# ExeBench: An ML-Scale Dataset of Executable C Functions

Jordi Armengol-Estapé
jordi.armengol.estape@ed.ac.uk
University of Edinburgh
UK

Jackson Woodruff
J.C.Woodruff@sms.ed.ac.uk
University of Edinburgh
UK

Alexander Brauckmann
alexander.brauckmann@ed.ac.uk
University of Edinburgh
UK

José Wesley de Souza Magalhães
jwesley.magalhaes@ed.ac.uk
University of Edinburgh
UK

Michael F.P. O'Boyle
mob@inf.ed.ac.uk
University of Edinburgh
UK

## Abstract

Machine-learning promises to transform compilation and software engineering, yet is frequently limited by the scope of available datasets. In particular, there is a lack of runnable, real-world datasets required for a range of tasks ranging from neural program synthesis to machine learning-guided program optimization. We introduce a new dataset, ExeBench, which attempts to address this. It tackles two key issues with real-world code: references to external types and functions and scalable generation of IO examples. ExeBench is the first publicly available dataset that pairs real-world C code taken from GitHub with IO examples that allow these programs to be run. We develop a toolchain that scrapes GitHub, analyzes the code, and generates runnable-snippets of code. We analyze our benchmark suite using several metrics, and show it is representative of real-world code. ExeBench contains 4.5M compilable *and* 700k executable C functions. This scale of executable, real functions will enable the next generation of machine learning based programming tasks.

***CCS Concepts:*** • **Software and its engineering** → **Compilers**.

***Keywords:*** Code Dataset, Compilers, C, Mining Software Repositories, Machine Learning for Code, Program Synthesis

**ACM Reference Format:**
Jordi Armengol-Estapé, Jackson Woodruff, Alexander Brauckmann, José Wesley de Souza Magalhães, and Michael F.P. O'Boyle. 2022. ExeBench: An ML-Scale Dataset of Executable C Functions. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming (MAPS '22), June 13, 2022, San Diego, CA, USA*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3520312.3534867

ACM ISBN 978-1-4503-9273-0/22/06…$15.00
https://doi.org/10.1145/3520312.3534867

## 1 Introduction

Large datasets have been instrumental in accelerating innovation in machine learning deployment from image processing [1] to text-generation [2]. In the area of programming languages, we have witnessed a flurry of activity based around code datasets [3]. These cover many programming languages and are used for tasks such as code completion [4], clone detection [5, 6], code translation [7].

However, none are suitable for the tasks of neural compilation [8], decompilation [9] and program synthesis[10, 11]. Here, the key requirement is that the programs are executable and cover a diverse range of task representing the space of programs a compiler is likely to encounter [12] as anything else is likely to lead to models that do not generalize [4, 13, 14]. Existing datasets for these tasks are either synthetics [15] or contain too few tasks to cover the range of tasks such tools will encounter [6]. Synthetic datasets generalize more poorly than real code [16], and we currently lack a dataset of runnable real code required for many problems.

Anghabench [17] provides 1.04M compilable C functions scrapped from GitHub, exactly the kind of code that can be used to train models that generalize [18]. However, while these programs are useful for code compression and other static tasks, the existing datasets cannot be used for more challenging learning tasks that involve benchmark execution. Anghabench functions often missing externally defined types and function dependencies, are not executable and do not provide a set of inputs and outputs. ExeBench addresses this problem by automatically finding external tyes and functions and producing IO examples for 687,843 functions, enabling tasks ranging from neural synthesis to machine-learning based program optimization.

We scrape GitHub, the largest host of open-source code in the world [19], isolating deduplicated functions [20] as separate files that are compilable and executable from a standard harness. We search the wider code repository for the appropriate header and library include files which when compiled give a correctly functioning executable. We develop a scalable test generation tool that uses function signatures to automatically create multiple test input/output pairs without any external intervention.

In summary, we contribute with:

- ExeBench, the first machine leaning-scale set of executable programs in C, providing 700k functions.
- An analysis of the characteristics of ExeBench, showing that it represents a diverse set of code, representative of GitHub code.
- A methodology for building such datasets that can be applied to other repositories of C code.

We make the ExeBench dataset and tools available in a machine learning-friendly format.[1]

## 2 Related Work

There has been considerable effort in developing code datasets and generating appropriate test data.

### 2.1 Datasets

We can broadly spit existing datasets into those sourced from competitons and those based on more real-world sources.

***Competition Sourced.*** Due to the difficulties of generating appropriate input data, many executable code data sets are sourced from program competitions. Here the input and outputs are defined as part of the competition and by construction are guaranteed to be appropriate for the successful code entries. CodeNet [6] is a large dataset, but in the style of POJ-104 [5], focuses on a limited set of problems. ProgRes [21] uses similar data sources for a neural program synthesis dataset, attempting to overcome the limited number of problems by considering sub-regions of programs and using program slices to select datasets for code sub-regions.

***Real-world.*** A number of projects have set out to source large quantities of data, from real wold sources such as Github [22] and other repositories such as SourceForge and JavaForge [23]. They have created compilable datasets from Github in Java [24–27], Scala [28] and C [17], and synthetically in OpenCL [29], Python [30, 31] and C [32]. Gistable is an executable Python dataset [33] providing 10,259 executable snippets from Github. Docable [34] explore the executability of online tutorials.

Executable real-world C, however, is more challenging due to library dependence, global variables and pointers. CSNIPPEX [35] generates compilable snippets from Stackoverflow posts and various works rely on compilable Stackoverflow snippets for their analyses [36]. The closest related work is Anghabench [17], a collection containing over one million C benchmarks created out of open-source repositories. Each program consists of individual functions extracted from code mined from Github plus complementary code inserted by a type inference engine to solve external function dependencies that otherwise prevent compilation. Although compilable, Anghabench programs cannot be executed because they lack a main routine and input data. This prevents them from being used for tasks that involve execution and, as there are no outputs, behavioral equivalence.

### 2.2 Test Generation

Generating useful input data for C functions is critical for an executable code dataset. This is a well studied area with different SMT [37, 38] and symbolic execution-based approaches [39]. Other techniques use random test generation to achieve the same aims [40]. While in principle, test generation is a well studied, in fails in practise as it is either too slow (SMT) or requires external specification of input domains. Our approach to input generation is automatic and scalable, allowing the efficient generation of tests on a large scale, while still providing significant potential for detecting certain common errors [41, 42] and obtaining performance results for generated code.

## 3 Overview

Figure 1 shows an overview of our methodology. We first expand macros (1) except include directives and extract the real definitions (2a) by fixing the missing includes (if required). As an alternative, parallel approach, we inject synthetic dependencies for dependencies that cannot be found (2b). We also extract the definition of the function we are considering itself (2c), and metadata associated to this function (2d). After having extracted the real and synthetic auxiliary definitions, we run the IO generation tool for each approach (3a/3b).

This example illustrates the kind of IO we generate:

**Listing 1.** IO Example

```
double area (double a, double b)
{
  return ((f(a) + f(b)) / 2.0) * (b - a);
}

Input: (52.0270061796, 81.5855476831),
Output: 1283.869200682478
```

### 3.1 Terminology

We tackle two main challenges, *compilability* and *executability* of standalone C functions. For the latter, since functions may make use of externally defined auxiliary functions or types, we consider two distinct approaches: *synthetic type*

[1] https://github.com/jordiae/exebench

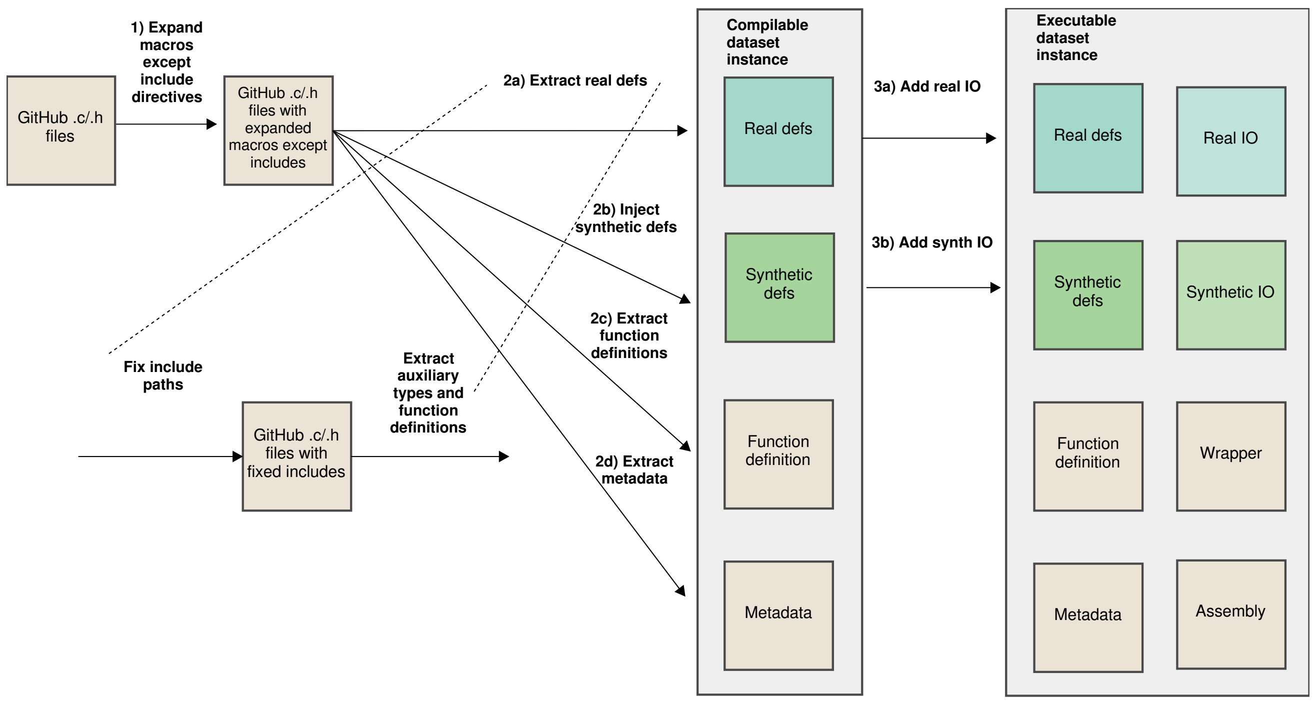

1) Expand macros except include directives
GitHub .c/.h files
GitHub .c/.h files with expanded macros except includes
2a) Extract real defs
2b) Inject synthetic defs
2c) Extract function definitions
2d) Extract metadata
Fix include paths
GitHub .c/.h files with fixed includes
Extract auxiliary types and function definitions
Compilable dataset instance
Real defs
Synthetic defs
Function definition
Metadata
3a) Add real IO
3b) Add synth IO
Executable dataset instance
Real defs
Real IO
Synthetic defs
Synthetic IO
Function definition
Wrapper
Metadata
Assembly

**Figure 1.** Overview of our method. We first expand macros except include directives. Then, we apply 4 independent, parallel steps: 2a) Extraction of real auxiliary function definitions, 2b) Injection of synthetic auxiliary function definitions, 2c) Extraction of function definition (the one of the function we are considering), 2d) Extraction of metadata. With that, we can run the IO generation tool with both the real auxiliary definitions (3a) and the synthetic ones (3b).

*and function declarations*, an existing scheme where type correct function and types declarations are inserted with no function bodies [17], and *real definitions* where we find the appropriate external header files and external C functions and types. Where real definitions are not found, we inject *synthetic type and function declarations* to make functions *compilable*, and *synthetic definitions* to make functions *executable*.

We distinguish between *synthetic IO* to refer to IO generated with synthetic auxiliary function definitions and types, and *real IO*, IO generated with the real auxiliary function and type definitions. We generate IO of varying complexity, distinguishing between *simple IO* which is IO for a constant or identity function, and *rich IO*, in which implementations need more complexity. Figure 2 shows a simplified Venn diagram of the subsets of functions considered in this work.

## 4 Data, Code Collection and Preprocessing

We use the large-scale GitHub Archive dataset as hosted in Google BigQuery[2] and download all `.c` and `.h` files. We use the Enry language identifier[3] to discard non-C header files and then extract function definitions from each file using Clang.

After extracting C function definitions, there are two key challenges - managing dependencies on external functions/-header files and generating test IO (section 5). For external dependencies management we experiment both with synthetic and real auxiliary types and functions definitions.

***Real Definitions.*** The ideal approach of building every repository as originally intended by the developers is not scaleable as projects have their own mechanism for specifying builds (e.g., repositories relying on following READMEs). Instead, we iterate over source files and apply heuristics to find the right include files enabling compilation of as many functions as possible. To find the appropriate include file, we build an SQL database with indices for filenames to allow rapid search and look up.

For each include directive, we check if it is present in the appropriate directory. For those that are missing: If it is a missing system include (`#include <lib.h>`), we remove it in the hope that functions present in that file not using that specific library will still be compilable. If it is a missing relative include (`#include "lib.h"`), we first check in the same repository if there exists a header file with the same name, and change the path accordingly. Otherwise, we check if

[2]https://cloud.google.com/blog/topics/public-datasets/github-on-bigquery-analyze-all-the-open-source-code
[3]https://github.com/src-d/enry

**Table 1.** Anghabench and GitHub statistics

| Dataset | Functions | Files | LOC per function | Repositories | Duplicated functions |
|---|---|---|---|---|---|
| Anghabench | 1,039,021 | 1,039,021 | 22.42 | 147 | 13.64% |
| GitHub | 8,362,855 | 9,036,556 | 18.13 | 386,288 | 48.73% |

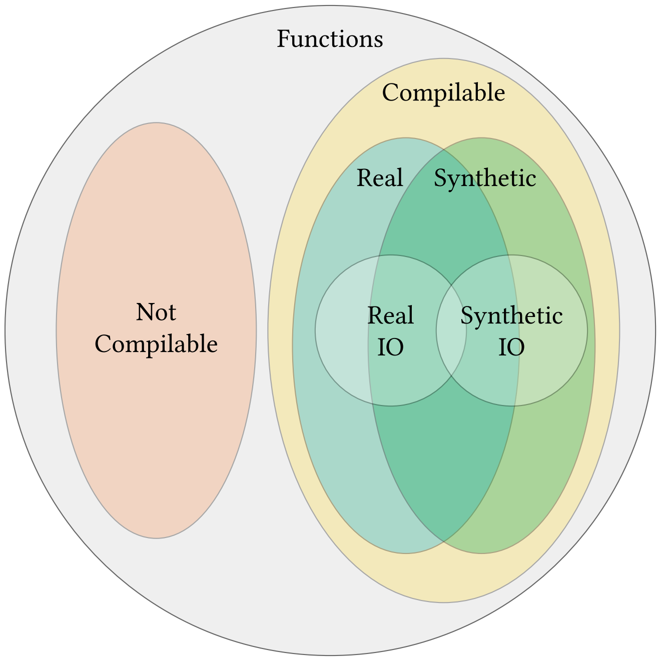


**Figure 2.** Venn diagram of the sets of functions we consider. We can make functions compilable either with synthetic or real declarations. We can make some of the compilable functions actually executable with IO by injecting synthetic definitions (aside from declarations) or making use of the real auxiliary function definitions.

such a file exists in the whole dataset using the SQL database and link to it. In the frequent case that there are multiples files with the same name, we prioritize those with the most stars, based on the assumption they are the most likely.

Table 1 shows statistics of the Anghabench and Github Archive datasets, before applying any preprocessing.

## 5 Execution and Unit Test Generation

Our goal is large scale test generation and we trade-off automation for context sensitivity e.g. some inputs may only be valid when positive. Capturing such context means we need either complex analysis or human assistance. Instead, we use function signatures to guide automatable and scalable test generation.

In the case of the synthetic approach, we generate 10 IO pairs per function, with 3 different seeds for the auxiliary definitions (30 IO in total). In the case of the approach with the real auxiliary function definitions, we generate 10 IO pairs per function.

### 5.1 Automatically Generating Specification Files from Function Headers

We use a JSON-based description of functions to generate inputs, based on FACC [43]. We developed a Python frontend that takes C functions as input and automatically generated the corresponding JSON files. The JSON IO format describes the variables that are live at entry and exit at of the function.

It includes several base types: integers `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`; floats `float16`, `float32`, `float64`; `unit`, and `string`. `array` and `struct` are parametric types. User types should be defined in the context of these base types. An example IO format description is:

**Listing 2.** IO Example

```
{   livein: ["x", "y", ...],
    liveout: ["x", "a"]
    typemap: {
        a: "int32",
        y: "float32"
        x: "MyComplexClass"
        ...
    }
}
```

Arrays need to have an associated length parameter, which is the size of the array to generate. While generating arrays of large size by default would avoid segmentation faults this would potentially add overhead and we settle in a default array size of 32. Strings use a standard C string format, with variable-length, null-terminated inputs.

The generated testing executable takes one argument, a JSON input file and produces as output a JSON output file. which specifies bindings between variables and their corresponding values.

**5.1.1 Global Variables and Arguments Passed as Reference.** C code relies heavily on global variables and arguments passed as reference (as opposed to by value), which makes testing more challenging. However, note that our specification does allow for global variables and arguments passed as reference to be both generated and evaluated. We consider all arguments passed as reference as outputs to be evaluated after the function execution. In addition, in case global variables used in the function are undefined (that is, in the case of synthetic dependencies), we consider them as part of the inputs to be randomly generated in each IO pair.

#### 5.1.2 Simple IO.

We define simple IO as the set of IO pairs that fulfills either one of the following conditions:

- *Constant*; each different input gives the same output. For example, if each of the $n$ IO pairs has the inputs I: $1, 2, \ldots, n$ and the outputs O: $0, 0, \ldots 0$ it is *constant*
- *Identity*:, the output variables have the same value as before executing the function.

While potentially useful to check correctness in some cases, simple IO is not useful for e.g. IO-based program synthesis, because the simplest implementation passing the tests can be a simple return value. However, simple IO has potential to be used to benchmark optimizations that already have correctness guarantees and do not need IO to verify behaviour, but to benchmark the execution itself. By design our tool can generate such simple IO instead of more complex IO depending on the functions, at the expense of being more scalable to real-world datasets than other possible approaches.

#### 5.1.3 Rich IO.

By rich IO we mean IO that does not fulfill the simple IO conditions, meaning that a constant or identity implementation cannot pass the tests. This IO has potential to be used in IO-guided program synthesis or code generation.

We report and study the typology of generated IO in Section 6.

#### 5.1.4 Generating Wrappers.

Our IO Generator produces JSON files with runnable values in them. To actually run a program, we must also generate a wrapper. The wrapper uses a C++ JSON libray to load values from JSON files into variables, then calls a function defining the user code.

We refer to the Supplementary Material for more details on the specification of our tool.

### 5.2 Safety of Executing Untrusted Code

In principle, memory leaks and segmentation faults of the executed code cannot affect the main program (the one getting or testing the IO pairs) because they are executed in separate processes with timeouts. However, in case of the code with real dependencies, executing untrusted code could indeed be unsafe for the host system. That is the reason why we execute all code in containers with limited permissions. Prospective users must be aware of this danger when executing the code with the real dependencies. In case of the Anghabench ones, safety is less of a concern because all system calls are replaced by dummy functions with the same signature.

## 6 Results

We present the results of applying our tool to two large-scale datasets, Anghabench and GitHub Public Archive, summarized in Table 2.

### 6.1 Anghabench

Anghabench contains 147 popular projects. Since it only provides synthetic declarations, we must also synthesize synthetic function definitions, and generate values for global variables to make them executable. Using the synthetic version of our approach, we generated 294k (28%) executable functions of which 37k contained rich IO. The approach with real auxiliary definitions cannot be applied to Anghabench due to the lack of the original context, but the functions considered in Anghabench are included in Github (Section 6.2).

### 6.2 GitHub

Over the total 8.5M extracted functions, around 77% were compilable with synthetic declarations. Of this 77%, we were able to make executable around 15% with synthetic auxiliary function definitions, of which around 16% were executable with rich IO. In the case of approach with the real auxiliary functions definitions, around 20% of the functions were compilable, with almost 5% of these functions being executable. Almost 10% of the executable functions with real definitions produced rich IO.

### 6.3 Representativeness

In this section, we study whether the obtained functions and IO are representative of the real-world distribution of C code.

***Distribution of Code Metrics.*** Figure 3 shows the distributions of various code complexity metrics on differnet samples of out extracted executable functions. We compare the behavior of our executable functions with that of the larger compilable, but not executable, Github population and show that are excutable functions match the larger population's characteristics.

The first code complexity metric we examine is the cyclomatic complexity, which measures number of linearly independent paths in a program. We also use the Halstead complexity metrics that are metrics to measure the effort of developing a program. Further, we examine the number of lines of code, the maximum Abstract Syntax Tree width, maximum Abstract Syntax Tree depth, and the number of basic blocks on unoptimized LLVM IR.

Figure 3 reveals that our functions are close to the samples found on GitHub (github-compilable), The distributions of our executable datasets align with this reference distribution, albeit the features are slightly smaller, indicating less complex samples. This is true for the Halstead complexities and the AST depth. The subsets with simple IO (not annotated with rich) match the reference distribution better for both the AnghaBench and GitHub subsets in terms of the complexity metrics, making them close to real-world code.

**Table 2.** Results in the original Anghabench dataset and in the Github crawling. Here we report results before deduplication to be closer to the real distribution of code and because real definitions may be different depending on the context (e.g., same function definition but different headers)

| Dataset | Method | Comp (% over total) | Exe (% over comp) | Rich IO (% over Exe) | Total |
|---|---|---|---|---|---|
| Angha | Synth | 1,039,021 (100%) | 294,396 (28.33%) | 36,665 (12.45%) | 1,039,021 |
| Angha | Real | 0 | 0 | 0 | |
| Github | Synth | 6,566,961 (77.49%) | 972,082 (14.80%) | 156,717 (16.12%) | 8,473,899 |
| Github | Real | 1,700,356 (20.07%) | 83,715 (4.92%) | 8,274 (9.88%) | |

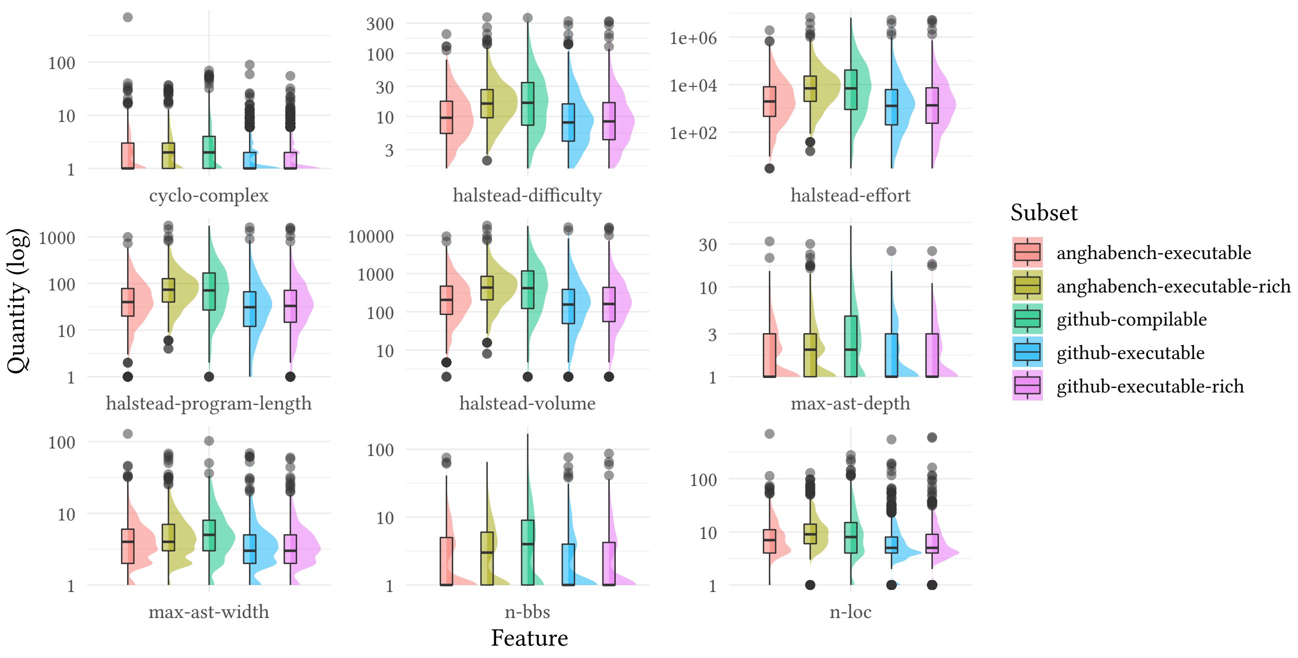


**Figure 3.** Distributions of feature quantities on different subsets of the dataset. AnghaBench and GitHub are the respective repositories. The GitHub reference distribution is denoted as github-compilable.

***Distribution of Code Embeddings.*** To compare the samples supported by our toolchain in more complex features, we use the code embeddings from InferCode [44]. InferCode is a pre-trained language model that predicts an embedding vector that captures semantic information of the input source code. We further use the UMAP algorithm for dimensionality reduction [45] to reduce the embeddings to a 2-dimensional space for better visualization.

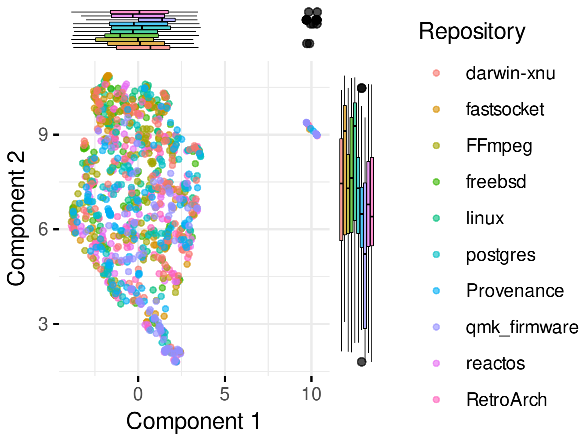
Figure 4a shows the reduced embeddings of the functions found in the 10 largest open-source projects of AnghaBench, colored by project. While we can see that many repositories share a similar feature space, several projects are dissimilar. This is most significant for the ffmpeg and qmk-firmware projects. Indeed, we can think of dissimilarities in the nature of the code of these two projects.

The distributions of compilable and executable functions in Figure 4b shows that they overlap, so the executable functions are a representative subset of the functions found in the 10 largest projects in AnghaBench. This supports that our methodology can indeed be used to make a representative subset of compilable functions executable.

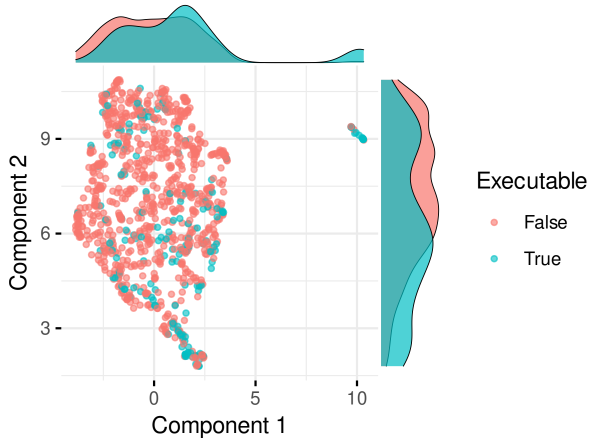
## 6.4 Error Analysis and Deduplication

***Error Analysis.*** The most common errors for generating the synthetic definitions are syntax errors due to unexpanded C macros, as reported in the original Anghabench paper. The most common compilation error with the real function definitions are missing header files that we were unable to find.

Regarding IO generation, our technique tends to perform poorer in programs that contain pointer arithmetic operations and conversions and/or a complex control flow structure. Figure 5 shows which program features lead to the lowest percentage values of successfully generated IO pairs. We can succeed only less than 5% of the programs that carry out pointer manipulation, as represented by the instructions



**(a)** Different repositories.



**(b)** Toolchain support.

**Figure 4.** Distribution of function embeddings wrt. different repositories and availability of IO examples.

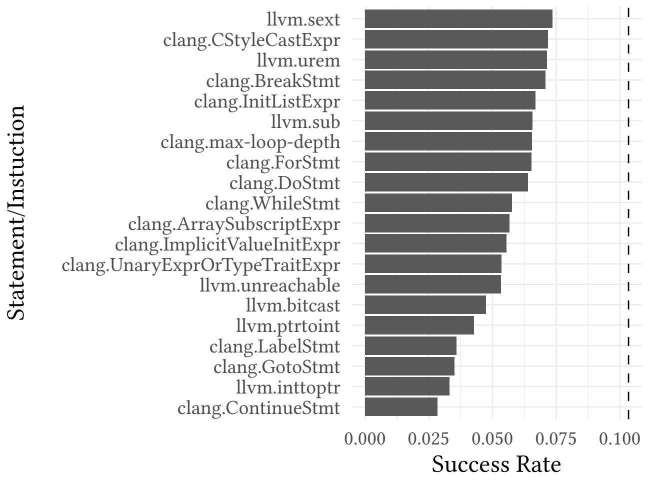


**Figure 5.** Statements or instructions that lead to lowest success rates when generating IO pairs. The success rate is the percentage of functions that are are executable and have at least one of the given statement or instruction over the functions that have the statement or instruction and are compilable. The dashed line represents the geometric mean accross the success rates of all features.

`inttoptr`, `ptrtoint`, and `bitcast`. The low percentage of correctly generated IO pairs in the case of programs that include `continue` and `goto` statements and `labels` seems to suggest that we are still likely to fail in cases of programs with more complex control flow structures.

***Duplication Analysis.*** Duplicated and near-duplicated instances are a common issue in machine learning datasets, affecting the reliability of the evaluation sets and the usefulness of training sets [20, 46, 47]. Here, we searched for exact duplicates of function definitions after tokenization, and found that close to 14% of the original Anghabench functions were duplicated, while almost half of the ones extracted from Github (Table 1). Note that this analysis considered all extracted functions, including the non-compilable ones.

## 7 ExeBench

The final dataset, which we call ExeBench, is constructed after deduplicating the outputs obtained in Section 6.

### 7.1 Deduplicating and Aggregating Code Datasets

We concatenate both Anghabench and Github dataset and apply exact deduplication based on function definition strings after tokenization. Before removal we check that external function references are identical as it is possible for otherwise identical functions to refer to external functions with the same names but different functionality. Table 3 shows the final statistics of ExeBench after deduplicating the outputs of Section 6.

### 7.2 Splits

We apply train-valid-test splitting, so that all researchers using this dataset can report comparable results and can be used as an actual benchmark instead of just a dataset. However, in this case we observe some challenges not present in more classical ML, related to the fact that not all functions have the same attributes (e.g., one function might have rich I/O or have no I/O at all). Thus we create different train, valid and test sets, ensuring that valid and test have all possible attributes to ease evaluation.

Table 4 shows the statistics of the different splits.

### 7.3 Row Fields

**func_def** C function definition.
**func_name** C function name.

**Table 3.** ExeBench stats for each subset after deduplicating and suggested uses

| Subset | Size | Suggested use |
|---|---:|---:|
| Not compilable | 735,700 | Unsupervised pretraining |
| Synth compilable | 4,189,813 | Compilation benchmarking |
| Real compilable | 893,561 | Compilation benchmarking |
| Compilable (synth ∪ real) | 4,485,817 | Compilation benchmarking |
| Synth simple IO | 644,758 | Optimization benchmarking |
| Real simple IO | 43,085 | Optimization benchmarking |
| Synth rich IO | 120,148 | IO-guided program synthesis, code generation, code search |
| Real rich IO | 4,464 | IO-guided program synthesis, code generation, code search |
| Total functions | 5,221,517 | |

**Table 4.** Exebench splits. Train, valid and test are disjoint. Train-not-compilable and the rest of train sets are disjoint. Train-synth-simple-IO and train-synth-rich-IO are also disjoint. However, Train-synth-compilable and Train-real-compilable intersect with all the train subsets with IO.

| Split | Synth comp | Real comp | Synth Exe | Real Exe | Synth Rich IO | Real Rich IO | Size |
|---|---|---|---|---|---|---|---:|
| train-not-compilable | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | 735,700 |
| train-synth-compilable | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | 4,179,345 |
| train-real-compilable | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | 884,543 |
| train-synth-simple-io | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | 644,758 |
| train-real-simple-io | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | 43,085 |
| train-synth-rich-io | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | 109,920 |
| valid-synth | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | 5,000 |
| valid-real | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ | 2,232 |
| test-synth | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | 5,000 |
| test-real | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ | 2,232 |

**func_sig** C function signature.
**path** Path in GitHub to the original file the function was extracted from.
**syn_defs** Synthetic dependencies (available).
**real_defs** Real dependencies (if available).
**syn_io** Synthetic IO (if available).
**real_io** Real IO (if available).
**wrapper** C++ wrapper to run the C function.
**asm** We provide diverse function assemblers (x86, ARM) with different optimizaton levels (O0, Ofast, Os).

### 7.4 Availability

We store the data in compressed JSON-L[4] files, the same format used in The Pile [48]. Every function and its attributes corresponds to one line in a JSON-L file. We plan to upload the dataset to the Huggingface Datasets[49, 50] Hub, allowing for easy, open access. With the dataset release we also plan to provide our infrastructure and utilities for actually running the functions and evaluating them in terms of the IO pairs.

### 7.5 Applications

Table 3 shows the suggested uses for each of subsets of ExeBench, including compilation and optimization benchmarking [51], and IO-guided program synthesis, code generation, and code search.

## 8 Conclusions

We present ExeBench, the first real-world ML-scale dataset of executable C code with IO examples. ExeBench contains 687,843 executable C functions and 4,485,817 compilable C functions, orders of magnitude more than previous efforts on open-source C repositories.

We discuss our novel methodology for generating executable code benchmarks from software repositories and demonstrate the the executable functions we extract are representative of real-world code.

ExeBench enables application of the latest generation of data-intensive machine learning models with use cases including neural compilation, decompilation, synthesis, and code evaluation.

[4]https://github.com/leogao2/lm_dataformat

## Broader Impact

We do not foresee any unfair usage of this work other than potential legal and ethical concerns on building models using other developers' code, which we believe to be alleviated by the fact that all the code we collected had already been processed into a public dataset before and that we include metadata to locate the original repository and check for license, if needed.

## Acknowledgements

Alexander Brauckmann and José Wesley de Souza Magalhães are partly sponsored by Huawei Research. We thank the reviewers for their insightful comments.

## References

[1] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009. doi: 10.1109/CVPR.2009.5206848.

[2] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020. doi: 10.48550/arXiv.2005.14165.

[3] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys*, 51:1–37, 7 2019. doi: 10.1145/3212695. URL http://dx.doi.org/10.1145/3212695.

[4] Vincent J Hellendoorn, Sebastian Proksch, Harald C Gall, and Ablerto Bacchelli. When code completion fails: A case study on real-world completions. *ICSE*, 2019. doi: 10.1109/ICSE.2019.00101.

[5] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. 2016. doi: 10.5555/3015812.3016002.

[6] Ruchir Puri, David S Kung, Wei Zhang, Giacomo Domeniconi, Vladmir Zolotov, Julian Dolby, Jie Chen, Mihir Choudbury, Lindsey Decker, Veronika Thost, Saurabh Buratti, Luca nad Pujar, and Ulrich Finkler. Project codenet: A large-scale AI for code dataset for learning a diversity of coding tasks. *Unpublished*, 2021. doi: 10.48550/arXiv.2105.12655.

[7] Ming Zhu, Karthik Suresh, and Chandan K Reddy. Multilingual code snippets training for program translation. *AAAI*, 2022.

[8] Jordi Armengol-Estapé and Michael O'Boyle. Learning c to x86 translation: An experiment in neural compilation. In *Advances in Programming Languages and Neurosymbolic Systems Workshop*, 2021. doi: 10.48550/arXiv.2108.07639. URL https://openreview.net/forum?id=444ug_EYXet.

[9] Omer Katz, Yuval Olshaker, Yoav Goldberg, and Eran Yahav. Towards neural decompilation. *CoRR*, abs/1905.08325, 2019. doi: 10.48550/arXiv.1905.08325. URL http://arxiv.org/abs/1905.08325.

[10] Maxwell Nye, Luke Hewitt, Joshua Tenenbaum, and Armando Solar-Lezama. Learning to infer program sketches. 2019. doi: 10.48550/arXiv.1902.06349.

[11] Bruce Collie and Michael FP O'Boyle. Program lifting using gray-box behavior. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 60–74. IEEE, 2021. doi: 10.1145/3425898.3426952.

[12] Richard Shin, Neel Kant, Kavi Gupta, CChristopher Bender, Brandon Trabucco, Rishabh Singh, and Dawn Song. Synthetic datasets for neural program synthesis. 2019.

[13] Bruce Collie, Jackson Woodruff, and Michael FP O'Boyle. Modeling black-box components with probabilitic synthesis. *GPCE*, 2020.

[14] Bruce Collie, Philip Ginsbach, Jackson Woodruff, Ajitha Rajan, and Michael FP O'Boyle. M3: Semantic api migrations. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 90–102. IEEE, 2020. doi: 10.48550/arXiv.2008.12118.

[15] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. 2017.

[16] Andres Goens, Alexander Brauckmann, Sebastian Ertel, Chris Cummins, Hugh Leather, and Jeronimo Castrillon. A case study on machine learning for synthesizing benchmarks. *MAPL*, 2019. doi: 10.1145/3315508.3329976.

[17] Anderson Faustino da Silva, Bruno Conde Kind, José Wesley de Souza Magalhães, Jerônimo Nunes Rocha, Breno Campos Ferreira Guimarães, and Fernando Magno Quinão Pereira. Anghabench: A suite with one million compilable c benchmarks for code-size reduction. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 378–390, 2021. doi: 10.1109/CGO51591.2021.9370322.

[18] Pang Wei Koh, Shiori Sagawa, Henrik Marklund, Sang Michael Xie, Marvin Zhang, Akshay Balsubramani, Weihua Hu, Michihiro Yasunaga, Richard Lanas Phillips, Irena Gao, Tony Lee, Etienne David, Ian Stavness, Wei Guo, Berton A Earnshaw, Imran S Haque, Sara Beery, Jure Leskovec, Anshul Kundaje, Emma Pierson, Sergey Levine, Chelsea Finn, and Percy Liang. Wilds: A benchmark of in-the-wild distribution shifts. *PMLR*, 2021. doi: 10.48550/arXiv.2012.07421.

[19] Georgios Gousios, Bogdan Vasilescu, Alexander Serebrenik, and Andy Zaidman. Lean GHTorrent: Github data on demand. *MSR*, 2014. doi: 10.1145/2597073.2597126.

[20] Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code. In *SPLASH '19: 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. ACM, 10 2019. ISBN ['9781450369954']. doi: 10.1145/3359591.3359735. URL http://dx.doi.org/10.1145/3359591.3359735.

[21] A large-scale benchmark for few-shot program induction and synthesis. *PMLR*, 2021.

[22] Vadim Markovtsev and Waren Long. Public Git archive: a big code dataset for all. 2018.

[23] Werner Janjic, Oliver Hummel, Marcus Schumacher, and Colin Atkinson. An unabridge source code dataset for research in software reuse. *MSR*, 2013. doi: 10.1109/MSR.2013.6624047.

[24] Pedro Martins, Rohan Achar, and Cristina V. Lopes. 50k-c: A dataset of compilable, and compiled, java projects. 2018. doi: 10.1145/3196398.3196450.

[25] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. The qualitas corpus: A curated collection of Java code for empirical studies. 2010. doi: 10.1109/APSEC.2010.46.

[26] Miltiadis Allamanis and Charles Sutton. Mining source code repositories at massive scale using language modeling. In *2013 10th IEEE Working Conference on Mining Software Repositories (MSR 2013)*. IEEE, 5 2013. ISBN ['9781467329361', '9781479903450']. doi: 10.1109/msr.2013.6624029. URL http://dx.doi.org/10.1109/msr.2013.6624029.

[27] Foyzul Hassan, Shaikh Mostafa, Edmund S L Lam, and Xiaoyin Wang. Automatic building of java projects in software repositories: A study on feasibility and challenges. *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2017. doi: 10.1109/ESEM.2017.11.

[28] Filip Kvrikava, Heather Miller, and Jan Vitek. Scala implicits are everywhere. *OOPSLA*, 2019. doi: 10.1145/3360589.

[29] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. Synthesizing benchmarks for predictive modeling. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*,

CGO '17, page 86–99. IEEE Press, 2017. ISBN 9781509049318. doi: 10.1109/CGO.2017.7863731.

[30] Xin Wang, Yasheng Wnag, Yao Wan, Fei Mi, Yitong Li, Pingui Zhou, Jin Liu, Hao Wu, Xin Jiang, and Qun Liu. Compilable neural code generation with compiler feedback. *CoRR*, 2022.

[31] Tamasz Korbak, Hady Elsahar, Marc Dymetman, and German Kruszewski. Energy-based models for code generation under compilability constraints. *CoRR*, 2021. doi: 10.48550/arXiv.2106.04985.

[32] Xinyun Chen, Dawn Song, and Yuandong Tian. Latent execution for neural program synthesis. *NeurIPS*, 2021. doi: 10.48550/arXiv.2107.00101.

[33] Eric Horton and Chris Parnin. Gistable: Evaluating the executability of python code snippets on github. *ICSME*, 2018.

[34] Samim Mirhosseini and Chris Parnin. Docable: Evaluating the executability of software tutorials. *FSE*, 2020. doi: 10.1145/3368089.3409706.

[35] Valerio Terragni, Yepng Lie, and Shing-Chi Cheung. CSNIPPEX: Automated synthesis of compilable code snippets from qa sites. *ISSTA*, 2016. doi: 10.1145/2931037.2931058.

[36] Di Yang, Aftab Hussain, and Cristina Videira Lopes. From query to usable ode: An analysis of stack overflow code snippets. *MSR*, 2016. doi: 10.1145/2901739.2901767.

[37] Rafael Dutra, Jonathan Bachrach, and Koushik Sen. SMTSampler: Efficient stimulus generation from complex SMT constraints. *ICCAD*, 2018. doi: 10.1145/3240765.3240848.

[38] Talia Ringer, Dan Grossman, Daniel Shwartz-Narbonne, and Serdar Tasiran. A solver-aided language for test input generation. *OOPSLA*, 2017.

[39] Willem Visser, Corina S Pasareanu, and Sarfraz Khushid. Test input generation with java pathfinder. *ISSTA*, 2004. doi: 10.1145/1007512.1007526.

[40] Thomas Lemberger. Plain random test generation with PRTest. *International Journal on Software Tools for Technology Transfer*, pages 871–873, 2021. doi: 10.1007/s10009-020-00568-x.

[41] Evan Maicus, Drumil Patel, Matthew Peveler, and Barbara Cutler. Random input and automated output generation in submitty. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, pages 1372–1372, 2020. doi: 10.1145/3328778.3372685.

[42] Sumukh Sridhara, Brian Hou, Jeffrey Lu, and John DeNero. Fuzz testing projects in massive courses. *L@S*, 2016. doi: 10.1145/2876034.2876050.

[43] Jackson Woodruff, Jordi Armengol-Estapé, Sam Ainsworth, and Michael F P O'Boyle. Bind the gap: Compiling real software to hardware FFT accelerators. *PLDI*, 2022.

[44] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. Infercode: Self-supervised learning of code representations by predicting subtrees. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1186–1197. IEEE, 2021. doi: 10.1109/ICSE43902.2021.00109.

[45] Leland McInnes, John Healy, and James Melville. Umap: Uniform manifold approximation and projection for dimension reduction, 2018. URL https://arxiv.org/abs/1802.03426.

[46] Björn Barz and Joachim Denzler. Do we train on test data? purging CIFAR of near-duplicates. *CoRR*, abs/1902.00423, 2019. doi: 10.3390/jimaging606004. URL http://arxiv.org/abs/1902.00423.

[47] Katherine Lee, Daphne Ippolito, Andrew Nystrom, Chiyuan Zhang, Douglas Eck, Chris Callison-Burch, and Nicholas Carlini. Deduplicating training data makes language models better. *CoRR*, abs/2107.06499, 2021. doi: 10.48550/arXiv.2107.06499. URL https://arxiv.org/abs/2107.06499.

[48] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. The pile: An 800gb dataset of diverse text for language modeling. *CoRR*, abs/2101.00027, 2021. doi: 10.48550/arXiv.2101.00027. URL https://arxiv.org/abs/2101.00027.

[49] Quentin Lhoest, Albert Villanova del Moral, Yacine Jernite, Abhishek Thakur, Patrick von Platen, Suraj Patil, Julien Chaumond, Mariama Drame, Julien Plu, Lewis Tunstall, Joe Davison, Mario Šaško, Gunjan Chhablani, Bhavitvya Malik, Simon Brandeis, Teven Le Scao, Victor Sanh, Canwen Xu, Nicolas Patry, Angelina McMillan-Major, Philipp Schmid, Sylvain Gugger, Clément Delangue, Théo Matussière, Lysandre Debut, Stas Bekman, Pierric Cistac, Thibault Goehringer, Victor Mustar, François Lagunas, Alexander Rush, and Thomas Wolf. Datasets: A community library for natural language processing. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 175–184, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.48550/arXiv.2109.02846. URL https://aclanthology.org/2021.emnlp-demo.21.

[50] Quentin Lhoest, Albert Villanova del Moral, Patrick von Platen, Thomas Wolf, Mario Šaško, Yacine Jernite, Abhishek Thakur, Lewis Tunstall, Suraj Patil, Mariama Drame, Julien Chaumond, Julien Plu, Joe Davison, Simon Brandeis, Victor Sanh, Teven Le Scao, Kevin Canwen Xu, Nicolas Patry, Steven Liu, Angelina McMillan-Major, Philipp Schmid, Sylvain Gugger, Nathan Raw, Sylvain Lesage, Anton Lozhkov, Matthew Carrigan, Théo Matussière, Leandro von Werra, Lysandre Debut, Stas Bekman, and Clément Delangue. huggingface/datasets: 1.15.1, November 2021. URL https://doi.org/10.5281/zenodo.5639822.

[51] Chris Cummins, Zacharias V Fisches, Tal Ben-Nun, Torsten Hoefler, Michael FP O'Boyle, and Hugh Leather. Programl: A graph-based program representation for data flow analysis and compiler optimizations. In *International Conference on Machine Learning*, pages 2244–2253. PMLR, 2021. doi: 10.48550/arXiv.2003.10536.