated in both figure 5.10 and figure 5.11, heralds a pivotal opportunity. This corpus of problems serves as a robust benchmark against which the performance of any generative model attempting to solve these conundrums across the difficulty spectrum can be meticulously measured and scrutinized.

In essence, this rich dataset not only illuminates the landscape of problem distribution but also serves as a formidable yardstick for evaluating the efficacy and adaptability of generative models in tackling challenges of varying complexity within the Codeforces domain.

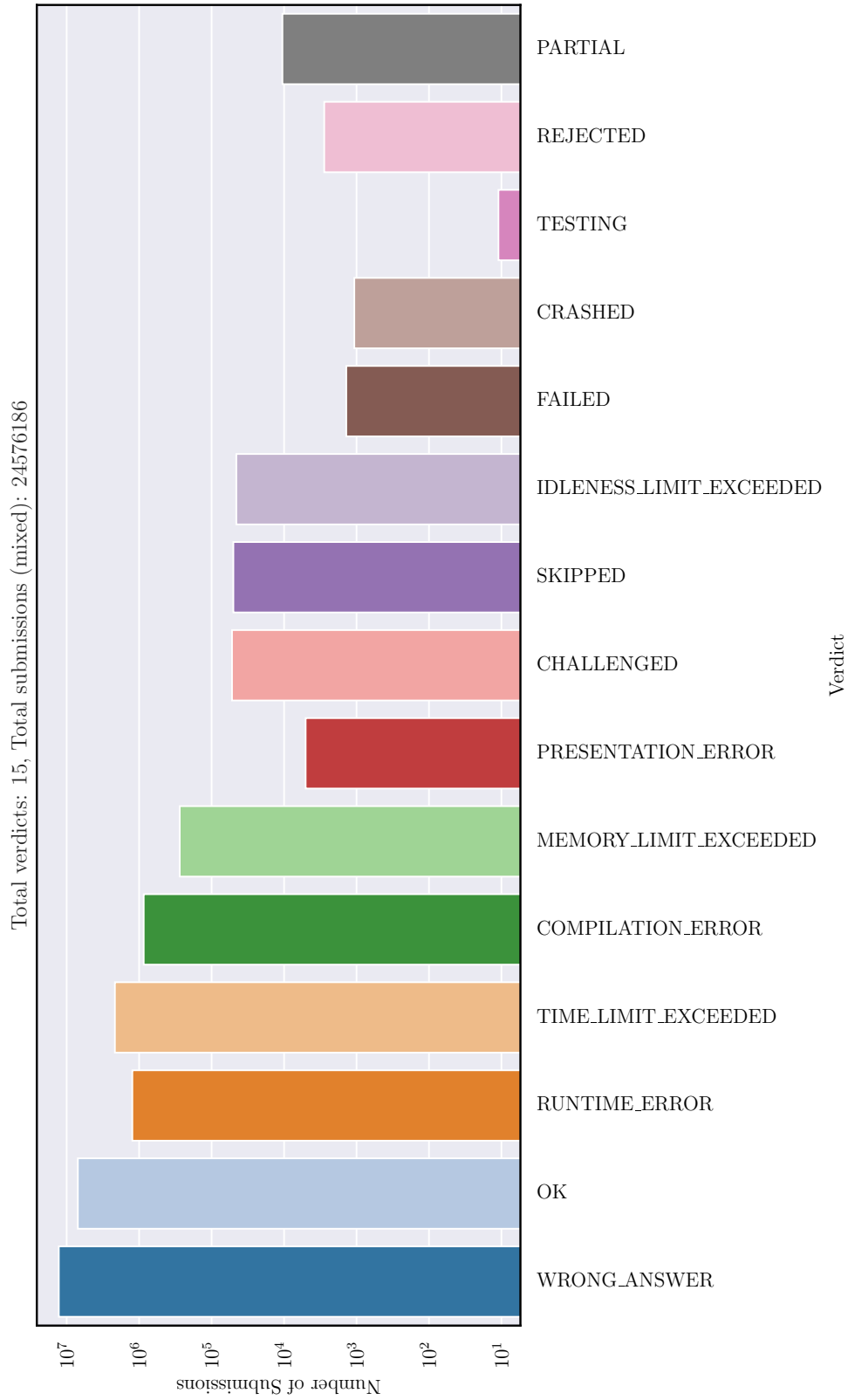


Figure 5.12: Distribution of number of submissions across all verdicts.

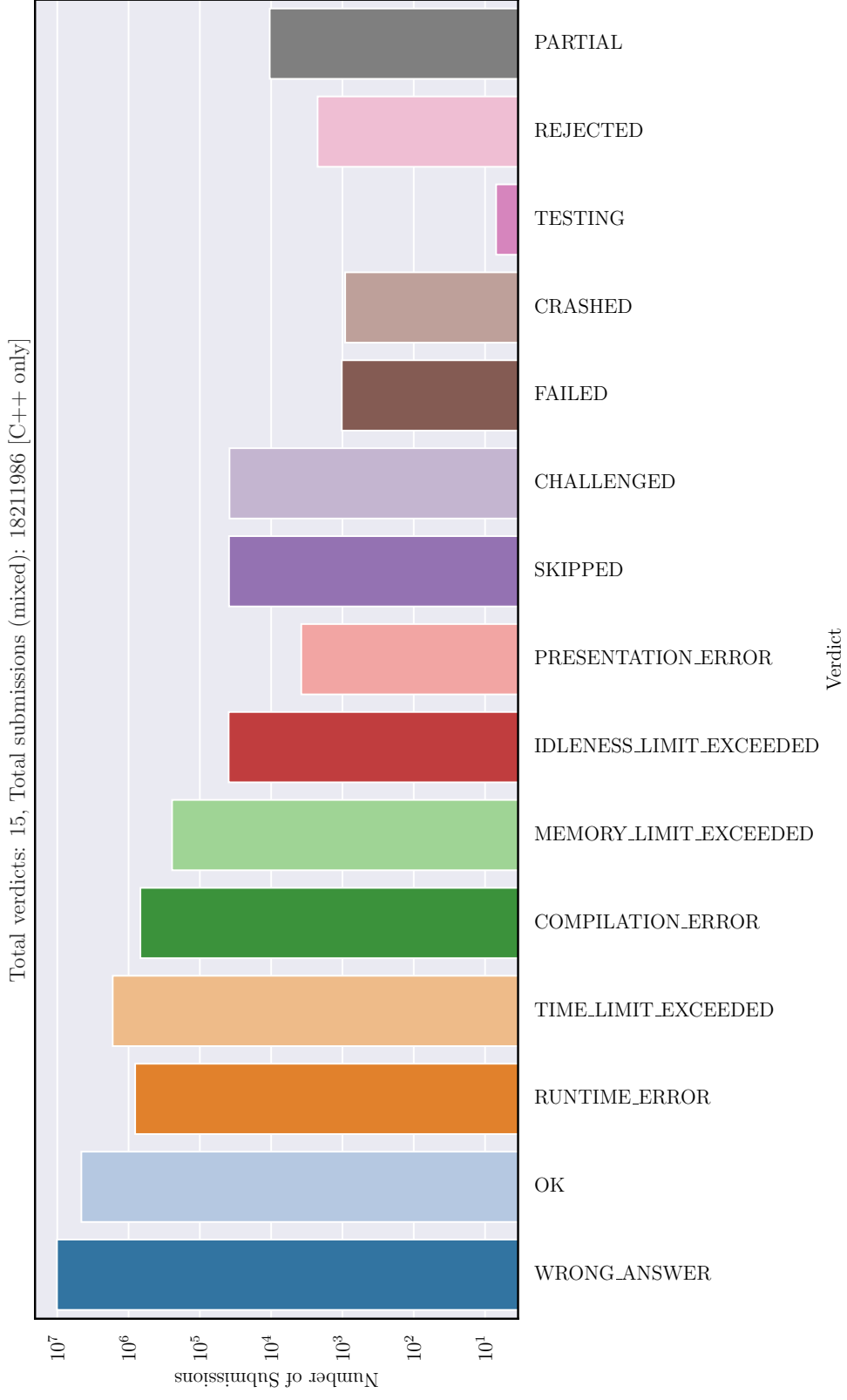


Figure 5.13: Distribution of number of submissions across all verdicts for C++.

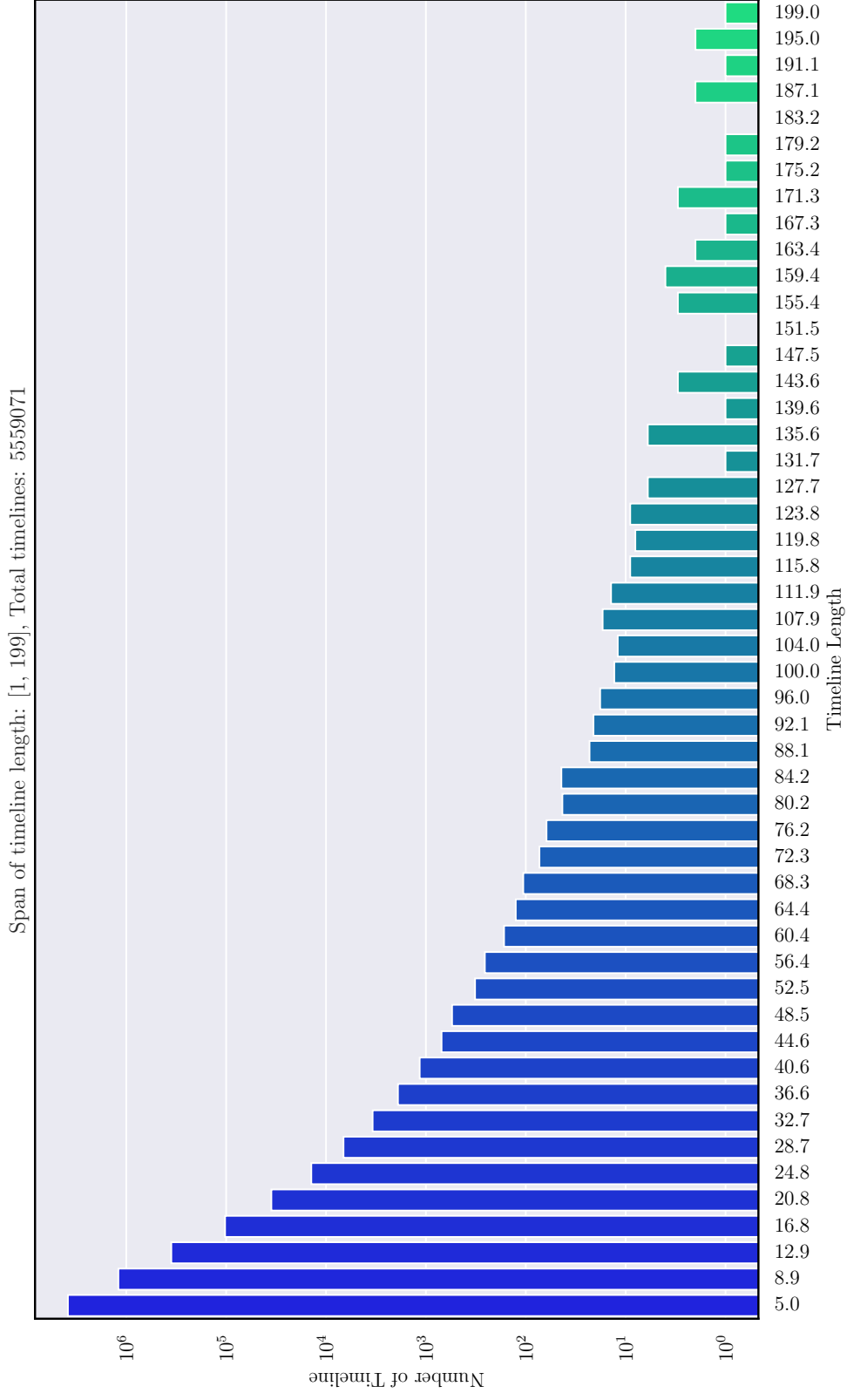


Figure 5.14: Distribution of number of timeline across timeline length.

In the examination of Codeforces data, an analysis encompassing approximately 25 million submissions is conducted, detailing the distribution across various verdicts. The specific verdicts assigned by Codeforces to individual code submissions, elucidating the outcomes derived from the execution against unit tests within their system, are encapsulated in figure 5.12.

Zooming in on submissions coded in C++, a focused investigation unveils the distribution of the number of submissions across all verdicts. As depicted in figures 5.4 to 5.6, it becomes evident that C++ exhibits a significantly higher number of mixed submissions, surpassing those in other programming languages by several orders of magnitude, as illustrated in figure 5.13.

Further delving into the temporal aspect, an analysis of approximately 5.5 million timelines is conducted, emphasizing the distribution across different timeline lengths. Each timeline encapsulates the chronological sequence of submissions leading to the moment when a particular user's submission attains an "OK" verdict for a specific problem. This exploration is intricately visualized in figure 5.14.

The culmination of insights from the verdict distribution (figures 5.12 and 5.13) and timeline analysis (figure 5.14) substantiates the premise that a robust dataset for the task of *automatic program repair* can be systematically constructed. These findings underscore the potential of leveraging Codeforces data to advance research and development in the realm of automatic program repair, showcasing the diverse patterns and characteristics inherent in the submissions and their associated execution outcomes over time.

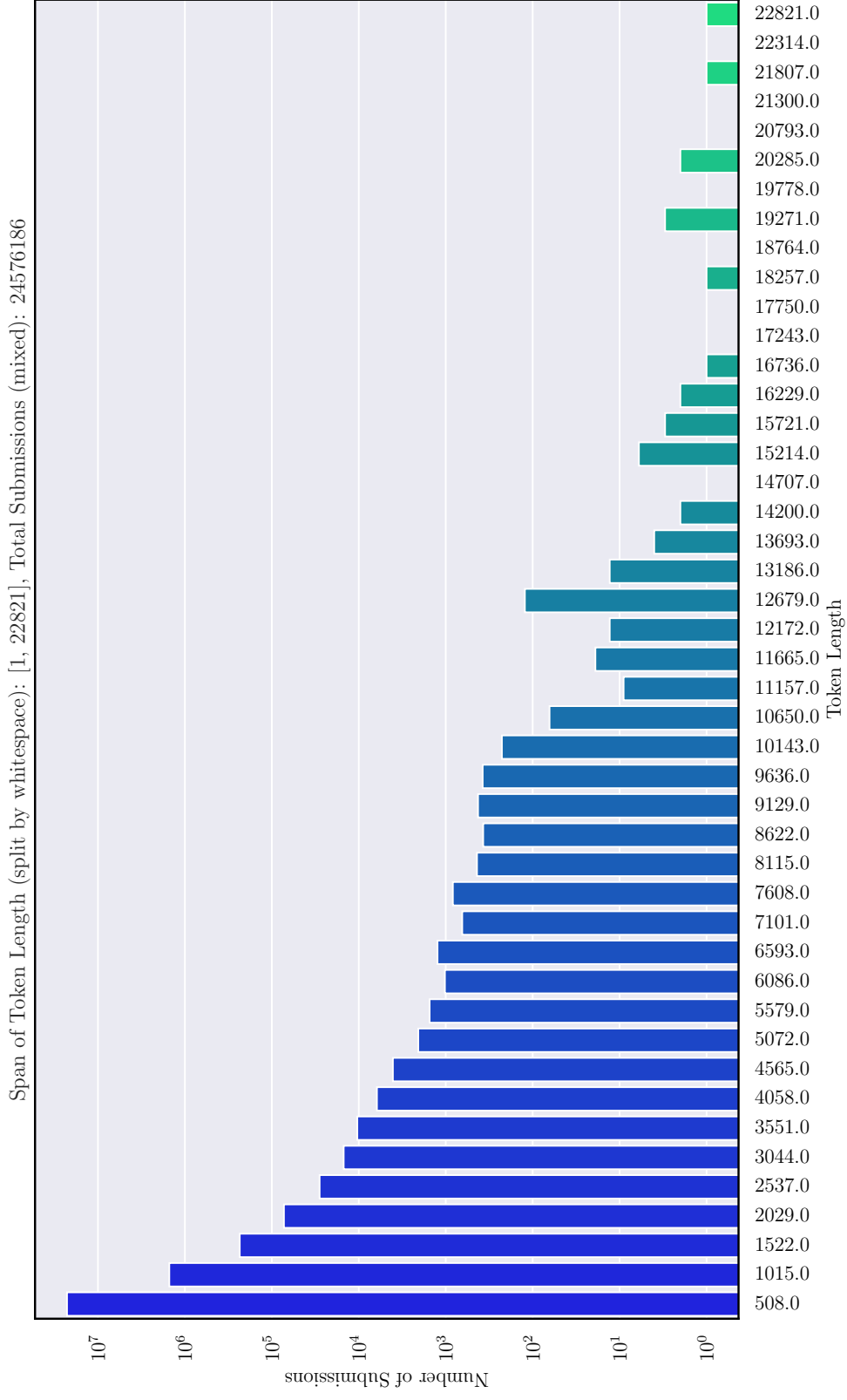


Figure 5.15: Distribution of number of mixed submissions across number of token.

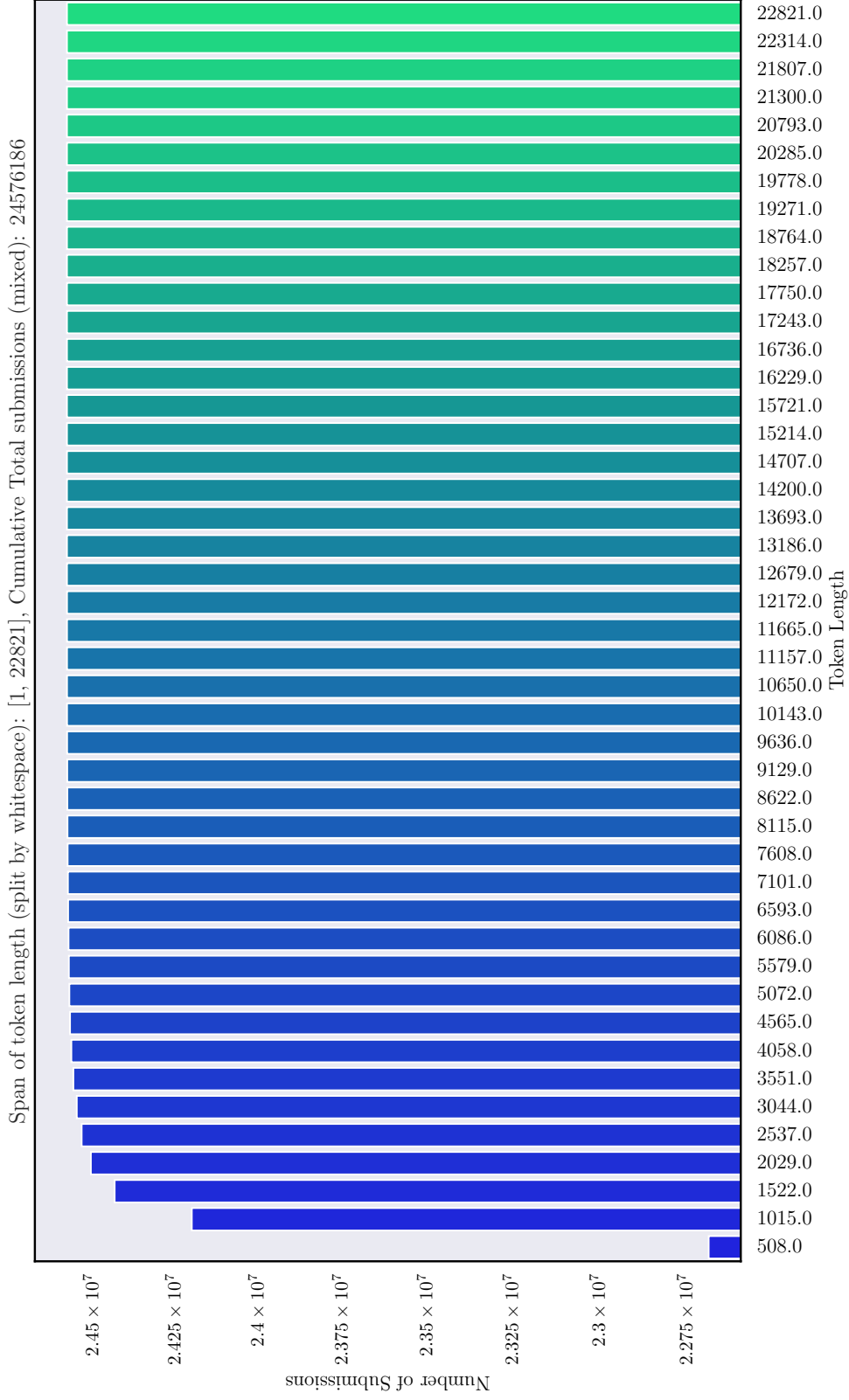


Figure 5.16: Cumulative distribution of number of mixed submissions across number of token.

In figure 5.15, we present a comprehensive analysis of the distribution of approximately 25 million mixed submissions based on the number of tokens. Tokens, in this context, are defined as the count of white-space separated substrings in a given code. The distribution provides valuable insights into the token lengths prevalent in the dataset, facilitating a nuanced understanding of the code structure and complexity.

Furthermore, figure 5.16 offers a cumulative perspective on the same dataset, depicting the accumulation of mixed submissions across different token ranges. Again, tokens are delineated as the number of white-space separated substrings in the code. This cumulative representation aids in identifying trends in the dataset and highlights the overall distribution of submissions as token length increases.

An examination of figures 5.15 and 5.16 reveals noteworthy findings. Specifically, more than 20 million submissions fall within the 500-token threshold, indicating a significant portion of the dataset with relatively concise code snippets. Furthermore, a substantial 24 million submissions are observed within the 1000-token range. This implies that the majority of submissions can be effectively handled by contemporary state-of-the-art Language Model Models (LLMs) that have demonstrated proficiency in processing code snippets of comparable lengths.

In summary, the visualizations in figures 5.15 and 5.16 underscore the manageable nature of the dataset in terms of token lengths. These insights are crucial for guiding the selection and optimization of LLMs for tasks associated with the given dataset, ensuring efficient and effective processing of mixed submissions in the context of white-space separated token counts.

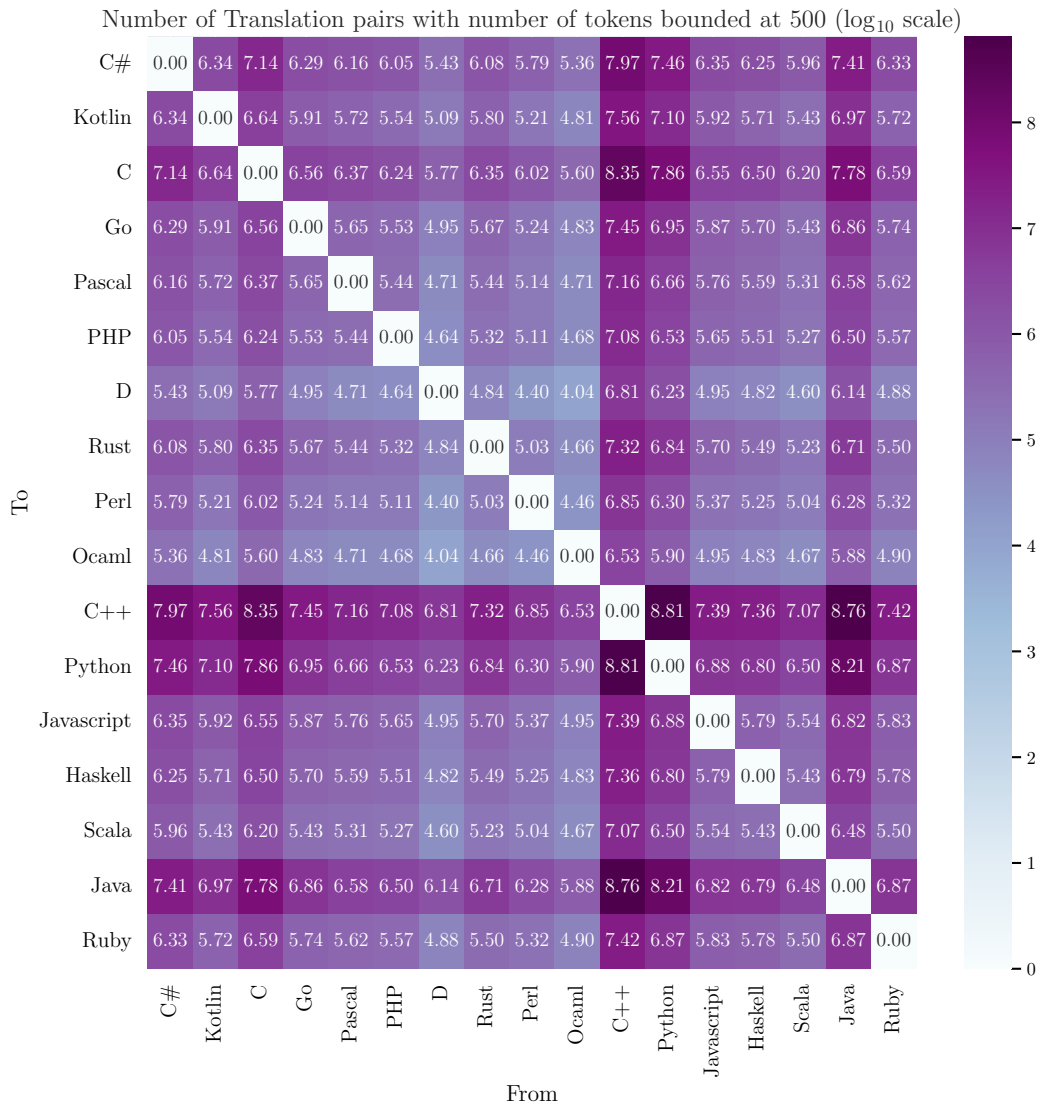


Figure 5.17: Distribution of number of translation pairs across pairs of languages limiting submissions with token length ≤ 500 with tokens defined in figure 5.15.

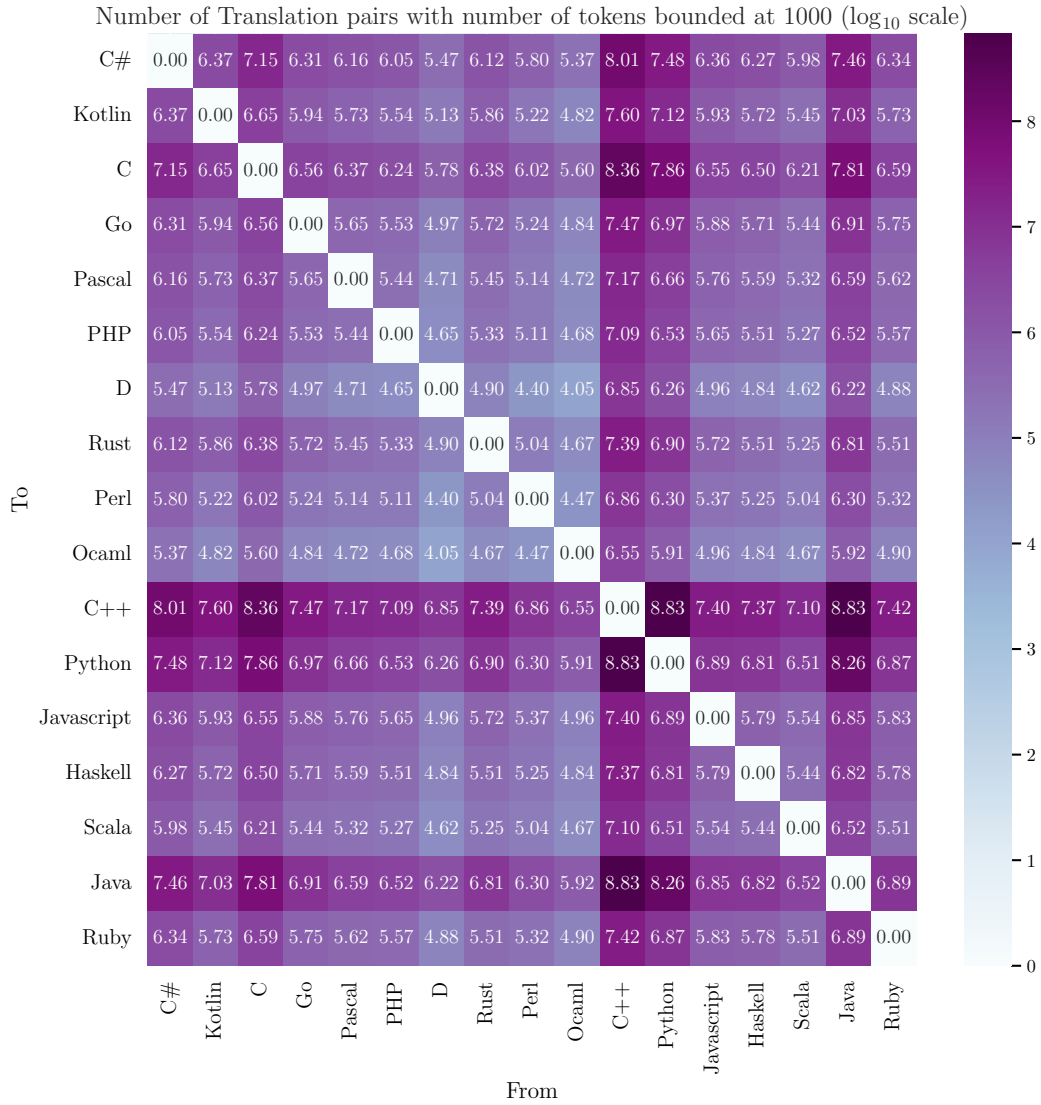


Figure 5.18: Distribution of number of translation pairs across pairs of languages limiting submissions with token length ≤ 1000 .

We now delve into the intricate landscape of translation pairs within the context of limited token lengths. Our investigation focuses on the distribution of the number of translation pairs across pairs of languages, specifically constraining submissions with a token length not exceeding 500, as elucidated by the token definitions in figure 5.15. The ensuing insights are encapsulated in the heatmap visualization presented in figure 5.17.

Furthermore, we extend our scrutiny to a broader spectrum by considering token lengths up to 1000. The distribution of translation pairs across pairs of languages under this extended constraint is portrayed in figure 5.18, providing a more comprehensive perspective on the interplay between token limitations and translation pairs.

It is essential to underscore that our approach to pairing submissions adheres to the principle of functional equivalence, as expounded in section 4.1. Functionally equivalent solutions, producing identical outputs for identical inputs, are considered interchangeable. Consequently, this equivalence criterion allows us to form translation pairs by juxtaposing any two submissions addressing the same problem but articulated in different languages.

The visual representations in figures 5.17 and 5.18 bring to light a fascinating observation – the considerable abundance of potential translation pairs. Astonishingly, the number of such pairs reaches approximately 1 billion for certain language combinations. This revelation underscores the vast space of linguistic diversity and the myriad ways in which problems are tackled across different languages. The heatmaps serve as a powerful tool for comprehending the distribution patterns, offering valuable insights into the multifaceted world of translation pairs under varying token length constraints.

5.4 Data Cleanup and Processing

In spirit of the above mentioned features of the raw data, the dataset was cleaned, and processed to produce train, validation, and test datasets for seven tasks, which is collectively called xCODEEVAL. It is a result of a number of crucial design principles and challenges as highlighted below.

Reasoning In terms of genre, *problem solving* posits a unique set of challenges that require (a) understanding a complex natural language problem description, (b) expertise in data structures and algorithms, (c) complex reasoning that goes beyond memorization,

and (d) generating programs of potentially hundreds of lines so that they can pass a comprehensive list of especially designed hidden tests. Given the current progress in LLMs and their instruction following capability [62], competition-level problems that humans find challenging, provide an interesting benchmark to test many aspects of intelligence [47, 67].

Multilinguality We aim to cover as *many programming languages* as possible regardless of the resource discrepancies. One of the main objectives of this benchmark is to assess the degree to which codes in different languages are parallel to one another. In addition to that, we also intend to evaluate the zero-shot cross-lingual capability of the LLMs.

Evaluation and its granularity We believe the current evaluation standards do not fully consider the idea of the *global* meaning representation of a program, which requires models to comprehend different interpretable code segments and connect both local and modular knowledge into a global representation. We propose *execution-based* evaluation with unit tests at the global level. While there are many benchmarks covering the local understanding of a code segment, there are only a few that work at a global level as shown in table 2.1. We consider a pair of codes to be equivalent, if they generate the same output for a given input regardless of syntax/languages (see section 4.1 for definition). To support this, we have developed `EXEC_EVAL`, a new standardized and distributed execution environment that supports 44 compilers/interpreters in all the languages in `XCODE_EVAL`. We also provide a large number of necessary unit tests (average of 50 per problem) for the relevant tasks (table 1.1). In this context, it is noteworthy that 44 out of 165 problems in the CodeContest’s test split have no private unit tests. Additionally, it contains 104 problems without complete collection of unit tests (as available in the source), thus are inadequate in assessing a solution’s correctness. We have identified this issue and excluded such problems from our evaluation sets (development and test splits).

Task difficulty and trainability We wish to focus on problems of different difficulty levels (from 800 to 3500 rating points, following Codeforces such that models with different capabilities can be benchmarked against difficulty levels. We also aim to provide sufficient training data for each task so that pre-trained LMs can be fine-tuned or small-scale models can be trained from scratch.

Data split Finally, balancing the validation and test distributions of text-code instances over multiple attributes such as problems, tags, and execution outcome (e.g., correct vs. wrong) is challenging. We propose a novel data split schema based on a geometric mean and a data

selection schema adapting a graph-theoretic solution to the circulation problem with lower and upper bounds [56] that can be applied for other benchmarks as well (section 5.4.2).

XCODEEVAL offers two classification, three generative, and two retrieval tasks. Furthermore, as discussed in chapter 2, evaluating the performance of LLMs on code generation tasks, the syntactic similarity measured by string matching algorithms are fairly unfair; rather the executability of the generated codes must be tested. This can be achieved as we have unit tests for all problems that were used by Codeforces. *Execution based evaluation* is described in chapter 4. Next two sections covers the data cleanup and processing, and the development of individual task datasets with their statistics in XCODEEVAL.

At this point there are 25M crawled submissions for a total of 7514 distinct algorithmic problems deduplicated out of 9K problems. As a reminder, each submissions $S_k \in \mathcal{S}$ represents a potential solution to a problem $P_i \in \mathcal{P}$, and a problem P_i can be solved by employing a set of data structure and algorithmic techniques $T_i \subset \mathcal{T}$, which is referred to as problem tags (e.g., 2-sat, binary search); see figure 5.20 for a complete list of tags in XCODEEVAL. A balanced distribution over \mathcal{T} and \mathcal{P} is attempted while preventing any possible data leakage to maintain the quality of the benchmark in the next section.

5.4.1 Validation-Test Split Creation

Held-Out Problems To prevent overlap of problems and submissions between training and validation/test splits, $N_h (= 1354)$ problems as set \mathcal{D}_{ho} for validation and test was put aside. This ensures that the problems in the validation and test sets are not seen in training and the model needs to generalize to be able to produce meaningful outputs for dataset generated from unseen problems.

From the held-out set \mathcal{D}_{ho} , a validation \mathcal{D}_{valid} and a test \mathcal{D}_{test} split was created maintaining a balanced tag distribution, and problem distribution. Further it was ensured that all the tags in these two sets also exist in the training data, which could be a requirement for certain tasks (e.g., tag classification in section 5.5.1.1). For this, random splits were created by iterating over a number of seeds. Let γ be the expected ratio of the number of submissions in \mathcal{D}_{valid} and \mathcal{D}_{test} , i.e., $\gamma = |\mathcal{D}_{valid}|/|\mathcal{D}_{test}|$. For each random split, a tag-wise ratio γ_t , the ratio of the number of submissions in \mathcal{D}_{valid} and \mathcal{D}_{test} for a tag $T \in \mathcal{T}$ was calculated. The geometric mean of $\{\gamma_T\}_{T \in \mathcal{T}}$ defines the ‘tag distribution’ score of a split. The split whose score is closest to γ was then selected. algorithm 2 de-

scribes the method, which ensures that the validation and test sets contain the same tag sets.²

5.4.2 Data Selection for Validation and Test

Though algorithm 2 creates two different sets of submissions with the same tag sets for both validation and test, each of the two sets may contain a few hundred thousand of submissions. For example, for tag classification (section 5.5.1.1), only C++ had 161,765 and 647,064 submissions for validation, test sets respectively. To make the testing and validation process computationally feasible, it was mandatory to reduce the sample size while maintaining a balanced distribution across problems and tags. Finding an optimal solution to this selection problem (i.e., how many submissions per problem and per tag to select) is nontrivial. This task was then formulated as a *circulation problem with lower and upper bounds* [56] within a flow network as explained below.

²Assuming that the training tag set is a super set containing all possible tags, the process ensures that no tags are new in validation or test sets.

Algorithm 2 Validation and Test Split Creation

Input: A held-out dataset \mathcal{D}_{ho} , a fraction value γ where $0 \leq \gamma \leq 1$, an integer N indicating number of seeds.

Output: $\mathcal{D}_{\text{valid}}, \mathcal{D}_{\text{test}}$ splits

Initialize: $count = 0, bestScore = \gamma + 1$

while $count < N$ **do**

$seed = getSeed()$

 Shuffle \mathcal{D}_{ho}

$\mathcal{D}_{\text{valid}} = \mathcal{D}_{\text{ho}}[0 : |\mathcal{D}_{\text{ho}}| \times \gamma]$

$\mathcal{T}_{\text{valid}} = \text{set of tags in } \mathcal{D}_{\text{valid}}$

$\mathcal{D}_{\text{test}} = \mathcal{D}_{\text{ho}}[|\mathcal{D}_{\text{ho}}| \times \gamma : |\mathcal{D}_{\text{ho}}|]$

$\mathcal{T}_{\text{test}} = \text{set of tags in } \mathcal{D}_{\text{test}}$

if $\mathcal{T}_{\text{valid}} \neq \mathcal{T}_{\text{test}}$ **then**

 continue

end if

for all T in $\mathcal{T}_{\text{valid}}$ **do**

$\gamma_T = \frac{\text{\#samples in } \mathcal{D}_{\text{valid}} \text{ with tag } T}{\text{\#samples in } \mathcal{D}_{\text{test}} \text{ with tag } T}$

end for

$\mu = geoMean(\{\gamma_T\}_{T \in \mathcal{T}_{\text{valid}}})$

if $|\gamma - bestScore| > |\gamma - \mu|$ **then**

$bestScore = \mu$

 save current split $\{\mathcal{D}_{\text{valid}}, \mathcal{D}_{\text{test}}\}$

$count = count + 1$

end if

end while

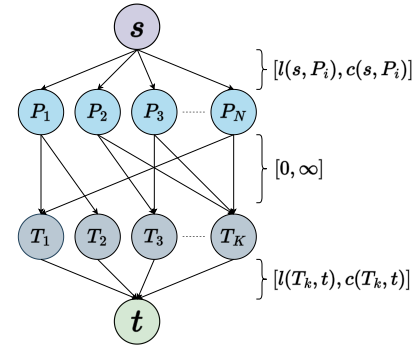


Figure 5.19: Flow network of for validation-test dataset creation.

Here the aim is to put bounds to the number of submissions selected for each problem and tag. Let p_i and t_k be the number of solutions for problem P_i and tag T_k , respectively. Let $G = (V, E)$ be a flow network (a directed graph) with the set of vertices $V = \{s, P_1, \dots, P_N, T_1, \dots, T_K, t\}$. Here s and t represent the source and sink of the flow network. Also $l(u, v), c(u, v)$ represents the lower and upper capacity of edge connected from u to v . Figure 5.19 shows the flow network used for the validation-test dataset creation. For each edge $e \in E$, the lower capacity $l(e)$ and upper capacity $c(e)$ are defined as follows.

1. Initialize $E = \emptyset$.
2. For each problem P_i , add edge (s, P_i) to E and assign $l(s, P_i) = \min(m_p, p_i)$ and $c(s, P_i) = \min(x_p, p_i)$, where m_p and x_p respectively refer to the minimum and maximum submissions to choose per problem if available with $m_p \leq x_p$, thus $0 \leq l(s, P_i) \leq c(s, P_i)$.

3. For each tag T_k , add edge (T_k, t) to E and assign $l(T_k, t) = \min(m_t, t_k)$ and $c(T_k, t) = \min(x_t, t_k)$, where m_t and x_t respectively refer to minimum and maximum submissions to choose per tag if available with $m_t \leq x_t$, thus $0 \leq l(T_k, t) \leq c(T_k, t)$.
4. For each P_i and T_k , add (P_i, T_k) to E if P_i has a tag T_k , and assign $l(P_i, T_k) = 0$, $c(P_i, T_k) = \infty$.

Now solution of the circulation problem can be directly adopted to find a flow $f : E \rightarrow \mathbb{Z}^3$ that satisfies:

$$\forall e \in E, l(e) \leq f(e) \leq c(e) \quad (5.1)$$

$$\forall u \in V, \sum_{v \in V} f(u, v) = 0 \quad (5.2)$$

In this case, f denotes a feasible flow when the above constraints are satisfied for some G . For each $e \in E$, $f(e)$ represents the following:

1. $f(s, P_i)$ denotes the number of submissions to be picked from problem P_i .
2. $f(T_k, t)$ denotes the number of submissions to be picked from tag T_k .
3. $f(P_i, T_k)$ denotes the number of submissions to be picked from P_i that has a tag T_k .

Here, $\sum_{k=1}^K f(T_k, t) = \sum_{i=1}^N f(s, P_i)$ is the total number of submissions selected, which can be controlled in a balanced way by setting the control variables m_p, m_t, x_p , and x_t . Now for the details about selection method of this control variables and their value for different tasks, along with a comparison to a random data selection strategy.

5.4.3 Control Variable Selection

Let M be the number of submissions to be selected for any set of submissions. Call (m_p, m_t, x_p, x_t) a valid tuple if the flow network has a feasible flow for the circulation problem defined in (5.4.2). Let $d = \lfloor (\sum_{i=1}^N f(s, P_i) - M)^2 / \Delta \rfloor$, representing the squared difference between number of submissions required and the submissions selected for the flow and Δ reduces the resolution of difference between number of submissions required and number of submissions selected. Here d defines a boundary from M where it has been

³ \mathbb{Z} denotes the set of integers.

allowed to choose an expected solution with $m_p, m_t, x_p,$ and x_t . Finally, the lexicographical ordering $(-d, m_t, -x_t, -x_p, m_p)$ is used to find the largest element in the collection of valid tuples which always exist if the search space is limited to a finite set. The largest element in this ordering depicts the nearest (close to M) selection of submissions that maximizes the minimum number of submissions per tag m_t . When there are many solutions with the same $(-d, m_t)$, reducing the maximum number of samples per tag, x_t has been prioritized. Similarly, x_p and m_p were also prioritized as defined in the lexicographical ordering.

5.4.3.1 Search Techniques

1. It was manually checked that $(m_p, m_t, x_p, x_t) = (1, 1, 1000, 1000)$ is a valid tuple for any set of submissions that were processed and $\Delta = 1000$ was chosen.
2. In Tag classification task (section 5.5.1.1) and Code compilation task (section 5.5.1.2), M is 2000, 10000 for any language for validation, test split respectively. For Code translation (section 5.5.2.3) M was 400, 2000 for the same.
3. Search largest tuple $(-d_1, m_{t_1}, -x_{t_1}, -x_{p_1}, m_{p_1})$ where $m_{t_1} \in \{1, 6, 11, \dots, 496\}$, $m_{p_1} \in \{1, 2, 3, \dots, 19\}$ and $x_{p_1} = x_{t_1} = 1000$. Since $(m_p, m_t, x_p, x_t) = (1, 1, 1000, 1000)$ is a valid solution, hence the set of valid tuples is nonempty. Let f_1 be the flow for the flow network defined for $m_{t_1}, -x_{t_1}, -x_{p_1}, m_{p_1}$. Let $f_{P_1} = \max_{1 \leq i \leq N} f_1(s, P_i)$, $f_{T_1} = \max_{1 \leq k \leq K} f_1(T_k, t)$ be the maximum flow through edges from s to P_i , and same through edges from T_k to t .
4. Now again search the largest tuple $(-d_2, m_{t_2}, -x_{t_2}, -x_{p_2}, m_{p_2})$ where $m_{t_2} \in \{m_{t_1}, m_{t_1} + 1, \dots, m_{t_1} + 49\}$, $x_{t_2} \in \{f_{T_1} - 100, f_{T_1} - 80, \dots, f_{T_1}\}$, $x_{p_2} \in \{f_{P_1} - 5, f_{P_1} - 4, \dots, f_{P_1}\}$, $m_{p_2} \in \{m_{p_1}, m_{p_1} + 1\}$. Since $m_{t_1}, f_{T_1}, m_{p_1}, f_{P_1}$ is included a solution is found in this step too. Define f_{P_2}, f_{T_2} similar to previous step.
5. Finally search the largest tuple $(-d_3, m_{t_3}, -x_{t_3}, -x_{p_3}, m_{p_3})$ where $m_{t_3} = m_{t_2}, x_{t_3} \in \{f_{T_2} - 100, f_{T_2} - 99, \dots, f_{T_2}\}$, $x_{p_3} = x_{p_2}, m_{p_3} = m_{p_2}$.

While it is not an exhaustive search, it prioritizes minimizing $x_t - m_t$ over $x_p - m_p$.

5.4.4 Results

Here is the performance of data selection using circulation problem technique with randomly selecting equal number of submissions for validation and test sets of all languages and

measured the skew $\tilde{\mu}_3$, and standard deviation σ of the distribution of tags in the selected data. Here lower value of $|\tilde{\mu}_3|$ means more symmetric distribution. On the other hand, a lower value of σ represents that the number of samples in each tag are closer to the mean.

Table 5.2: Comparison of skew and standard deviation of tags using circulation problem technique and random data selection (lower value is better).

Language	Skew, $\tilde{\mu}_3$				Std. deviation, σ			
	Validation		Test		Validation		Test	
	Random	Circ.	Random	Circ.	Random	Circ.	Random	Circ.
Tag Classification								
C	2.778	2.499	2.848	2.440	249.161	213.849	880.881	772.549
C++	2.405	1.873	2.315	1.655	233.530	157.889	1154.538	751.023
Python	2.731	2.365	2.689	2.173	265.193	240.248	1125.133	992.904
Java	2.652	1.990	2.545	2.050	258.587	207.881	1175.790	972.703
C#	3.066	2.598	2.971	2.506	314.219	291.813	846.426	760.069
Code Translation								
C	2.744	2.455	2.941	2.332	117.298	99.261	267.214	215.881
C++	2.424	2.112	2.287	1.565	131.632	120.979	243.100	150.498
Python	2.533	2.379	2.635	2.294	123.710	110.076	271.219	237.179
Java	2.558	2.208	2.605	1.827	134.314	114.840	259.510	193.211
C#	3.147	2.532	2.943	2.395	103.838	96.747	250.049	220.615
PHP	2.506	2.744	2.520	2.730	59.321	59.877	270.582	278.530
Rust	2.520	2.393	2.534	2.311	59.269	60.253	269.352	264.507
Go	2.807	2.359	2.676	2.424	72.415	66.666	266.565	254.986
Javascript	2.611	2.611	2.473	2.473	64.090	64.090	246.483	246.483
Ruby	2.875	2.686	2.968	2.762	74.153	70.760	280.000	271.539
Kotlin	2.865	2.576	3.108	2.534	59.765	56.114	266.430	257.155

Table 5.3: Size of the datasets for each task and the evaluation metrics.

Task Type	Task	Lang	Train	Validation	Test	Metric
Classification	Tag classification	11	5,500,913	19,087	76,498	mcc
	Code compilation	11	19,915,150	6,394	30,388	accuracy
Generative	Program synthesis	11	5,545,785	108	979	pass@k
	Code translation	11	5,548,683	7,194	20,890	pass@k
	Automatic program repair	11	4,677,164	5,224	18,028	pass@k
Retrieval	Code-Code retrieval	17	45,270	2,335	9,508	Recall,Prec@k
	NL-Code retrieval	17	55,924	2,780	11,157	

5.5 Tasks in xCODEEVAL with Statistics

xCODEEVAL features two classification, three generative, and two retrieval tasks. Table 5.3 summarizes the sizes of the datasets for each task and the evaluation metrics. For *Program Synthesis* train data P_i, S_k comes from 7514 problems of 11-17 languages where the input for validation and test data is only natural language text (problem description) independent of programming languages. For all other tasks, validation and test samples are reported for total number of languages. For both of the sub-tasks of tag classification and automatic code repair total number of samples are same. In contrast table 5.4 gives a more detailed breakdown of the tasks per language. It should be noted that the validation and test splits for *Program Synthesis* are same across all the languages as they solve the same problems (to produce solution in different languages from same *nl* description). From here on ‘sample’ would mean a submission or a problem depending on the task that has survived the data processing step described in section 5.4.

5.5.1 Classification Tasks

5.5.1.1 Tag Classification

This task is formulated as a multi-label classification problem in two settings: Code-to-Tag (Code2Tag) and Problem Description-and-Code to Tag (DesCode2Tag). In Code2Tag, given a code C in any language, the task is to predict the corresponding tag set \mathbb{T} . In DesCode2Tag, the natural language problem description is also given as input in addition to the code. The performance difference between *Code2Tag* and *DesCode2Tag* settings can suggest the if the problem description can help models to identify the problem tags (i.e., the type of solution needed).

Table 5.4: Dataset statistics per language and task.

Split	C	C#	C++	Go	Java	Javascript	Kotlin	PHP	Python	Ruby	Rust	Total
Tag Classification												
Train	178509	79219	3716097	25645	704270	15744	49449	6313	679710	15253	30704	5500913
Validation	1732	2265	2027	1636	2003	1616	1733	905	2031	2215	924	19087
Test	6338	6136	10002	6548	9254	6467	6932	3621	8641	8862	3697	76498
Code Compilation												
Train	503458	170407	15147814	53561	2007940	36949	104970	18099	1793141	26362	52449	19915150
Validation	1000	1000	1000	212	1000	454	482	102	1000	50	94	6394
Test	5000	5000	5000	814	5000	1676	1940	392	5000	242	324	30388
Program Synthesis												
Train	179706	79773	3748656	25789	708265	15947	51903	6412	683199	15380	30755	5545785
Validation	108	108	108	108	108	108	108	108	108	108	108	108
Test	979	979	979	979	979	979	979	979	979	979	979	979
Code Translation												
Train	179716	79786	3751303	25796	708408	15951	51950	6413	683222	15382	30756	5548683
Validation	788	759	1083	477	986	415	433	378	896	508	471	7194
Test	1778	1810	2010	1838	1989	1661	2000	1764	1998	2002	2040	20890
Automatic Program Repair												
Train	135430	37104	3412242	13106	575039	8878	16369	3649	462500	5166	7681	4677164
Validation	744	742	730	295	738	184	313	191	739	343	205	5224
Test	2005	2041	2080	1436	2080	646	1990	1166	2080	1595	909	18028

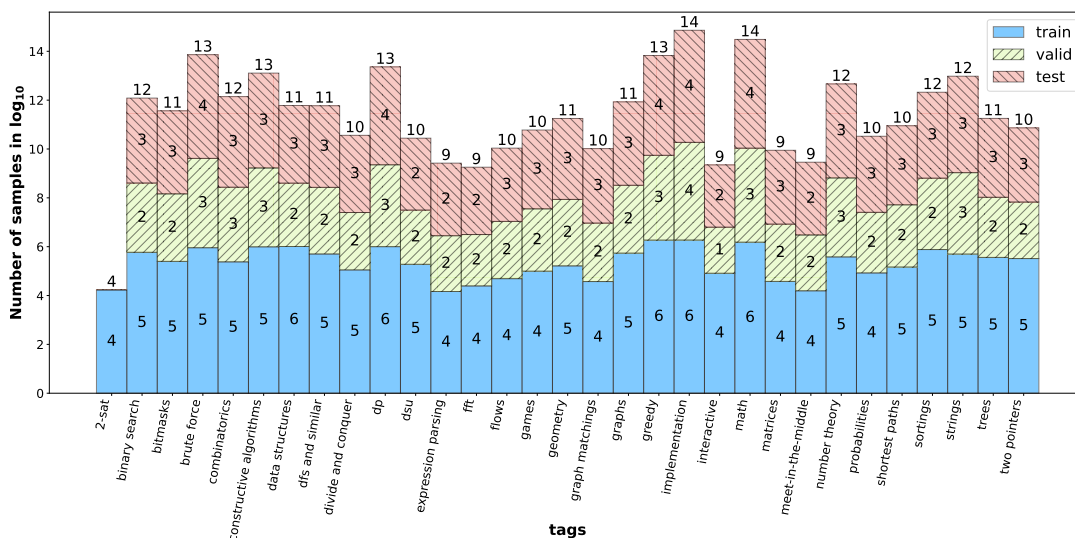


Figure 5.20: Tag distribution in *tag classification* task in xCODEEVAL.

For these tasks, the split for validation and test is done with a ratio of 1 : 5 (i.e., $\gamma = 0.2$) using algorithm 2. To get the final $\mathcal{D}_{\text{valid}}$ and $\mathcal{D}_{\text{test}}$ with a feasible number of samples, the flow network-based data selection approach with the details of control variable settings presented in section 5.4.

The distribution of the samples according to the tags is presented in figure 5.20. In XCODEEVAL, often multiple tags are assigned to the same problem as each problem can be solved in multiple ways or with a combination of multiple techniques (e.g. figure 5.1). This dataset is further broken into a language-specific tag classification task, in which each programming language has its own *Code2Tag* and *DesCode2Tag* settings.

5.5.1.2 Code Compilation

Given a code C in a language L and its compiler or interpreter version B , the *code compilation* task is to decide whether the code compiles or not. The validation and test splits are created using a modified version of algorithm 2 that balances the partition based on the compilation outcome of the code instead of the tags of the problem that the code belongs to with a ratio γ of 1 : 5. Then a simplified version of the circulation problem is used to prevent too many codes coming from a single problem, and also to ensure a balanced output distribution. The details of hyper-parameter settings of the circulation problem technique are presented in section 5.4. In the flow network construction, tags $\{T_k\} = \{true, false\}$ as *true* if the code compiles or not. Furthermore true and false examples are present in equal numbers in both validation and test dataset.

5.5.2 Generative Tasks

XCODEEVAL proposes three generative tasks which require a global understanding of programming languages. For the evaluation of generative tasks, execution-based evaluation is promoted instead of lexical similarity. All the generative tasks are evaluated using ExecEval execution engine. XCODEEVAL provides complete unit tests for all problems in the validation and test dataset which also satisfy the conditions of the input-output specification of the problem.

5.5.2.1 Program Synthesis

Given a problem described in natural language, program synthesis task is to write a program that solves the problem. We can express each sample in the dataset as a tuple (C, P, l, L) , where C denotes a solution code written in a programming language L for the problem P , and l denotes the compiler/interpreter version of the code. All code samples in the dataset are unique and marked as a correct solution (PASSED outcome) to the problem. The

validation and test splits are created from the heldout problems using algorithm 2 with a ratio (γ) of 1 : 9. The generated code is judged based on executions on the unit tests.

5.5.2.2 Automatic Program Repair (APR)

We consider APR as a task to synthesize a fix for a detected program bug. We create a bug-fix pair by matching a buggy code (1-5 execution outcome in section 4.2.2) with a PASSED solution. Given a bug-specific defect, the objective of this task is to generate a correct fix that passes all the unit tests.

Let $\mathbb{C} = \{C_1, \dots, C_m\}$ be the set of programs submitted by a participant in a chronological order in order to solve a specific problem P . Some of these submissions can be ‘buggy’, while some can be PASSED. We create the ‘bug-fix’ pairs from \mathbb{C} as follows.

1. We iterate over \mathbb{C} and mark the PASSED ones as ‘fixed’. Let C_j^* is one such case.
2. For each buggy submission that was made before C_j^* , we measure its lexical similarity with C_j^* and select the one (say C_k where $k < j$) with the highest similarity score to pair it with C_j^* and form a bug-fix pair (C_k, C_j^*) . We use `difflib`⁴ to measure the similarity.
3. With each bug-fix pair (C_k, C_j^*) , we also include the corresponding problem description P and execution outcome V_k (section 4.2.2) of C_k .
4. The tuple (C_k, C_j^*, P, V_k) represents a sample in our APR task.

We repeat this process for each participant and problem to create the final APR dataset. As reported in table 5.3, it comprises more than 5M practical bug-fix pairs and supports 11 programming languages. For data selection in APR, we considered execution outcome (section 4.2.2) as *tags* in the network flow construction (section 5.4.2).

Due to the large input specification of the APR task, sometimes the input sequence length becomes too large. However, we have not compromised the benchmarks by selecting only small sequence length samples but rather keep them as challenging tasks for the language models.

⁴<https://docs.python.org/3/library/difflib.html>

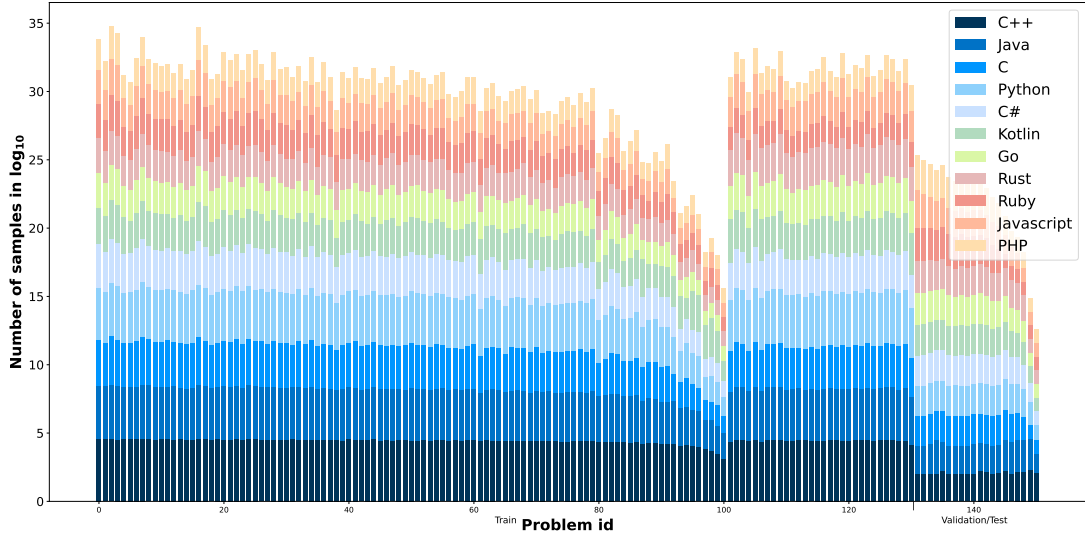


Figure 5.21: Distribution of samples across all problems in the train, validation, test splits for all languages in the code translation task.

5.5.2.3 Code Translation

Each sample in the code translation data can be expressed as a tuple (\mathcal{C}, P, l, L) , where \mathcal{C} denotes a set of solution codes in a programming language L for the problem P , and l denotes the compiler/interpreter version of the code. All codes in set \mathcal{C} are unique and guaranteed to be marked as a correct (PASSED outcome) solution to the problem by the compiler/interpreter.

The validation and test splits are created from the held-out problems using algorithm 2 with a ratio (γ) of 1 : 5, and employ the data selection method with flow network (section 5.4) to have a practical evaluation setup while ensuring a balanced distribution over problems and tags. Figure 5.21 shows the distribution of the machine translation tasks.

5.5.3 Code Retrieval

Code retrieval tasks typically aim to measure the mere semantic relatedness between a natural language (NL) query and a programming language (Code) code. However, a code that is relevant, can still be buggy and thus be misleading (see an example in figure 5.22). The candidate code in the left of figure 5.22 has a bug highlighted in red and that in the right has a fix highlighted in green. Both of the proposed NL-Code and Code-Code retrieval tasks ensure differentiating between them and pose a more challenging task

that aims to comprehend both the semantic and logical similarity. In view of this, we propose two new and more challenging retrieval tasks in our benchmark, which require a deeper understanding of the NL query and code. In particular, we propose NL-Code and Code-Code retrieval tasks that involve identifying a *correct code* from a large pool of candidates containing similar codes. In both tasks, for each programming language, we aggregate all the submitted codes and their test cases to create a retrieval corpus and a testbed for evaluating their correctness against test cases. The datasets for the subtasks and the evaluation schema are discussed below.

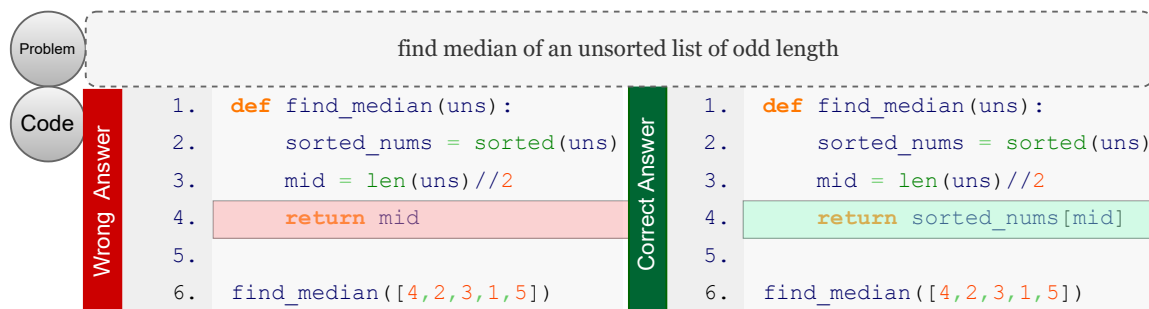


Figure 5.22: A code retrieval example.

5.5.3.1 NL-Code Retrieval

This task involves matching an NL problem description to the most relevant and correct code from a pool of candidates. An example of an NL description and its corresponding codes are showed in figure 5.1. To gather data for this task, only the instances where the NL description is valid and there is at least one correct solution code (i.e., with execution outcome PASSED) has been used. For an NL problem description, all the correct solutions has been considered as positive examples and all the wrong (or buggy) solutions as the negative examples.

5.5.3.2 Code-Code Retrieval

Given an input code (as query), this task involves finding functionally similar codes (i.e., passes the same set of test-cases, defined in section 4.1) from a collection of candidates. We ensure that the query code solves a specific problem (i.e., correct solution without any detected bugs) and evaluate whether the retrieved candidate also solves the same problem or not. To collect data for this task, we only consider the programming problems which

Table 5.5: Statistics of retrieval datasets for both *nl-code* and *code-code*.

Lang	Subtask	Train			Dev			Test Size	Retrieval Corpus Size
		Size	Pos	Neg	Size	Pos	Neg		
C#	NL-Code	4,886	75,478	55,709	215	6,623	4,971	862	251,147
	Code-Code	4,404	69,100	55,015	202	5,895	4,956	813	
C++	NL-Code	6,192	613,747	609,150	283	27,152	26,851	1,134	18,212,508
	Code-Code	6,192	555,383	609,150	283	24,796	26,851	1,134	
C	NL-Code	5,205	149,189	147,116	221	11,688	11,717	887	787,516
	Code-Code	4,398	122,904	146,113	205	9,514	11,706	830	
D	NL-Code	3,367	7,639	3,680	137	357	142	550	15,984
	Code-Code	1,974	4,272	2,745	82	220	119	305	
Go	NL-Code	3,768	25,692	19,006	170	1,483	784	683	68,237
	Code-Code	3,093	21,819	18,126	151	1,254	757	574	
Haskell	NL-Code	3,178	15,159	7,152	178	2,200	947	715	44,682
	Code-Code	2,308	11,879	6,378	163	1,894	931	615	
Javascript	NL-Code	2,616	15,636	13,733	139	1,344	1,352	559	56,917
	Code-Code	1,990	12,845	12,705	120	1,161	1,313	449	
Java	NL-Code	5,941	394,341	375,937	264	18,271	16,670	1,057	2,523,044
	Code-Code	5,802	321,150	375,696	259	14,575	16,643	1,027	
Kotlin	NL-Code	4,023	46,559	25,645	167	1,939	1,115	669	121,569
	Code-Code	3,242	39,879	24,993	134	1,671	1,086	528	
Ocaml	NL-Code	1,429	2,334	1404	100	231	141	401	7,012
	Code-Code	487	905	760	51	131	109	180	
PHP	NL-Code	1,969	6,379	8,977	141	902	837	567	29,179
	Code-Code	1,183	4,377	6,796	100	724	745	403	
Pascal	NL-Code	4,441	113,381	105,327	221	10,310	8,693	887	494,473
	Code-Code	3,958	97,329	104,520	213	8,662	8,689	848	
Perl	NL-Code	1,280	3,911	1,964	106	565	346	427	11,035
	Code-Code	680	2,631	1,536	66	459	313	319	
Python	NL-Code	4,941	317,696	285,609	224	18,097	16,061	896	2,290,854
	Code-Code	4,747	267,046	285,291	221	14,884	16,059	874	
Ruby	NL-Code	2,357	15,274	7,333	168	2,498	894	676	44,934
	Code-Code	1,749	12,750	6,738	155	2,229	882	592	
Rust	NL-Code	3,864	30,696	14,962	140	750	310	560	59,829
	Code-Code	3,066	26,798	14,329	106	610	292	433	
Scala	NL-Code	2,558	7,863	5,226	149	874	469	600	24,780
	Code-Code	1,529	5,270	4,092	125	725	448	454	

have at least *two* correct code solutions that pass all the corresponding test cases (i.e., with execution outcome `PASSED`). From each of these problems, we randomly choose one correct solution as a (code) query and pair it with the other correct solutions as positive examples and the corresponding wrong solutions (i.e., with execution outcome `WRONG`

ANSWER) as negative examples.

Retrieval Corpus Metadata and Evaluation Protocol: We preserve the problem specifications and execution outcomes (e.g., PASSED, WRONG ANSWER) for each candidate code in our retrieval database. For both the NL-code and code-code retrieval tasks, we use this information to determine the correctness of a retrieved code, checking if that solves the same programming problem as the input query by passing all its unit tests or not.

Evaluation Metrics: We evaluate the retrieval performance in terms of accuracy@ k also called top- k accuracy. We consider $k \in \{1, 10, 100\}$.

Our retrieval benchmark has 17 programming languages and our training dataset is the largest that provides annotations of similar codes that are found logically equivalent or correct based on the passing of test cases. For evaluation purposes (i.e., for test sets), we release the input problem description (in NL-Code) or the input code (in Code-Code) only and keep all other metadata confidential. Covered programming languages and their data statistics in both tasks are summarized in table 5.5. Here |Size| denotes the number of instances in the datasets. For each train/validation instance we provide multiple positive and negative examples and |Pos| and |Neg| refer to the that total number of positive and negative annotations.

Chapter 6

Model Training and Results

We now delve into the intricate facets of a Dense Passage Retrieval (DPR) model, elucidating the methodology employed in its training alongside a comprehensive exposition of the ensuing evaluation results. The DPR model, a sophisticated neural architecture designed for information retrieval, assumes a pivotal role in the encoding and indexing of queries within the retrieval system. Subsequently, these encoded queries undergo processing by FAISS (Facebook AI Similarity Search), a state-of-the-art similarity search library introduced by Johnson et al. [34]. FAISS operates on the principle that documents with lower ranks in the retrieval process correspond to a higher degree of similarity to the input query, thereby facilitating the extraction of relevant documents in accordance with their hierarchical rank order. This chapter thus serves as a comprehensive guide, offering a refined understanding of the intricacies underlying the DPR model's training and evaluation, culminating in a judicious elucidation of the interplay between DPR and FAISS in the retrieval of pertinent documents.

6.1 Model Training

6.1.1 Model Architecture

DPR uses two separate BERT-based [16] encoders $E_P(\cdot)$, $E_Q(\cdot)$ for retrieval corpus codes (passages) and search codes (questions) respectively which maps a natural text to a vector in \mathbb{R}^d . During inference E_p is used to build the index for all M codes of the retrieval corpus, then for each search codes (question) q , k passages p_i are retrieved from the corpus such that similarity between p_i, q denoted by $\text{sim}(p_i, q)$ are maximized. As Karpukhin et al. [36] have shown, after lot of studies are available on possibilities for the similarity function, it is found that they perform closely which motivates the use of simpler function i.e. inner product

between the embeddings, more formally the similarity between passage p and question q is defined as

$$\text{sim}(q, p) = E_Q(q)^T E_P(p).$$

This also implies that the model should learn to embed codes in such a way that the codes labelled as similar in the dataset should produce embeddings that maximize (6.1.1).

In this work the pre-trained language model *CodeBERT* [18] and *Starencoder* [46] were used as the encoder implementation for both the question and passage encoders $E_Q(\cdot)$, $E_P(\cdot)$. The 125M parameter *CodeBERT*, and *Starencoder* models are masked language models pre-trained on *CodeSearchNet* [30], and *The Stack* [40] datasets, respectively that has been specifically designed for tasks related to understanding and generating code. *CodeBERT* was developed by Microsoft Research and is based on the BERT [16] architecture, which has achieved strong results on a variety of natural language processing tasks. The model has been pre-trained on a large dataset and has already learned dense representations encapsulating a lot about the structure and patterns of natural language, including code-related text. Similarly *Starencoder* was trained by the bigcode project with much larger dataset. By using the these as the encoders, we can leverage this pre-trained knowledge to improve the performance of our indexer. This choice forces $d = 768$, and the sequence length of 512 for *CodeBERT* and 1024 for *Starencoder*, the limit of number of tokens of the code that the model will process using CodeBERT’s own SentencePiece tokenizer, or Starencoder’s own GPT2Tokenizer.

During inference, given a question q at run-time, DPR derives its embedding $v_q = E_Q(q)$ and FAISS retrieves the top k passages with embeddings closest to v_q with closeness measured by eq. (6.1.1).

6.1.2 Training Dataset

To achieve the goal the training dataset is structured as a collection of m samples

$$\mathcal{D} = \{(q_i, p_i^+, p_{i,1}^-, \dots, p_{i,n}^-)\}_{i=1}^m$$

, where q_i represents a code and p_i^+ denotes a code that is similar to q_i and $p_{i,j}^-$ are codes that are not similar to q_i . Then the loss function is defined as

$$L(q_i, p_i^+, p_{i,1}^-, \dots, p_{i,n}^-) = -\log \frac{e^{\text{sim}(q_i, p_i^+)}}{e^{\text{sim}(q_i, p_i^+)} + \sum_{j=1}^n e^{\text{sim}(q_i, p_{i,j}^-)}}.$$

Note that minimizing this loss function implies learning an encoding E_Q, E_P such that $\text{sim}(q_i, p_i^+) \gg \text{sim}(q_i, p_{i,j}^-)$. This is the *Inverse Cloze Test* (ICT) objective function eluded in section 3.2.

Informally the positive passages are annotated to be similar to question and negative passage is annotated different from the question. Interestingly enough the model is not directly fed dataset in this representation, rather the positive passages in other samples of the batch are assumed as negative passages at training time along with the negative passages annotated in the sample. Thus the input representation is $\mathcal{D}_{\text{input}} = \{(q_i, p_i^+, p_i^-)\}_{i=1}^m$ is fed to the model. This mixing of annotations from different samples of the batch is aptly named by Karpukhin et al. [36] as *in batch negatives*. In section 6.3, more will be covered on the effects of *in batch negatives*. Next section covers in greater detail, the construction xCODEEVAL retrieval dataset from raw data as introduced in section 5.5.3.

6.1.3 Data Preparation

Here we create samples with question, positive and negative passage dataset from the collection of all submissions that have verdict *PASSED* as we can not make any strong claim about impure submissions that did not produce acceptable outputs for some problem. We use the functional similarity defined for both *nl-code* and *code-code* case to pair up questions and passages. The NL-Code and Code-Code datasets were prepared by making a 20:1:5 split for train, validation, and test split as discussed in section 5.5.3 ensuring a disjoint split of problems between train, and validation and test dataset. Furthermore the validation and test split is derived from the heldout set discussed in section 5.4.2. The main difference between NL-Code and Code-Code dataset is that of the question as in NL-Code the question is the problem itself and in Code-Code a randomly chosen code for the problem is considered as the question.

Table 6.1: List of major hyperparameters and their values for *CodeBERT* on left and *Starencoder* on right.

Attribute	Value (<i>CodeBERT</i>)	Value (<i>Starencoder</i>)
Sequence length	510	1024
Dropout	0.1	0.1
Initial learning rate	2×10^{-5}	2×10^{-5}
Optimizer	Adam	Adam
Train batch size	64/128	48
Dev batch size	64/128	48
Number of epoch	40	40
Gradient accumulation step	1	4

Finally, *CodeBERT* [18] and *Starencoder* [46] both were trained over the *Code-Code* datasets for all languages combined into a single dataset resulting in the following number of samples:

- Train data: 50,802 samples.
- Validation data: 2,636 samples.
- Test data: 10,378 samples.
- Corpus data: 25,032,700 codes.

And only *Starencoder* was trained for the *NL-Code* dataset as the sequence length of *CodeBERT* is too small for NL data. The *NL-Code* dataset for all languages were combined to a single dataset resulting in following number of samples:

- Train data: 61,898 samples.
- Validation data: 2,900 samples.
- Test data: 11,701 samples.
- Corpus data: 25,032,700 codes.

We train *CodeBERT* on 4 *NVIDIA A100 40GB* GPUs taking around 14 days to train each models up to 40 epochs each. *Starencoder* was trained on 4 *NVIDIA A100 80GB* GPUs and also took around 14 days for 40 epochs for each of *code-code* and *nl-code* retrieval.

6.2 Indexing the Corpus

The code corpus, constituting an expansive repository comprising approximately 25 million distinct codes, serves as the foundational dataset from which our advanced search engine derives its results. A preprocessing phase, as previously illustrated, involved the segregation of the corpus based on programming languages, subsequently encoding them with our highly sophisticated and rigorously trained language model.

To effectuate the embedding process, four *NVIDIA P100 32GB* GPUs were employed. This formidable hardware configuration was chosen to ensure optimal computational efficiency and expedited processing. The batching strategy for this embedding task was painstakingly calibrated, with **1024** as the batch size for the *CodeBERT* model and **512** for the *Starencoder* model, underscoring our commitment to precision and performance optimization.

The encoding operation culminated in the generation of embeddings, which encapsulate the subtle semantic representations of the codes. Executed across the aforementioned GPUs, this intricate procedure produced embeddings of exceptional depth and granularity. The resultant embeddings, embodying the semantic essence of each code snippet, were stored in a binary file, magnifying the overall corpus size to approximately ~ 73 gigabytes.

This exhaustive process, marked by the fusion of cutting-edge hardware resources and state-of-the-art language models, stands as a testament to the rigor and sophistication inherent in our approach to code corpus analysis. The resulting binary file, encapsulating the distilled knowledge within the embeddings, now serves as a foundational reservoir from which our search engine seamlessly retrieves and delivers insightful and contextually relevant results to the discerning user.

6.3 Model Evaluation

The primary hyperparameter under investigation in this study pertains to the training and validation batch size, a critical aspect in the optimization of machine learning models. As delineated in figure 6.1, a discernible performance disparity is evident between accuracy@ k scores of *CodeBERT* with batch size 64, 128 respectively at checkpoint 35. This observation underscores the efficacy of the *in batch negatives* strategy, wherein the utilization of a higher batch size (128) correlates with an augmented model performance, primarily attributed to

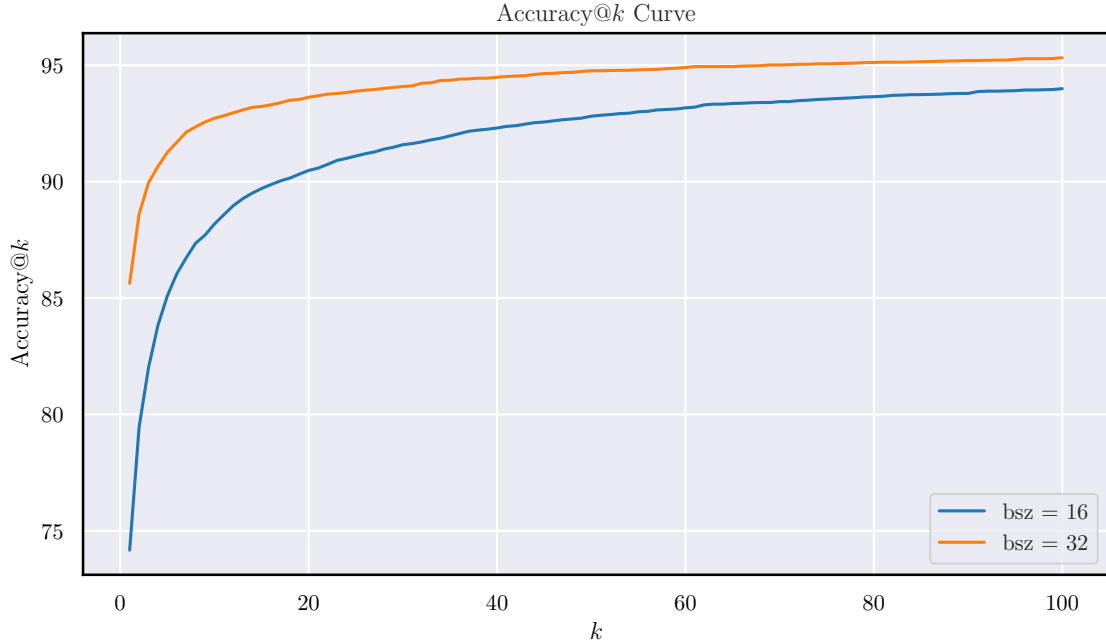


Figure 6.1: Comparison of performance between batch sizes.

Table 6.2: Summary of performance of both *CodeBERT* and *Starencoder* in both tasks.

Tasks	metric	C	C#	C++	D	Go	Haskell	Java	Javascript	Kotlin	Ocaml	PHP	Pascal	Perl	Python	Ruby	Rust	Scala	AVG.
CodeBERT																			
Code-Code (α)	Acc@k	61.39	51.79	40.83	61.33	72.93	58.76	47.46	76.03	66.83	66.26	72.21	56.92	65.71	59.53	69.55	43.39	69.52	61.20
Code-Code (γ)	Acc@k	68.23	72.68	71.42	46.41	65.76	60.58	76.73	52.87	55.26	35.3	44.88	67.55	40.32	72.47	63.32	42.84	59.57	58.60
Starencoder																			
Code-Code (α)	Acc@k	56.43	56.05	39.96	62.82	66.30	56.71	49.30	69.63	63.42	58.44	64.80	52.71	56.38	55.92	61.38	58.10	66.69	58.53
Code-Code (γ)	Acc@k	68.66	74.50	70.49	17.35	62.62	60.03	74.71	50.70	52.06	33.72	49.88	65.35	40.50	68.33	61.71	48.58	59.76	56.41
NL-Code	Acc@k	82.28	89.99	83.81	68.98	90.26	81.68	84.72	85.33	84.74	85.45	80.71	82.21	81.33	84.57	87.17	82.23	89.71	83.83

an increased density of negative annotations per question.

The inherent implication of this superiority prompts a deliberate focus on the model trained with a batch size of 128 for further in-depth analysis. Figure 6.2 systematically presents a comparative evaluation of *CodeBERT*'s performance across various checkpoints, denoted by n^{th} epoch intervals (here batch size is 128). This granularity in assessing performance variations at distinct training epochs elucidates the model's evolution and provides valuable insights into its convergence and stability. Consequently, the choice of an optimal checkpoint emerges as a crucial consideration for achieving the desired balance between training efficacy and computational efficiency.

The empirical analysis, delineated through the visual representations encapsulated in figures 6.3 to 6.6, provides a nuanced insight into the performance dynamics of *CodeBERT*

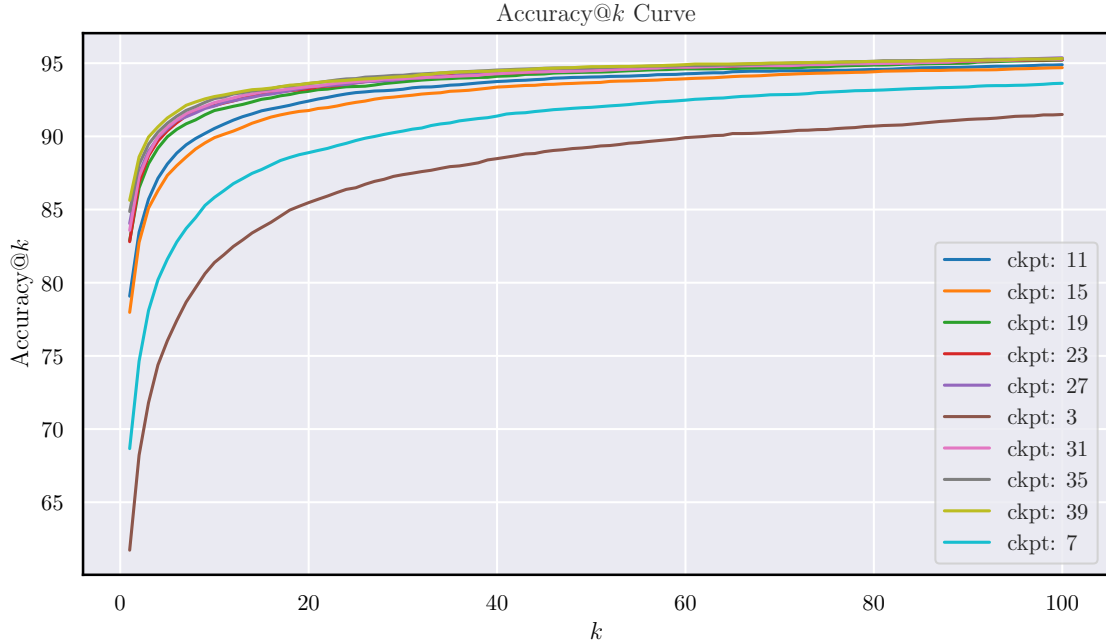


Figure 6.2: Comparison of performance between different epochs.

and *Starencoder* within the realm of the *Code-Code* dataset. These figures compare the accuracy@ k scores between all 17×17 languages, where for every pair of languages question codes come from one language and corpus of the second language is used for retrieval. Formally a cell (x, y) in the 17×17 matrix of top- k accuracy scores denotes the score for code queries from language x and the retrieval corpus of language y . Noteworthy is the discernible precision exhibited by both models in a mono-lingual context with average retrieval accuracy is 95.5. However, the efficacy of retrieval experiences a discernible degradation in a cross-lingual setting as the accuracy becomes 59.1, particularly pronounced in languages characterized by a paucity of training samples. For *starencoder* the average mono-lingual retrieval accuracy is 84.19, and average cross-lingual score is 56.93. This discerns the difficulty of the models in learning a multi-lingual understanding of codes.

In tandem, the evaluation on the *NL-Code* dataset underscores the exceptional performance of *Starencoder*, thus substantiating its prowess in natural language understanding within the code retrieval domain. The succinct summary presented in table 6.2 juxtaposes the performance metrics for both models across tasks, revealing a relative underperformance of *Starencoder* [46] vis-à-vis *CodeBERT* [18] fine-tuned on our retrieval tasks for $k = 100$. For *Code-Code*, (α) denotes the average score for codes of any given language as the corpus, similarly (γ) denotes average score for codes of any fixed language as query. For *NL-Code*,

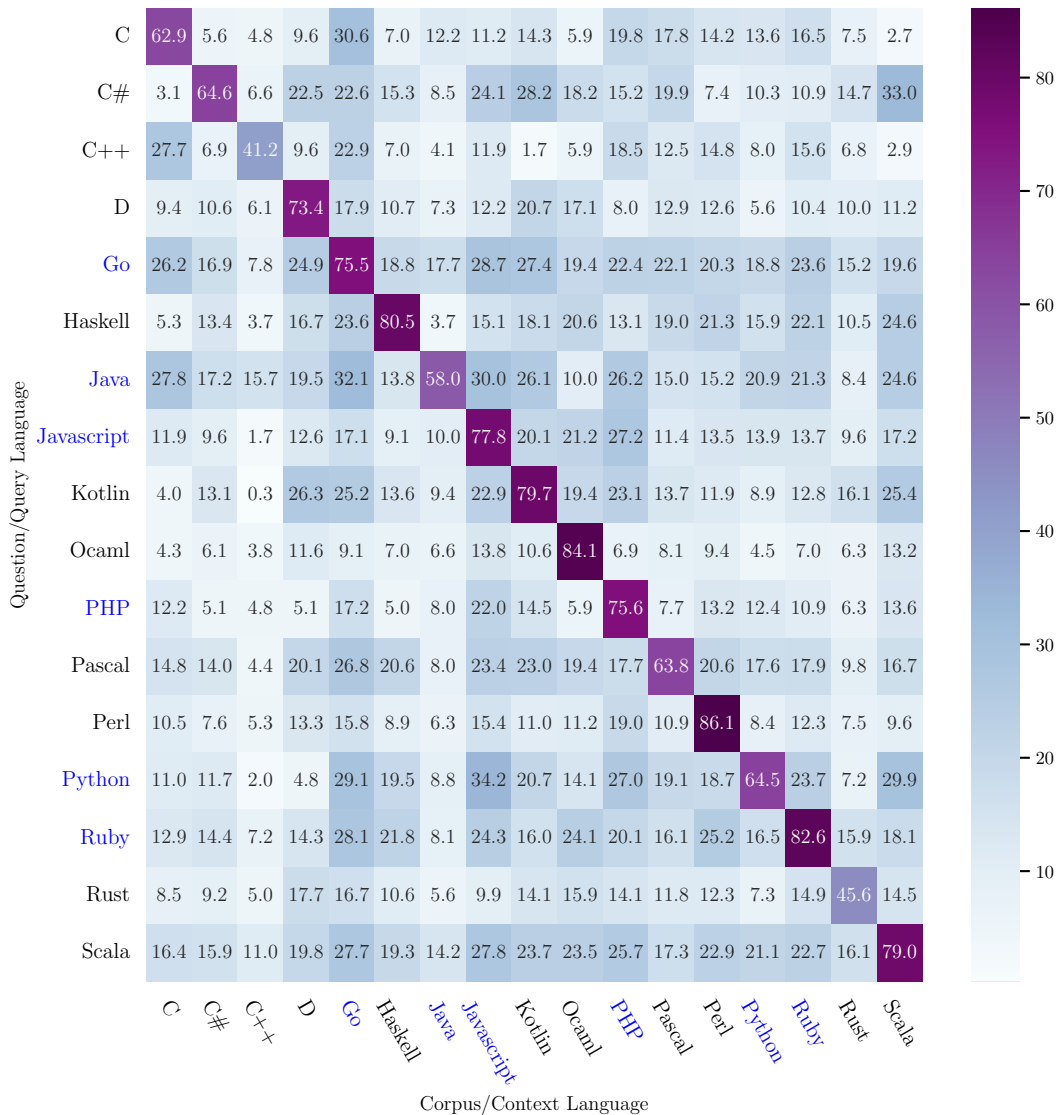


Figure 6.3: Comparison of top-1 accuracy across all language pairs for *CodeBERT*.

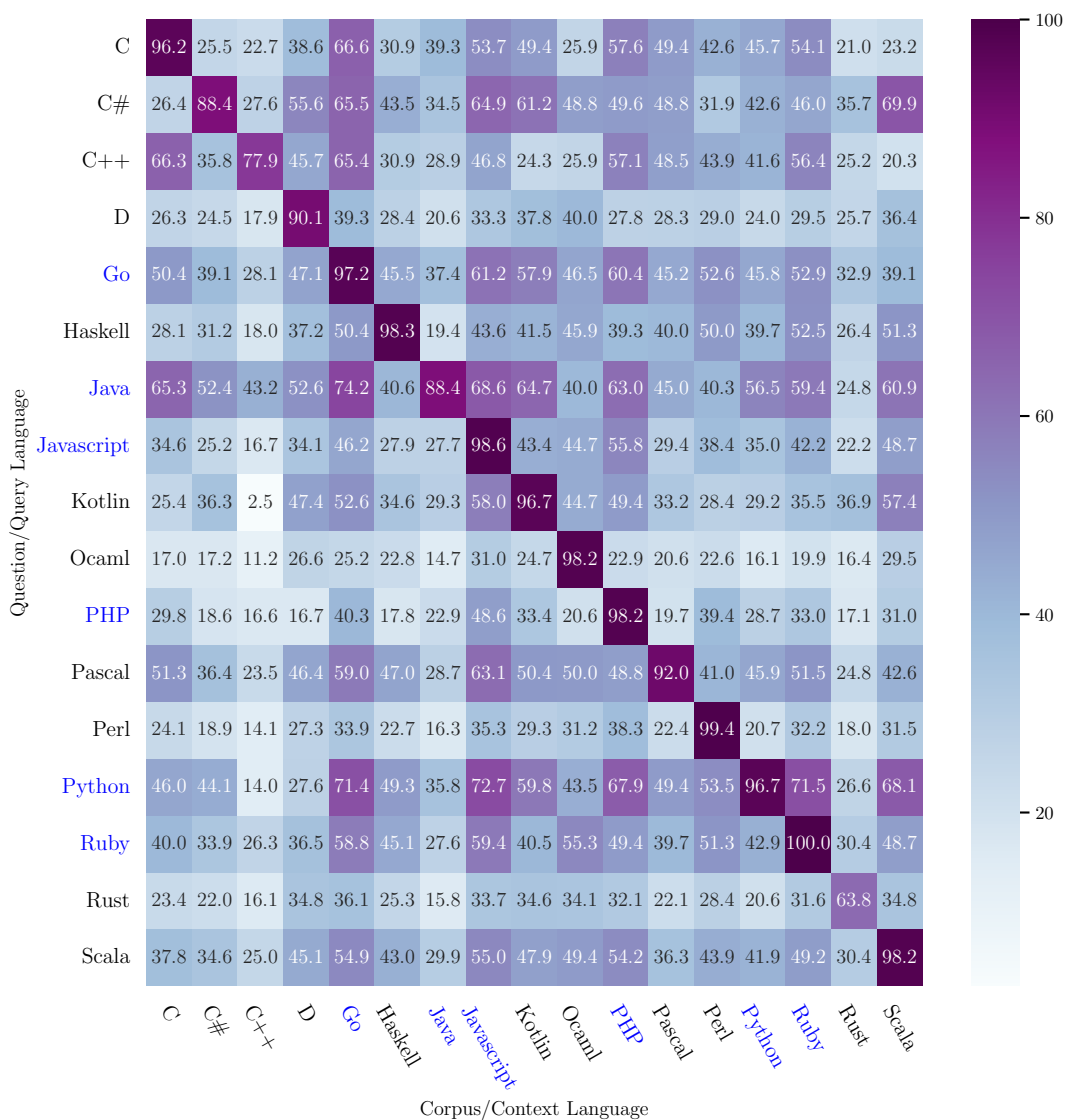


Figure 6.4: Comparison of top-10 accuracy across all language pairs for *CodeBERT*.

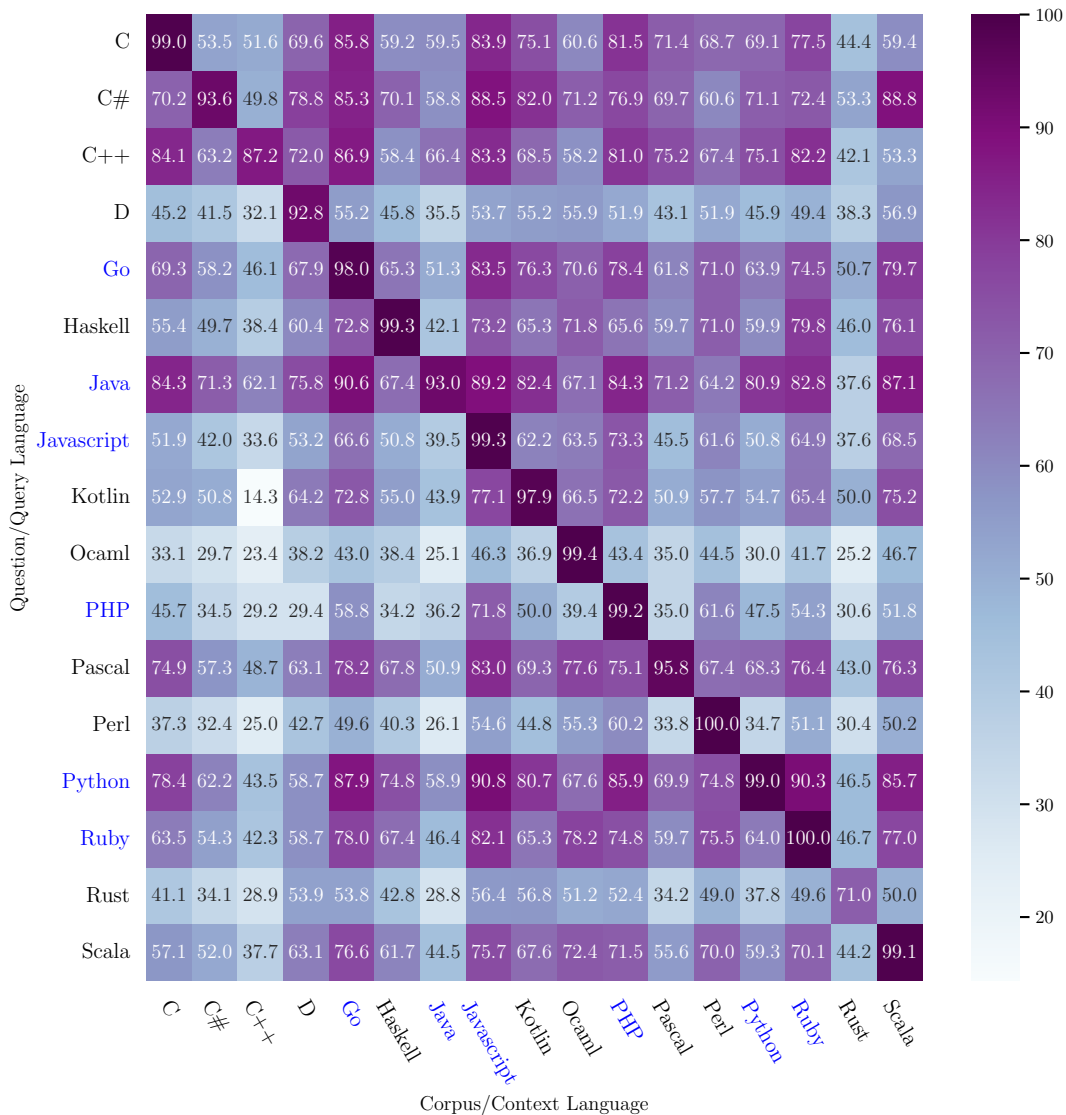


Figure 6.5: Comparison of top-100 accuracy across all language pairs for *CodeBERT*.

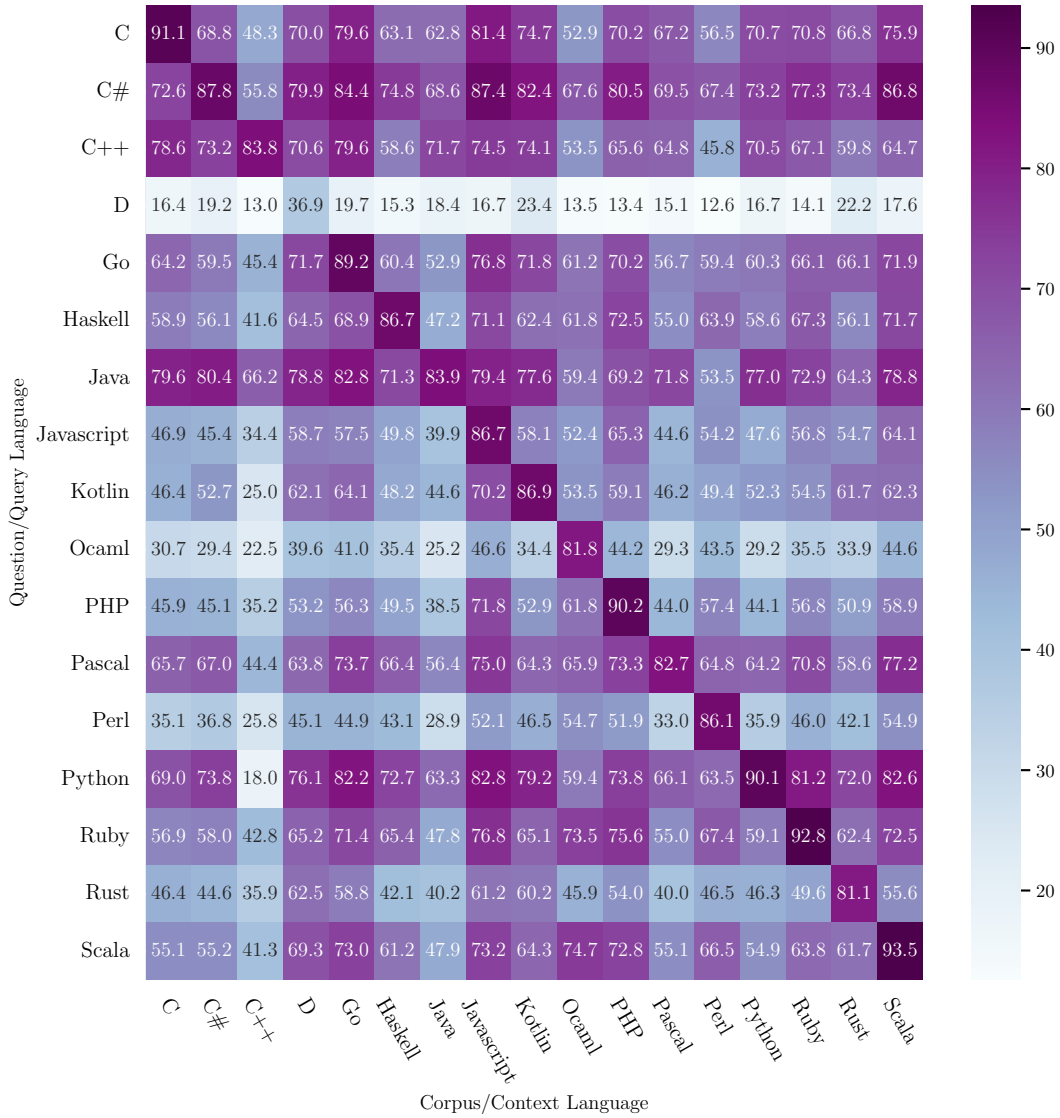


Figure 6.6: Comparison of top-100 accuracy across all language pairs for *Starencoder*.

the scores are reported for corpus of different languages.

Several salient observations emerge from this comparative analysis:

- The adoption of the *in batch negatives* strategy adversely impacts multi-lingual datasets, exacerbating the challenge by annotating disparate codes of distinct languages, yet sharing the same problem, as negatives. This is particularly evident in languages with fewer samples, such as C++, Java, and Python.
- The deleterious effect of in batch negatives is more pronounced in languages boasting a larger sample size, exemplified by C++, Java, and Python.
- The cross-lingual acumen of *CodeBERT*, owing to its training on a diverse array of languages (*Java, Python, Go, Ruby, Javascript, C#*), is discernible in its superior performance across these languages, substantiating the model's robust cross-lingual code understanding.
- *Starencoder* exhibits diminished performance for language *D*, ostensibly attributable to the scarcity of resources in both the *XCODEEVAL* and *The Stack* datasets. The magnitude of corpus size is identified as a potential contributing factor, with larger corpora potentially leading to attenuated scores for both *CodeBERT* and *Starencoder*.
- In the context of *NL-Code*, *Starencoder* manifests commendable performance, albeit with a marginal diminution for language *D*, mirroring a parallel phenomenon observed in the *Code-Code* dataset.

These nuanced insights, gleaned from a meticulous examination of the comparative performance metrics, serve to inform a deeper understanding of the underlying intricacies and challenges inherent in the multi-lingual code retrieval paradigm.

Chapter 7

Code Search Engine

In order to facilitate user interactions, a sophisticated website has been carefully crafted employing cutting-edge technologies such as *ReactJS* and *Material-UI*. This dynamic combination not only ensures a seamless and responsive user experience but also reflects a commitment to employing industry-leading frameworks for frontend development.

The backend of the system, serving as the backbone for query processing and information retrieval, has been expertly developed utilizing the *Flask* library in the Python programming language. Flask, renowned for its simplicity and flexibility, lends itself seamlessly to the creation of robust web applications, allowing for the receipt of user queries and the subsequent provision of query results from the Dense Passage Retrieval (DPR) system.

Subsequent sections of this research delve into the intricacies of the user interface, dissecting the user-centric design principles implemented to enhance usability and overall user satisfaction. A thorough analysis of the system's performance metrics is also provided, shedding light on the efficiency and responsiveness of the implemented architecture.

Moreover, a comprehensive examination of the backend HTTP server is undertaken, exploring its role in handling and processing user queries, as well as its seamless integration with the DPR system. This multifaceted approach not only underscores the technological prowess employed in the development process but also serves to elucidate the nuanced components that collectively contribute to the operational excellence of the web application.

In essence, this research endeavors to provide a comprehensive and scholarly exploration of the technologies underpinning the development of the website, elucidating the intricacies of the user interface, evaluating system performance, and dissecting the functionality

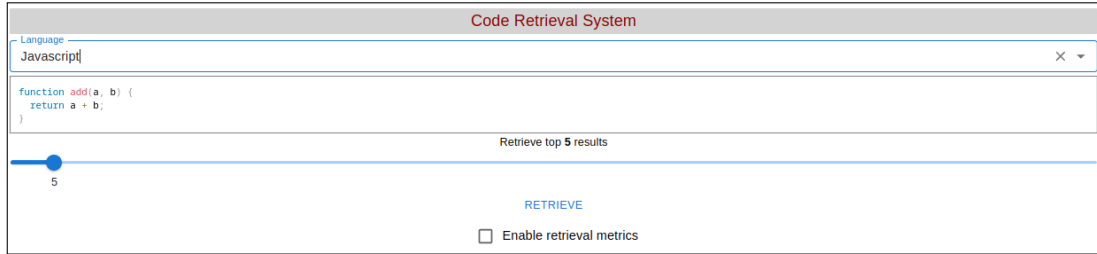


Figure 7.1: The query input section of UI.

of the backend HTTP server. Through this thorough examination, the reader gains a profound understanding of the intuitive design and seamless integration that characterize this state-of-the-art web application.

7.1 User Interface

The User Interface (UI) of the system is attentively crafted to ensure a seamless and intuitive user experience. It is composed of three fundamental components, each serving a distinct yet interrelated purpose: *Query Input*, *Retrieval Metrics*, and *Search Results Display*. The comprehensive examination of these components is imperative for a thorough understanding of the intricacies and functionalities of the UI.

7.1.1 Query Input

This part of the UI is for user input interactions. Figure 7.1 shows the initial condition of the UI and figure 7.2 shows the input field with a code. It consists of several components as listed below:

1. *Drop down menu for selecting language.* Here the users can select in which programming language are they going to type in the query code.
2. *Code editor.* This component is the main text input for the query. This has some advanced text editor features built into it, which are later discussed in details. In every 200ms, if there is a change in the code written in the editor, the new query results will be retrieved. This is a multiline text input field for users to input their code queries. Since users are expected to write codes here, the text input is enhanced with some code editor features such as:

```

Language
Rust

#![allow(unused_imports)]
use std::io::{self, prelude::*};
use std::str;
use std::mem::swap;
use std::cmp::*;
use std::collections::*;

fn solve<R: BufRead, W: Write>(scan: &mut Scanner<R>, w: &mut W){
    let mut t = scan.token::<i16>();
    // let mut t = 1;
    while { t -= 1; t + 1 } > 0 {
        let n = scan.token::<u64>();
        let s = scan.token::<String>();
        let mut mx = 0;

        for ch in s.chars() {
            mx = max(mx, ch as i32);
        }

        writeln!(w, "{}", mx - ('a' as i32) + 1);
    }
}

fn main(){
    let (stdin, stdout) = (io::stdin(), io::stdout());
    let mut scan = Scanner::new(stdin.lock());
    let mut out = io::BufWriter::new(stdout.lock());

    solve(&mut scan, &mut out);
}

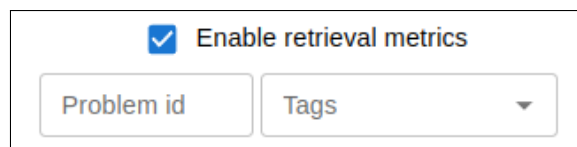
// Source: https://codeforces.com/profile/EbTech
// List of codes taken: imports, Scanner struct and its implementation,
// codes inside main function, and solve function's definition
struct Scanner<R> {
    reader: R,
    buf_str: Vec<u8>,
    buf_iter: str::SplitWhitespace<'static>,
}

impl<R: BufRead> Scanner<R> {
    fn new(reader: R) -> Self {
        Self { reader, buf_str: vec![], buf_iter: "".split_whitespace() }
    }
    fn token<T: str::FromStr>(&mut self) -> T {
        loop {
            if let Some(token) = self.buf_iter.next() {
                return token.parse().ok().expect("Failed parse");
            }
            self.buf_str.clear();
            self.reader.read_until(b'\n', &mut self.buf_str).expect("Failed read");
            self.buf_iter = unsafe {
                let slice = str::from_utf8_unchecked(&self.buf_str);
                std::mem::transmute(slice.split_whitespace())
            }
        }
    }
}
}

```

Figure 7.2: The query input filled up with an example Rust code.

- (a) Automatic indentation detection where upon entering a newline the cursor is placed at matching indentation level as the scope of code the cursor is in.
 - (b) Code highlighting is enabled when the language is selected. This marks different keywords and variables to better read the query code. Both figures 7.1 and 7.2 shows the syntax highlighting example.
3. *Number of search results to request.* This is a query parameter settings to represent the number of query result to retrieve from the database. Thus as the name suggests, it controls the query performance as retrieving more documents is computationally more costly. As seen in figures 7.1 and 7.2 show this value to be 5, and 100 respectively.
 4. *Search button.* A button which will request to backend for the query results. This is only a helper button for accessibility reasons as query results are requested on any change of the code editor text (throttled by 200ms for performance).
 5. *Button to enable query accuracy metrics.* This button allows the user to peak into the search accuracy of their result. As discussed in chapter 6, two documents are considered to be similar if they solve the same algorithmic problem. As for the code corpus of the DPR model, there is an unique id representing the problem it solves, Thus if a user knows the unique id of the problem that the query code belongs to, they can measure the performance of the results retrieved. Another way to measure performance would be to include the algorithmic techniques that the code in query uses, which is a known information for all the codes in the corpus and hence this also allows another measure of relevance of the retrieved results. When enabled it shows two more input fields for the unique problem id and the algorithmic tags as shown in figure 7.3. Also figure 7.4 shows the correct information to go along with the code shown in figure 7.2.



The image shows a rectangular box containing a checked checkbox with a blue checkmark and the text "Enable retrieval metrics". Below this checkbox are two input fields. The first is a text input field labeled "Problem id". The second is a dropdown menu labeled "Tags" with a small downward-pointing triangle on its right side.

Figure 7.3: Inputs to generate a metric report on the retrieved results.

RETRIEVE

Enable retrieval metrics

Problem id:

Tags:

Retrieved 100 search results in 167.600 μ s

8 results found from same problem as 1760-B out of 100 results.

Figure 7.4: Inputs with example data to generate a metric report on the retrieved results.

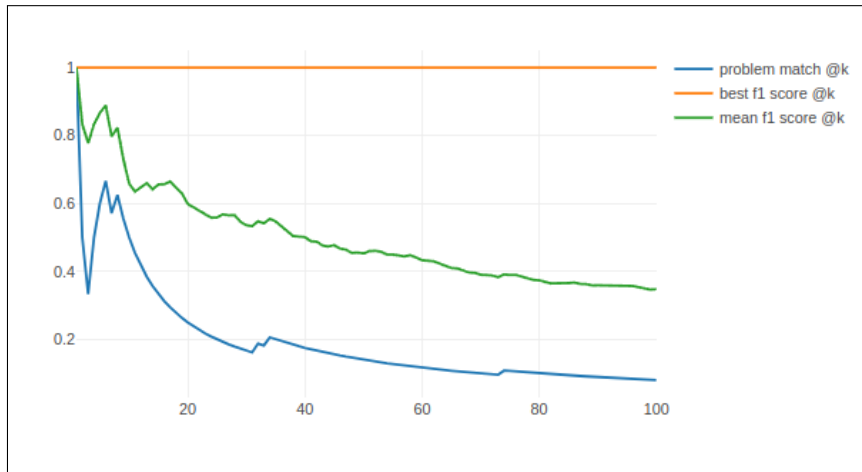


Figure 7.5: Retrieval metrics in UI.

Retrieved result #1

source: codeforces problem id: 1760-B id: 182046850 user: nikgaevoy language: Rust 2021 verdict: OK

Techniques used:

F1 score of tags: **1**

Figure 7.6: Retrieved result metadata display without enabling search metrics.

Retrieved result #4

source: codeforces **problem id: 1760-B** id: 181976898 user: bearking language: Rust 2021 **verdict: OK**

Techniques used: **greedy** **implementation** **strings**

F1 score of tags: 1

```

use std::io::{self, BufRead};
fn main() {
    let mut line = String::new();
    io::stdin().read_line(&mut line).expect("");
    let t: i32 = line.trim().parse().expect("");
    let reader = io::stdin();
    for _ in 0..t
    {
        reader.lock().lines().next();
        let res_string =
            reader.lock()
                .lines().next().unwrap().unwrap();
        let mut max_index=1;
        for c in res_string.chars()
        {
            max_index=std::cmp::max(max_index, (c as u32)-('a' as u32)+1);
        }
        println!("{}", max_index);
    }
}

```

Figure 7.8: Retrieved code displayed upon clicking on the result metadata display.

Retrieved result #1

source: codeforces **problem id: 1760-B** id: 182046850 user: nikgaevoy language: Rust 2021 **verdict: OK**

Techniques used: **greedy** **implementation** **strings**

F1 score of tags: 1

Retrieved result #2

source: codeforces **problem id: 1131-E** id: 50375937 user: kenkoooo language: Rust **verdict: WRONG_ANSWER**

Techniques used: **dp** **greedy** **strings**

F1 score of tags: 0.6666666666666666

Retrieved result #3

source: codeforces **problem id: 1131-E** id: 50377601 user: kenkoooo language: Rust **verdict: RUNTIME_ERROR**

Techniques used: **dp** **greedy** **strings**

F1 score of tags: 0.6666666666666666

Figure 7.7: Retrieved result metadata display with search metrics enabled.

7.1.2 Retrieval Metrics

This section of the UI is displayed when search metrics checkbox is selected and user has performed some query. This consists of a plot of three lines as follows:

1. Some basic counts are shown in figure 7.4 such as number of retrieved results and number of results that match the problem id.
2. *Problem match@k*. This line shows number of codes that are from the problem id as specified in the search metrics input in the top k results. Figure 7.5 shows the retrieval metrics, normalized problem id match cumulative sum, and cumulative max and mean of F1 score of input tags and tags of the retrieved results.
3. *best f1 score@k*. This line shows the maximum f1 score between tags selected by user and the tags of the codes in top k results. See figure 7.5 for an example.
4. *mean f1 score@k*. This line shows the average f1 score between tags selected by user and the tags of the codes in top k results. See figure 7.5 for an example.

7.1.3 Retrieved Results Display

This section of the UI is for displaying the retrieved results in order of relevance ranking when a query is performed. For each result a expandable block is displayed. Several metadata are displayed along with the retrieved code to help the user in inference. The displayed information for each result as shown in figures 7.6 and 7.7 is listed below:

1. *Rank of the result*. It is shown as ‘Search result #i’ for i^{th} result.
2. *Source*. This shows the source website for the code. For our case it is always ‘Codeforces’ as the current database consists of data from ‘Codeforces’ only.
3. *Problem id*. This value is the problem id that the retrieved code belongs to. If the search metrics are enabled then it is also colored according match or mismatch with the user inputted problem id.
4. *Id*. An unique id for the code.
5. *User*. The author of the code.
6. *Language*. The programming language in which the code is written in.
7. *Verdict*. The execution outcome of the code according to the source website. This value is ‘OK’ if the code actually solves the problem indicated by the problem id in the retrieved results, otherwise shows one of ‘WRONG_ANSWER’, ‘COMPI- LATION_ERROR’, ‘RUNTIME_ERROR’, ‘TIME_LIMIT_EXCEEDED’, ‘MEM- ORY_LIMIT_EXCEEDED’.

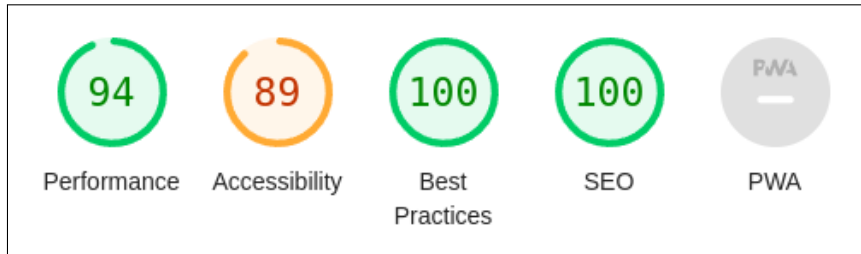


Figure 7.9: Lighthouse score of the UI,

8. *Tags*. List of algorithmic techniques that the retrieved code needs to use in order to solve the problem to which it belongs to. If the verdict is 'OK' then the code should be using a subset of the tags shown here, otherwise wrong codes have no guarantee of using the techniques mentioned in tags.
9. *F1 score*. The f1 score of tags of retrieved code and tags by user input in search metrics input.
10. *The retrieved code*. This component shows the retrieved code with syntax highlighting. This part is initially not displayed and is shown when the user clicks anywhere on the i^{th} result. Example of the retrieved code displayed can be seen from figure 7.8.

The UI has been audited with google's Lighthouse and the scores are as shown in figure 7.9.

7.2 Backend

The implemented HTTP server stands as a testament to the adept utilization of Python, specifically employing the Flask library for streamlined web service development. Positioned as the linchpin of a sophisticated information retrieval system, the server is carefully designed to interface with a dense vector database, colloquially referred to as the index, housing a comprehensive corpus.

At its core, the backend orchestrates a symphony of functionalities, with its primary mandate being the loading of the dense vector index associated with the textual corpus. Furthermore, it exhibits an exemplary responsiveness to incoming requests, a feat achieved by encoding the query code through a dedicated question encoder. This encoded query is then subjected to a rigorous search operation within the index, unveiling similar codes in consonance with the pre-defined notion of similarity, as expounded in (6.1.1). In this context, passages within the corpus are construed as codes, while the query code serves as the interrogative counterpart.

The runtime architecture of this server is bifurcated into two discrete stages, each wielding paramount significance in the overall system operation. The first phase, labeled **Initialization**, assumes a pivotal role in setting the foundation for subsequent operations. This phase encompasses the loading of the dense vector index, laying the groundwork for efficient and expeditious query handling in the subsequent phase.

The second phase, denoted as *Search*, constitutes the essence of the server's functionality. Here, the encoded query undergoes a complete search against the index, leveraging the principles delineated in (6.1.1). The search operation aims to discern codes within the corpus that resonate with the encoded query, thereby furnishing a set of results encapsulating semantically analogous elements.

In essence, the HTTP server, with its meticulous design and bifurcated runtime architecture, stands as a testament to the fusion of robust programming practices, leveraging the prowess of Python and the Flask library, and advanced information retrieval methodologies. The server seamlessly navigates the intricate interplay between vector representations and similarity metrics, epitomizing a sophisticated and efficient system for code retrieval within a comprehensive textual corpus.

7.2.1 Initialization

This step covers the initialization of index, and metadata database (a separate database to associate the indexed code with their other metadata that are not encoded in DPR). An in memory KV-pair database is used for better performance at the cost of higher memory. It is possible to opt-in for a slower databases based on disk storage. As the database is very large ($\sim 25\text{M}$ codes) both in memory and disk storage solutions have high tradeoff. Due to huge index size of $\sim 73\text{GB}$ it takes some time to boot.

7.2.2 Retrieval

After initialization, the HTTP server is started which listens for any retrieval request. An API endpoint is exposed that expects a JSON request body with three values (e.g. ‘code’, ‘n_docs’, and ‘tags’) where ‘code’ is the query code, ‘n_docs’ is the number of top results to retrieve, and ‘tags’ are the inputs from the metric as shown in figure 7.4.

7.2.3 Benchmark

Here we show a simple benchmark to measure the performance of the retrieval backend when a lot of request is made asynchronously by many agents. The retrieval is performed on a database of $\sim 25\text{M}$ codes loaded in memory. The table 7.1 shows time required in μs to process a request by the backend where N is the number of simulated agents performing the query request to the backend and R is the total number of requests performed by all N agents. The mean time to process a single query is $4508.9\mu\text{s}$ with std. deviation of $611.4\mu\text{s}$. This testify for the superior efficiency of our backend retrieval system.

Table 7.1: Request/second benchmark of retrieval backend.

$N \backslash R$	100	1000	10000	100000
10	4633.123	4187.428	4183.531	4213.226
50	5755.965	4087.244	4187.364	4256.504
100	5888.068	4178.531	4174.995	4293.091
150	5618.593	4127.591	4130.753	4226.417

Chapter 8

Conclusion

8.1 Executability

In our rigorous exploration documented in chapter 4, we embarked upon a pedantic journey to delineate the delicate concepts of executability and functional similarity within the realm of code analysis. The crux of our endeavor was to establish executability as a metric encapsulating the ability to ascertain the functional correctness of code, while concurrently highlighting the pivotal role of functional similarity as the linchpin for assessing the relevance between diverse code segments.

Our findings illuminate the paradigm shift introduced by this novel protocol, positioning it as the benchmark for evaluating LLMs in the context of code-related tasks. The execution-based evaluation protocol, as espoused in our research, posits executability as the foundational criterion for gauging the success of code. By elevating executability to a paramount status, we advocate for its intrinsic importance in the overall assessment of code quality and performance within the ambit of LLMs.

Moreover, our research underscores the criticality of functional similarity in various facets. Not only does it play a pivotal role in constructing retrieval datasets for XCODEEVAL, as expounded in chapter 5, but it also emerges as a cornerstone in the training and evaluation processes of our DPR models, as discussed in chapter 6.

In essence, our work contributes not only to the theoretical underpinnings of code analysis but also establishes a pragmatic and standardized approach to evaluating LLMs in the intricate domain of code-related tasks. The discerning recognition of executability and functional similarity as key pillars in this evaluation framework not only enhances our

understanding of code semantics but also lays the foundation for future advancements in the field.

8.2 XCODEEVAL and ExecEval

We are pleased to introduce XCODEEVAL in chapter 5, a pioneering large-scale multilingual multitask benchmark arduously designed for the fine-tune and evaluation of code-based large language models. Encompassing seven distinct tasks related to code understanding, generation, translation, and retrieval, XCODEEVAL operates across a rich spectrum of up to 17 programming languages. Central to its efficacy is the implementation of an advanced execution-based evaluation protocol, which enhances the depth and granularity of performance assessments.

Complementing XCODEEVAL, we present ExecEval in section 4.2, an innovative multilingual code execution engine engineered to seamlessly support all programming languages featured within the benchmark. This specialized execution engine, ExecEval, serves as a critical component within the framework, contributing to the holistic evaluation of large language models.

In essence, the synergistic integration of XCODEEVAL and ExecEval forms a distinctive framework that offers a paradigm shift in the examination and analysis of large language models. By facilitating comprehensive investigations, this framework not only promotes a deeper understanding but also enhances interpretability, thus opening avenues for profound exploration.

The utilization of extensive metadata and the adoption of an execution-based evaluation methodology are pivotal elements in this framework. Through these facets, we anticipate the unraveling of new scaling laws and the identification of emergent capabilities. Our aspiration is that researchers, leveraging the unique features embedded in XCODEEVAL and the capabilities of ExecEval, will embark on journeys of exploration, contributing to the collective understanding of large language models.

In conclusion, XCODEEVAL, in conjunction with ExecEval, embodies a sophisticated and forward-thinking approach, fostering an environment conducive to groundbreaking research. We remain optimistic that this framework will not only enrich the existing discourse on

large language models but also serve as a catalyst for future breakthroughs in the field.

8.3 DPR Model

In the subsequent section, as discussed in chapter 6, the intricate tasks of retrieval for both code-code and natural language (nl-code) domains were effectively addressed. The utilization of dense document representations for retrieval tasks was demonstrated as a cost-effective approach, particularly when the storage and indexing of the corpus are deemed manageable within the confines of the host system’s computational resources.

An insightful revelation emerged during the exploration of model performance, where a discernible enhancement in outcomes was observed with the adoption of higher batch sizes. It is imperative to underscore that augmenting batch sizes concurrently amplifies the memory requisites for training. A pivotal benchmark for this observation is the indispensable necessity of a 40GB GPU when training with a batch size of 128 for the *CodeBERT* architecture. This signifies that the pursuit of training with higher batch sizes necessitates the acquisition of modern GPUs endowed with expanded memory capacities.

Furthermore, the selection of a lower batch size for the *Starencoder* model can be attributed to the intrinsic intricacies associated with an increase in sequence length, which subsequently induces a polynomial escalation in memory requirements. In this specific instance, an 80GB GPU becomes a prerequisite for training purposes.

The comprehensive examination of both *code-code* models delineates their performance across a spectrum of mono-lingual and cross-lingual settings. The empirical results underscore the models’ remarkable proficiency in mono-lingual scenarios with an average over 90% accuracy, exhibiting a noticeable degradation in performance when subjected to cross-lingual evaluations, albeit maintaining a commendable score of above 55% throughout.

An accurate dissection of the *nl-code* retrieval task reveals an exemplary performance, attaining accuracy levels surpassing 80%. This achievement not only validates the efficacy of the model but also accentuates its potential for real-world applications in natural language understanding and code retrieval domains.

8.4 Code Search Engine

The preceding exposition, as delineated in Chapter 7, carefully elucidated a practicable instantiation of a sophisticated code search engine. This endeavor seamlessly integrated the substantive findings from both chapters 5 and 6. Noteworthy is the observation that the resultant search engine exhibited an exemplary level of responsiveness, demonstrating an impressive capability to retrieve a voluminous array of 100 documents from the corpus within the temporal confines of mere microseconds, irrespective of the intricacy of the queries posed.

Upon extrapolation to a real-world deployment scenario, it becomes evident that the preeminent impediment to optimal performance would be contingent upon the network latency inherent in the communication between the end-user and the server infrastructure. This underscores the paramount significance of network optimization strategies in mitigating latency concerns and ensuring the expeditious delivery of search results.

Furthermore, an intricately designed User Interface (UI) was crafted, attaining a commendable level of responsiveness. The UI not only facilitates seamless interaction but also offers a nuanced and insightful statistical analysis of the results retrieved. This strategic integration of statistical insights adds an invaluable layer of depth to the user experience, augmenting the utility of the search engine in facilitating informed decision-making and comprehensive data exploration. In essence, the culmination of these chapters manifests in a robust and efficient code search engine, poised to make substantial contributions in both research and practical application domains.

8.5 Future Works

It is imperative to underscore the versatile applicability of the DPR model, particularly in addressing the code-clone and plagiarism detection tasks. In this context, the model exhibits efficacy by encoding all suspected codes as both corpus and queries, subsequently assessing their relevance through the utilization of the FAISS framework. This innovative approach underscores the adaptability of the DPR model beyond conventional natural language understanding tasks, extending its utility to the intricate domain of code analysis.

Moreover, it is noteworthy that any RAG model can seamlessly integrate with the model trained within this research framework. The Bing chat platform stands out as a prime exemplar of the successful deployment of the RAG model, leveraging its capabilities for search result retrieval and subsequent generation of coherent text based on the retrieved information. This underscores the practical significance and real-world impact of the RAG model, as exemplified through the success of Bing chat.

To propel this project further, an elusive yet imperative task is to delve into the analysis and enhancement of cross-lingual retrieval performance. This facet necessitates an extensive exploration of language barriers and the development of strategies to mitigate them, thereby augmenting the model's efficacy in multilingual contexts. Furthermore, the imperative to cultivate generative models, not strictly bound by the constraints of retrieval augmentation, becomes apparent. These models can significantly contribute to addressing coding challenges presented in platforms such as XCODEEVAL, thereby expanding the scope and impact of the undertaken research initiative.

In conclusion, the potential ramifications of this research are expansive, encompassing advancements in code analysis, multi-lingual retrieval, and generative modeling for coding challenges. By elucidating the intricacies of model adaptability and real-world implementations, this research endeavors to contribute substantively to the burgeoning field of natural language processing and artificial intelligence.

Bibliography

- [1] R. Agashe, S. Iyer, and L. Zettlemoyer, “Juice: A large scale distantly supervised dataset for open domain context-based code generation,” *arXiv preprint arXiv:1910.02216*, 2019.
- [2] W. U. Ahmad, S. Chakraborty, B. Ray, and K. Chang, “Unified pre-training for program understanding and generation,” in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6-11, 2021*, K. Toutanova, A. Rumshisky, L. Zettlemoyer, D. Hakkani-Tür, I. Beltagy, S. Bethard, R. Cotterell, T. Chakraborty, and Y. Zhou, Eds. Association for Computational Linguistics, 2021, pp. 2655–2668. [Online]. Available: <https://doi.org/10.18653/v1/2021.naacl-main.211>
- [3] W. U. Ahmad, M. G. R. Tushar, S. Chakraborty, and K.-W. Chang, “Avatar: A parallel corpus for java-python program translation,” *arXiv preprint arXiv:2108.11590*, 2021.
- [4] B. Athiwaratkun, S. K. Gouda, Z. Wang, X. Li, Y. Tian, M. Tan, W. U. Ahmad, S. Wang, Q. Sun, M. Shang, S. K. Gonugondla, H. Ding, V. Kumar, N. Fulton, A. Farahani, S. Jain, R. Giaquinto, H. Qian, M. K. Ramanathan, R. Nallapati, B. Ray, P. Bhatia, S. Sengupta, D. Roth, and B. Xiang, “Multi-lingual evaluation of code generation models,” 2022. [Online]. Available: <https://arxiv.org/abs/2210.14868>
- [5] J. Austin, A. Odena, M. I. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. J. Cai, M. Terry, Q. V. Le, and C. Sutton, “Program synthesis with large language models,” *CoRR*, vol. abs/2108.07732, 2021. [Online]. Available: <https://arxiv.org/abs/2108.07732>
- [6] B. Berabi, J. He, V. Raychev, and M. T. Vechev, “Tfix: Learning to fix coding errors with a text-to-text transformer,” in *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, ser. Proceedings

- of Machine Learning Research, M. Meila and T. Zhang, Eds., vol. 139. PMLR, 2021, pp. 780–791. [Online]. Available: <http://proceedings.mlr.press/v139/berabi21a.html>
- [7] A. Borji, “A categorical archive of chatgpt failures,” 2023.
- [8] J. Cambronero, H. Li, S. Kim, K. Sen, and S. Chandra, “When deep learning met code search,” in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 964–974.
- [9] F. Cassano, J. Gouwar, D. Nguyen, S. Nguyen, L. Phipps-Costin, D. Pinckney, M.-H. Yee, Y. Zi, C. J. Anderson, M. Q. Feldman, A. Guha, M. Greenberg, and A. Jangda, “Multipl-e: A scalable and extensible approach to benchmarking neural code generation,” 2022.
- [10] S. Chandel, C. B. Clement, G. Serrato, and N. Sundaresan, “Training and evaluating a jupyter notebook data science assistant,” *arXiv preprint arXiv:2201.12901*, 2022.
- [11] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, “Evaluating large language models trained on code,” *CoRR*, vol. abs/2107.03374, 2021. [Online]. Available: <https://arxiv.org/abs/2107.03374>
- [12] X. Chen, K. Lakhota, B. Oguz, A. Gupta, P. S. H. Lewis, S. Peshterliev, Y. Mehdad, S. Gupta, and W. Yih, “Salient phrase aware dense retrieval: Can a dense retriever imitate a sparse one?” *CoRR*, vol. abs/2110.06918, 2021. [Online]. Available: <https://arxiv.org/abs/2110.06918>
- [13] Y. Cheng and L. Kuang, “Csrs: code search with relevance matching and semantic matching,” in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, 2022, pp. 533–542.
- [14] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, P. Schuh, K. Shi, S. Tsvyashchenko,

- J. Maynez, A. Rao, P. Barnes, Y. Tay, N. Shazeer, V. Prabhakaran, E. Reif, N. Du, B. Hutchinson, R. Pope, J. Bradbury, J. Austin, M. Isard, G. Gur-Ari, P. Yin, T. Duke, A. Levskaya, S. Ghemawat, S. Dev, H. Michalewski, X. Garcia, V. Misra, K. Robinson, L. Fedus, D. Zhou, D. Ippolito, D. Luan, H. Lim, B. Zoph, A. Spiridonov, R. Sepassi, D. Dohan, S. Agrawal, M. Omernick, A. M. Dai, T. S. Pillai, M. Pellat, A. Lewkowycz, E. Moreira, R. Child, O. Polozov, K. Lee, Z. Zhou, X. Wang, B. Saeta, M. Diaz, O. Firat, M. Catasta, J. Wei, K. Meier-Hellstern, D. Eck, J. Dean, S. Petrov, and N. Fiedel, “Palm: Scaling language modeling with pathways,” *CoRR*, vol. abs/2204.02311, 2022. [Online]. Available: <https://doi.org/10.48550/arXiv.2204.02311>
- [15] C. Cool and N. J. Belkin, *Interactive information retrieval: history and background*. Facet, 2011, p. 1–14.
- [16] J. Devlin, M. Chang, K. Lee, and K. 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, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, Eds. Association for Computational Linguistics, 2019, pp. 4171–4186. [Online]. Available: <https://doi.org/10.18653/v1/n19-1423>
- [17] L. Di Grazia and M. Pradel, “Code search: A survey of techniques for finding code,” *ACM Computing Surveys*, vol. 55, no. 11, pp. 1–31, 2023.
- [18] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, “Codebert: A pre-trained model for programming and natural languages,” in *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, ser. Findings of ACL, T. Cohn, Y. He, and Y. Liu, Eds., vol. EMNLP 2020. Association for Computational Linguistics, 2020, pp. 1536–1547. [Online]. Available: <https://doi.org/10.18653/v1/2020.findings-emnlp.139>
- [19] J. Finnie-Ansley, P. Denny, B. A. Becker, A. Luxton-Reilly, and J. Prather, “The robots are coming: Exploring the implications of openai codex on introductory programming,” in *Proceedings of the 24th Australasian Computing Education Conference*, ser. ACE ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 10–19. [Online]. Available: <https://doi.org/10.1145/3511861.3511863>

- [20] L. Fu, H. Chai, S. Luo, K. Du, W. Zhang, L. Fan, J. Lei, R. Rui, J. Lin, Y. Fang, Y. Liu, J. Wang, S. Qi, K. Zhang, W. Zhang, and Y. Yu, “Codeapex: A bilingual programming evaluation benchmark for large language models,” *CoRR*, vol. abs/2309.01940, 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2309.01940>
- [21] J. A. Goldstein, G. Sastry, M. Musser, R. DiResta, M. Gentzel, and K. Sedova, “Generative language models and automated influence operations: Emerging threats and potential mitigations,” 2023.
- [22] X. Gu, H. Zhang, and S. Kim, “Deep code search,” in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 933–944.
- [23] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, “Unixcoder: Unified cross-modal pre-training for code representation,” in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, S. Muresan, P. Nakov, and A. Villavicencio, Eds. Association for Computational Linguistics, 2022, pp. 7212–7225. [Online]. Available: <https://doi.org/10.18653/v1/2022.acl-long.499>
- [24] R. Guo, S. Kumar, K. Choromanski, and D. Simcha, “Quantization based fast inner product search,” in *Artificial intelligence and statistics*. PMLR, 2016, pp. 482–490.
- [25] R. Gupta, S. Pal, A. Kanade, and S. Shevade, “Deepfix: Fixing common c language errors by deep learning,” in *Proceedings of the aai conference on artificial intelligence*, vol. 31, no. 1, 2017.
- [26] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song, and J. Steinhardt, “Measuring coding challenge competence with APPS,” in *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, J. Vanschoren and S. Yeung, Eds., 2021. [Online]. Available: <https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c24cd76e1ce41366a4bbe8a49b02a028-Abstract-round2.html>
- [27] C. Hidey and K. McKeown, “Identifying causal relations using parallel Wikipedia articles,” in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 1424–1433. [Online]. Available: <https://aclanthology.org/P16-1135>

- [28] X. Hu, G. Li, X. Xia, D. Lo, S. Lu, and Z. Jin, “Summarizing source code with transferred api knowledge,” 2018.
- [29] J. Huang, C. Wang, J. Zhang, C. Yan, H. Cui, J. P. Inala, C. Clement, N. Duan, and J. Gao, “Execution-based evaluation for data science code generation models,” *arXiv preprint arXiv:2211.09374*, 2022.
- [30] H. Husain, H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, “Codesearchnet challenge: Evaluating the state of semantic code search,” *CoRR*, vol. abs/1909.09436, 2019. [Online]. Available: <http://arxiv.org/abs/1909.09436>
- [31] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, “Mapping language to code in programmatic context,” in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*, E. Riloff, D. Chiang, J. Hockenmaier, and J. Tsujii, Eds. Association for Computational Linguistics, 2018, pp. 1643–1652. [Online]. Available: <https://doi.org/10.18653/v1/d18-1192>
- [32] G. Izacard, M. Caron, L. Hosseini, S. Riedel, P. Bojanowski, A. Joulin, and E. Grave, “Unsupervised dense information retrieval with contrastive learning,” *Transactions on Machine Learning Research*, 2022. [Online]. Available: <https://openreview.net/forum?id=jKN1pXi7b0>
- [33] M. Izadi, R. Gismondi, and G. Gousios, “Codefill: Multi-token code completion by jointly learning from structure and naming sequences,” in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 401–412. [Online]. Available: <https://doi.org/10.1145/3510003.3510172>
- [34] J. Johnson, M. Douze, and H. Jégou, “Billion-scale similarity search with GPUs,” *IEEE Transactions on Big Data*, vol. 7, no. 3, pp. 535–547, 2019.
- [35] R. Just, D. Jalali, and M. D. Ernst, “Defects4j: A database of existing faults to enable controlled testing studies for java programs,” in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 437–440. [Online]. Available: <https://doi.org/10.1145/2610384.2628055>

- [36] V. Karpukhin, B. Oguz, S. Min, L. Wu, S. Edunov, D. Chen, and W. Yih, “Dense passage retrieval for open-domain question answering,” *CoRR*, vol. abs/2004.04906, 2020. [Online]. Available: <https://arxiv.org/abs/2004.04906>
- [37] O. Khattab and M. Zaharia, “Colbert: Efficient and effective passage search via contextualized late interaction over bert,” in *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 39–48. [Online]. Available: <https://doi.org/10.1145/3397271.3401075>
- [38] J. Kim, S. Lee, S.-w. Hwang, and S. Kim, “Towards an intelligent code search engine,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 24, no. 1, 2010, pp. 1358–1363.
- [39] K. Kim, D. Kim, T. F. Bissyandé, E. Choi, L. Li, J. Klein, and Y. L. Traon, “Facoy: a code-to-code search engine,” in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 946–957.
- [40] D. Kocetkov, R. Li, L. B. Allal, J. Li, C. Mou, C. M. Ferrandis, Y. Jernite, M. Mitchell, S. Hughes, T. Wolf, D. Bahdanau, L. von Werra, and H. de Vries, “The stack: 3 tb of permissively licensed source code,” 2022.
- [41] S. Kulal, P. Pasupat, K. Chandra, M. Lee, O. Padon, A. Aiken, and P. Liang, “Spoc: Search-based pseudocode to code,” *CoRR*, vol. abs/1906.04908, 2019. [Online]. Available: <http://arxiv.org/abs/1906.04908>
- [42] Y. Lai, C. Li, Y. Wang, T. Zhang, R. Zhong, L. Zettlemoyer, S. W.-t. Yih, D. Fried, S. Wang, and T. Yu, “Ds-1000: A natural and reliable benchmark for data science code generation,” *arXiv preprint arXiv:2211.11501*, 2022.
- [43] K. Lee, M.-W. Chang, and K. Toutanova, “Latent retrieval for weakly supervised open domain question answering,” in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Florence, Italy: Association for Computational Linguistics, Jul. 2019, pp. 6086–6096. [Online]. Available: <https://aclanthology.org/P19-1612>
- [44] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. tau Yih, T. Rocktäschel, S. Riedel, and D. Kiela, “Retrieval-augmented generation for knowledge-intensive nlp tasks,” 2021.

- [45] D. Li, Y. Shen, R. Jin, Y. Mao, K. Wang, and W. Chen, “Generation-augmented query expansion for code retrieval,” *arXiv preprint arXiv:2212.10692*, 2022.
- [46] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, Q. Liu, E. Zheltonozhskii, T. Y. Zhuo, T. Wang, O. Dehaene, M. Davaadorj, J. Lamy-Poirier, J. Monteiro, O. Shliazhko, N. Gontier, N. Meade, A. Zebaze, M.-H. Yee, L. K. Umapathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. Murthy, J. Stillerman, S. S. Patel, D. Abulkhanov, M. Zocca, M. Dey, Z. Zhang, N. Fahmy, U. Bhattacharyya, W. Yu, S. Singh, S. Luccioni, P. Villegas, M. Kunakov, F. Zhdanov, M. Romero, T. Lee, N. Timor, J. Ding, C. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu, J. Robinson, C. J. Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy, D. Fried, D. Bahdanau, Y. Jernite, C. M. Ferrandis, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, “Starcoder: may the source be with you!” 2023.
- [47] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago, T. Hubert, P. Choy, C. de Masson d’Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Goyal, A. Cherepanov, J. Molloy, D. J. Mankowitz, E. S. Robson, P. Kohli, N. de Freitas, K. Kavukcuoglu, and O. Vinyals, “Competition-level code generation with alphacode,” *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022. [Online]. Available: <https://www.science.org/doi/abs/10.1126/science.abq1158>
- [48] Libretexts, “6.3: Equivalence relations and partitions,” Jul 2020. [Online]. Available: https://math.libretexts.org/Courses/Monroe_Community_College/MTH_220_Discrete_Math/6%3A_Relations/6.3%3A_Equivalence_Relations_and_Partitions
- [49] S.-C. Lin and J. Lin, “A dense representation framework for lexical and semantic matching,” 2023.
- [50] C. Liu, X. Xia, D. Lo, C. Gao, X. Yang, and J. Grundy, “Opportunities and challenges in code search tools,” *ACM Computing Surveys (CSUR)*, vol. 54, no. 9, pp. 1–40, 2021.
- [51] S. Liu, Y. Chen, X. Xie, J. K. Siow, and Y. Liu, “Retrieval-augmented generation for code summarization via hybrid GNN,” in *International Conference on Learning Representations*, 2021. [Online]. Available: <https://openreview.net/forum?id=zv-typ1gPxA>

- [52] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, “Codexglue: A machine learning benchmark dataset for code understanding and generation,” in *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, J. Vanschoren and S. Yeung, Eds., 2021. [Online]. Available: <https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c16a5320fa475530d9583c34fd356ef5-Abstract-round1.html>
- [53] Y. Luan, J. Eisenstein, K. Toutanova, and M. Collins, “Sparse, Dense, and Attentional Representations for Text Retrieval,” *Transactions of the Association for Computational Linguistics*, vol. 9, pp. 329–345, 04 2021. [Online]. Available: https://doi.org/10.1162/tacl_a_00369
- [54] Z. Manna and R. J. Waldinger, “Toward automatic program synthesis,” *Commun. ACM*, vol. 14, no. 3, p. 151–165, mar 1971. [Online]. Available: <https://doi.org/10.1145/362566.362568>
- [55] A. V. Miceli Barone and R. Sennrich, “A parallel corpus of python functions and documentation strings for automated code documentation and code generation,” in *Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*. Taipei, Taiwan: Asian Federation of Natural Language Processing, Nov. 2017, pp. 314–319. [Online]. Available: <https://aclanthology.org/I17-2053>
- [56] D. Mount, “Lecture 17 network flow: Extensions,” 2017. [Online]. Available: <https://www.cs.umd.edu/class/fall2017/cmsc451-0101/Lects/lect17-flow-circ.pdf>
- [57] N. Muennighoff, Q. Liu, A. Zebaze, Q. Zheng, B. Hui, T. Y. Zhuo, S. Singh, X. Tang, L. von Werra, and S. Longpre, “Octopack: Instruction tuning code large language models,” *arXiv preprint arXiv:2308.07124*, 2023.
- [58] T. Nguyen, M. Rosenberg, X. Song, J. Gao, S. Tiwary, R. Majumder, and L. Deng, “MS MARCO: A human generated machine reading comprehension dataset,” *CoRR*, vol. abs/1611.09268, 2016. [Online]. Available: <http://arxiv.org/abs/1611.09268>
- [59] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, “Codegen: An open large language model for code with multi-turn program synthesis,” *arXiv preprint*, 2022.

- [60] R. Nogueira, Z. Jiang, R. Pradeep, and J. Lin, “Document ranking with a pretrained sequence-to-sequence model,” in *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online: Association for Computational Linguistics, Nov. 2020, pp. 708–718. [Online]. Available: <https://aclanthology.org/2020.findings-emnlp.63>
- [61] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura, “Learning to generate pseudo-code from source code using statistical machine translation,” in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 574–584.
- [62] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray *et al.*, “Training language models to follow instructions with human feedback,” *arXiv preprint arXiv:2203.02155*, 2022.
- [63] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, “Bleu: a method for automatic evaluation of machine translation,” in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, Jul. 2002, pp. 311–318. [Online]. Available: <https://aclanthology.org/P02-1040>
- [64] M. R. Parvez, S. Chakraborty, B. Ray, and K.-W. Chang, “Building language models for text with named entities,” *arXiv preprint arXiv:1805.04836*, 2018.
- [65] M. R. Parvez, W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, “Retrieval augmented code generation and summarization,” in *Findings of the Association for Computational Linguistics: EMNLP 2021*. Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 2719–2734. [Online]. Available: <https://aclanthology.org/2021.findings-emnlp.232>
- [66] R. Puri, D. S. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. Choudhury, L. Decker *et al.*, “Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks,” *arXiv preprint arXiv:2105.12655*, 2021.
- [67] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever *et al.*, “Improving language understanding by generative pre-training,” 2018.
- [68] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, “Exploring the limits of transfer learning with a unified text-to-text

- transformer,” *J. Mach. Learn. Res.*, vol. 21, pp. 140:1–140:67, 2020. [Online]. Available: <http://jmlr.org/papers/v21/20-074.html>
- [69] N. Reimers and I. Gurevych, “Sentence-bert: Sentence embeddings using siamese bert-networks,” *arXiv preprint arXiv:1908.10084*, 2019.
- [70] S. P. Reiss, “Semantics-based code search,” in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 243–253.
- [71] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, “Codebleu: a method for automatic evaluation of code synthesis,” *CoRR*, vol. abs/2009.10297, 2020. [Online]. Available: <https://arxiv.org/abs/2009.10297>
- [72] K. H. Rosen, *Discrete mathematics and its applications*. The McGraw Hill Companies, 2007.
- [73] B. Roziere, M.-A. Lachaux, L. Chausson, and G. Lample, “Unsupervised translation of programming languages,” *Advances in Neural Information Processing Systems*, vol. 33, 2020.
- [74] B. Roziere, J. M. Zhang, F. Charton, M. Harman, G. Synnaeve, and G. Lample, “Leveraging automated unit tests for unsupervised code translation,” *arXiv preprint arXiv:2110.06773*, 2021.
- [75] R. L. Russell, L. Y. Kim, L. H. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. M. Ellingwood, and M. W. McConley, “Automated vulnerability detection in source code using deep representation learning,” in *17th IEEE International Conference on Machine Learning and Applications, ICMLA 2018, Orlando, FL, USA, December 17-20, 2018*, M. A. Wani, M. M. Kantardzic, M. S. Mouchaweh, J. Gama, and E. Lughofer, Eds. IEEE, 2018, pp. 757–762. [Online]. Available: <https://doi.org/10.1109/ICMLA.2018.00120>
- [76] S. Sachdev, H. Li, S. Luan, S. Kim, K. Sen, and S. Chandra, “Retrieval on source code: a neural code search,” in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2018, pp. 31–41.
- [77] C. Sadowski, K. T. Stolee, and S. Elbaum, “How developers search for code: a case study,” in *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, 2015, pp. 191–201.

- [78] H. Sajnani, “Large-scale code clone detection,” *PhD Thesis, University of California, Irvine*, 2016.
- [79] M. Sanderson and W. B. Croft, “The history of information retrieval research,” *Proceedings of the IEEE*, vol. 100, no. Special Centennial Issue, pp. 1444–1451, 2012.
- [80] A. Shrivastava and P. Li, “Asymmetric lsh (alsh) for sublinear time maximum inner product search (mips),” *Advances in neural information processing systems*, vol. 27, 2014.
- [81] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, “Towards a big data curated benchmark of inter-project code clones,” in *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*. IEEE Computer Society, 2014, pp. 476–480. [Online]. Available: <https://doi.org/10.1109/ICSME.2014.77>
- [82] X. Tang, B. Qian, R. Gao, J. Chen, X. Chen, and M. Gerstein, “Biocoder: A benchmark for bioinformatics code generation with contextual pragmatic knowledge,” *CoRR*, vol. abs/2308.16458, 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2308.16458>
- [83] THUDM, “Codegeex: A multilingual code generation model,” <https://github.com/THUDM/CodeGeeX>, 2022.
- [84] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, “An empirical study on learning bug-fixing patches in the wild via neural machine translation,” *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 4, pp. 19:1–19:29, 2019. [Online]. Available: <https://doi.org/10.1145/3340544>
- [85] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” *Advances in neural information processing systems*, vol. 30, 2017.
- [86] Y. Wan, J. Shu, Y. Sui, G. Xu, Z. Zhao, J. Wu, and P. Yu, “Multi-modal attention network learning for semantic source code retrieval,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 13–25.
- [87] H. Wang, J. Li, H. Wu, E. Hovy, and Y. Sun, “Pre-trained language models and their applications,” *Engineering*, 2022. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S2095809922006324>

- [88] K. Wang, N. Thakur, N. Reimers, and I. Gurevych, “GPL: Generative pseudo labeling for unsupervised domain adaptation of dense retrieval,” in *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Seattle, United States: Association for Computational Linguistics, Jul. 2022, pp. 2345–2360. [Online]. Available: <https://aclanthology.org/2022.naacl-main.168>
- [89] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi, “Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation,” in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, M. Moens, X. Huang, L. Specia, and S. W. Yih, Eds. Association for Computational Linguistics, 2021, pp. 8696–8708. [Online]. Available: <https://doi.org/10.18653/v1/2021.emnlp-main.685>
- [90] Z. Wang, G. Cuenca, S. Zhou, F. F. Xu, and G. Neubig, “Mconala: a benchmark for code generation from multiple natural languages,” *arXiv preprint arXiv:2203.08388*, 2022.
- [91] Z. Wang, S. Zhou, D. Fried, and G. Neubig, “Execution-based evaluation for open-domain code generation,” *arXiv preprint arXiv:2212.10481*, 2022.
- [92] J. Wei, M. Bosma, V. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai, and Q. V. Le, “Finetuned language models are zero-shot learners,” in *International Conference on Learning Representations*, 2022. [Online]. Available: <https://openreview.net/forum?id=gEZrGCozdqR>
- [93] C. S. Xia and L. Zhang, “Less training, more repairing please: revisiting automated program repair via zero-shot learning,” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, A. Roychoudhury, C. Cadar, and M. Kim, Eds. ACM, 2022, pp. 959–971. [Online]. Available: <https://doi.org/10.1145/3540250.3549101>
- [94] P. Yin, B. Deng, E. Chen, B. Vasilescu, and G. Neubig, “Learning to mine aligned code and natural language pairs from stack overflow,” in *Proceedings of the 15th International Conference on Mining Software Repositories*, 2018, pp. 476–486.

- [95] P. Yin, W.-D. Li, K. Xiao, A. Rao, Y. Wen, K. Shi, J. Howland, P. Bailey, M. Catasta, H. Michalewski *et al.*, “Natural language to code generation in interactive data science notebooks,” *arXiv preprint arXiv:2212.09248*, 2022.
- [96] H. Yu, B. Shen, D. Ran, J. Zhang, Q. Zhang, Y. Ma, G. Liang, Y. Li, T. Xie, and Q. Wang, “Codereval: A benchmark of pragmatic code generation with generative pre-trained models,” *arXiv preprint arXiv:2302.00288*, 2023.
- [97] J. Zhang, S. Panthaplackel, P. Nie, J. J. Li, and M. Gligoric, “Coditt5: Pretraining for source code and natural language editing,” 2022.
- [98] V. Zhong, C. Xiong, and R. Socher, “Seq2sql: Generating structured queries from natural language using reinforcement learning,” *CoRR*, vol. abs/1709.00103, 2017.
- [99] Y. Zhou, S. Liu, J. K. Siow, X. Du, and Y. Liu, “Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks,” in *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. B. Fox, and R. Garnett, Eds., 2019, pp. 10 197–10 207. [Online]. Available: <https://proceedings.neurips.cc/paper/2019/hash/49265d2447bc3bbfe9e76306ce40a31f-Abstract.html>
- [100] M. Zhu, A. Jain, K. Suresh, R. Ravindran, S. Tipirneni, and C. K. Reddy, “Xlcost: A benchmark dataset for cross-lingual code intelligence,” 2022. [Online]. Available: <https://arxiv.org/abs/2206.08474>
- [101] Q. Zhu, Z. Sun, Y. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, “A syntax-guided edit decoder for neural program repair,” in *ESEC/FSE ’21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, Eds. ACM, 2021, pp. 341–353. [Online]. Available: <https://doi.org/10.1145/3468264.3468544>
- [102] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian, “Productivity assessment of neural code completion,” in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, ser. MAPS 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 21–29. [Online]. Available: <https://doi.org/10.1145/3520312.3534864>

Appendix A

Resources

Resources that are made available online in tandem with this project are listed below:

Table A.1: Links to resources made public.

xCODEEVAL	https://github.com/ntunlp/xCodeEval https://huggingface.co/datasets/NTU-NLP-sg/xCodeEval
ExecEval	https://github.com/ntunlp/ExecEval
Code Search Engine (frontend)	https://gitlab.com/Jackal_1586/code-search-engine
Code Search Engine (backend)	https://github.com/Jackal1586/dpr_xCodeEval