Project CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks

Project CodeNet Team
IBM Research and MIT-IBM Watson AI Lab

Abstract

Advancements in deep learning and machine learning algorithms have enabled breakthrough progress in computer vision, speech recognition, natural language processing and beyond. In addition, over the last several decades, software has been built into the fabric of every aspect of our society. Together, these two trends have generated new interest in the fast-emerging research area of "AI for Code". As software development becomes ubiquitous across all industries and code infrastructure of enterprise legacy applications ages, it is more critical than ever to increase software development productivity and modernize legacy applications. Over the last decade, datasets like ImageNet, with its large scale and diversity, have played a pivotal role in algorithmic advancements from computer vision to language and speech understanding. In this paper, we present *"Project CodeNet"*, a first-of-its-kind, very large scale, diverse, and high-quality dataset to accelerate the algorithmic advancements in AI for Code. It consists of 14M code samples and about 500M lines of code in 55 different programming languages. Project CodeNet is not only unique in its scale, but also in the diversity of coding tasks it can help benchmark: from code similarity and classification for advances in code recommendation algorithms, and code translation between a large variety programming languages, to advances in code performance (both runtime, and memory) improvement techniques. CodeNet also provides sample input and output test sets for over 7M code samples, which can be critical for determining code equivalence in different languages. As a usability feature, we provide several pre-processing tools in Project CodeNet to transform source codes into representations that can be readily used as inputs into machine learning models.

1 Introduction

There is a growing trend towards using AI for building tool support for code [1][2]. Artificial intelligence can manipulate and generate computer code, but can it do so with high quality? Many researchers are fascinated by this possibility, encouraged by AI successes in other domains and tantalized by the vision of computers programming computers. Given the success of non-AI tools for code, why should we consider AI to augment or possibly replace them? Some recent deep-learning models[3] for code have received a lot of publicity: trained on vast amounts of data and using novel architectures, they sometimes generate surprisingly convincing looking code. Also, where traditional coding tools contain heuristics, AI can help re-tune and adapt those more easily. Based on the training data from past experience, AI can help prioritize when there is more than one sound answer [4]. An AI-based tool may handle incomplete or invalid code more robustly, thus expanding its scope. In addition, AI can incorporate signals usually ignored by traditional tools for code, such as the natural language in identifiers or comments. In the enterprise environment, developers often face code written by large teams over many years. Developers must manipulate such code to modernize it,

Preprint. Under review.

fix bugs, improve its performance, evolve it when requirements change, make it more secure, and/or comply with regulations. These manipulations are challenging, and tool support can make developers more productive at performing them. It is well known that the latest advancements in deep learning algorithms rely on large-scale, diverse, and high quality datasets to demonstrate increasingly complex and powerful models. Over the last decade, datasets like ImageNet with its large-scale and diversity have played a pivotal role in algorithmic advancements from computer vision, to language and speech understanding. In this paper, we present"Project CodeNet", a first of its kind very large scale, diverse, and high-quality dataset to accelerate the algorithmic advances in AI for Code.

The rest of the paper is organized as follows. Section 2 introduces the Project CodeNet dataset. Related datasets are discussed in Section 3, and the differentiation of CodeNet with respect to these related datasets is elaborated in Section 4. Section 5 describes how CodeNet was curated and Section 6 enumerates the usability features of Project CodeNet with several pre-processing tools to transform source codes into representations that can be readily used as inputs into machine learning models. Section 7 describes important baseline experiments with the CodeNet dataset, and Section 8 concludes the paper with further uses of the CodeNet dataset and future enhancements.

2 The CodeNet Dataset

The CodeNet dataset consists of a large collection of code samples with extensive metadata. It also contains documented tools to transform code samples into intermediate representations and to access the dataset and make tailored selections. Our goal is to provide the community a large, high-quality curated dataset that can be used to advance AI techniques for source code.

Project CodeNet is derived from the data available on two online judge websites: AIZU [5] and AtCoder [6]. Online judge websites pose programming problems in the form of courses and contests. The dataset consists of submitted solutions, whose correctness is judged by an automated review process. Problem descriptions, submission outcomes, and associated metadata are available via various REST APIs.

Scale. Project CodeNet has approximately 14 million submissions to over 4000 problems. To our knowledge, this is the largest dataset so far of a similar kind. A large fraction (over 50%) of the code samples are known to compile and run correctly on the prescribed test cases. With the abundance of code samples, users can extract large benchmark datasets that are customized to their downstream use.

Diversity. The problems in Project CodeNet are mainly for instructional purposes and range from elementary to sophisticated exercises that require advanced algorithms to solve. The submitters range from beginners to experienced coders. Some submissions are correct while others contain different types of errors, accordingly labeled. The submissions are in many different languages.

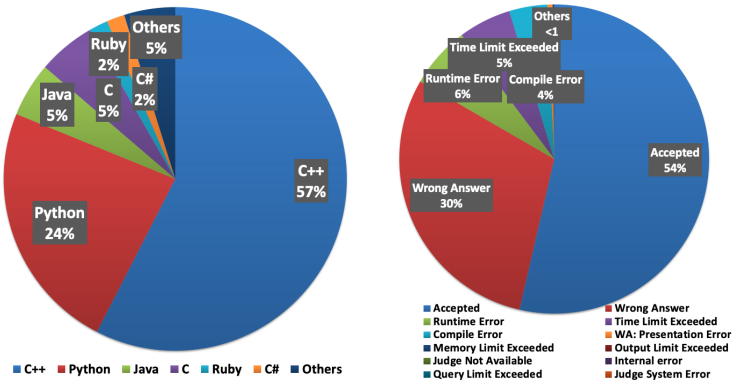
Code Samples. Each code sample is a single file and includes inputting the test cases and printing out the computed results. The file name uses standard extensions that denote the programming language; e.g., `.py` for Python. The majority of code samples contain only one function, although submissions to more complex problems might have several functions.

Metadata. The metadata enables data queries and selections among the large collection of problems, languages, and source files. The metadata is hierarchically organized in two levels. The first is the dataset level, which describes all problems. The second is the problem level, which details all source code submissions pertaining to a single problem. Metadata and data are separated in the dataset structure.

At the dataset level, a single CSV file lists all problems and their origins, along with the CPU time and memory limits set for them. Additionally, every problem has an HTML file with a detailed description of the problem, the requirements and constraints, and the IO examples.

At the problem level, every problem has a CSV file. The metadata for each submission is summarized in Table 6 in the supplement, which lists the fields contained in each CSV file as well as the corresponding descriptions.

Statistics. The dataset contains a total of 13,916,868 submissions, divided into 4053 problems, of which five have no submissions. Among the submissions, 53.6% (7,460,588) are accepted, 29.5% are marked with wrong answer, and the remaining rejected due to their failure to meet run time or



(a) Languages (b) Status

Figure 1: Percentage of submissions per language (left) and per status (right).

memory requirements. The dataset contains submissions in 55 different languages; 95% of them are coded in C++, Python, Java, C, Ruby, and C#. C++ is the most common language, with 8,008,527 submissions (57% of the total), of which 4,353,049 are accepted. See Figure 1 for a summary.

3 Related Datasets

A wide variety of datasets for source code exist, with many targeting one or a small number of tasks. Such tasks include clone detection, vulnerability detection [7, 8], cloze test [9], code completion [10, 11], code repair [12], code-to-code translation, natural language code search [13], text-to-code generation [14], and code summarization [13]. A detailed discussion of these tasks and their respective datasets is available in [15]. Project CodeNet, on the other hand, is not built on a specific task but aims at being comprehensive and supporting broad use. Two popular datasets of a similar kind are POJ-104 [16] and GCJ [17] (derived from Google Code Jam).

3.1 POJ-104

POJ-104 was collected from a pedagogical online judge system. The code samples are submissions to 104 programming problems. With 500 submissions to each problem, there is a total of 52,000 code samples in the dataset. This dataset has been used by quite a few authors for code classification, code similarity, and clone detection.

POJ-104 is faced with several limitations.

1. The number of problems is small. As a consequence, for problem classification, the accuracy percentage is generally in the mid to high nineties [18]. This dataset is considered too easy to drive further advance.
2. The code samples are in C and C++, but the two languages are not distinguished. Although they are closely related, mixing them leads to parsing errors and a reduction of useful code samples [18].
3. Important metadata are not present.
   (a) Code samples are not annotated with the results of the judging system. Information on whether the submission is compilable or accepted is missing. Therefore, for certain applications where code correctness is important, additional preprocessing efforts are needed and useful code samples are reduced [18].
   (b) The dataset does not contain the problem statement, although some example problems are described in [19].
   (c) Information on how to execute the code samples is absent.
4. Some problems are identical (e.g., problems 26 and 62).
5. Some submissions are near duplicates of each other, although the percentage of such cases is low compared to other datasets.

3.2 GCJ

GCJ was collected from the submissions to the Google Code Jam competition from 2008 to 2020. Similar to Project CodeNet, the submissions cover a wide variety of programming languages, with C++, Java, Python, and C being the predominant ones. The C++ subset has been extracted into a POJ-104-like benchmark and has been used in some publications. This benchmark has 297 problems and approximately 280K submissions. The number of submissions is imbalanced among problems.

GCJ is advantageous over POJ-104 in size and language diversity, but we believe that an even larger dataset such as Project CodeNet can better serve the community. GCJ does not contain any metadata in its current form, although the metadata can in principle be retrieved from the Google Code Jam website. Furthermore, GCJ potentially faces issues 4 and 5 above because no analysis on identical problems and near duplicates was performed.

4 CodeNet Differentiation

Table 1: Related datasets comparison

	CodeNet	GCJ	POJ
Total number of problems	4048	332	104
Number of programming languages	55	20	2
Total number of code samples	13,916,828	2,430,000	52,000
C++/C subset data size (code samples)	8,008,527	280,000	52,000
Percentage of problems with test data	51%	0%	0%
Task: Memory Consumption Prediction	Yes	No	No
Task: Runtime Performance Comparison	Yes	No	No
Task: Error Prediction	Yes	No	No
Task: Near duplicate prediction	Yes	No	No

A useful code dataset has certain desired properties. We constructed CodeNet according to these requirements. In the following, we discuss how CodeNet differentiates itself from the existing datasets along these lines. Table 1 is a comparison with related datasets.

Large scale. A useful dataset should contain a large number and variety of data samples to expose the realistic and complex landscape of data distributions one meets in practice. CodeNet is the largest dataset in its class - it has approximately 10 times more code samples than GCJ and its C++ subset is approximately 10 times larger than POJ-104.

Rich annotation. To enable the dataset for a wide range of applications and use cases, it is important to include information beyond which problem a code sample solves. For the dataset class in question, it is useful to know whether a code sample solves the problem correctly, and if not, the error category (e.g., compilation error, runtime error, and out-of-memory error). Since the source code is supposed to solve a programming problem, it is advantageous to know the problem statement and have a sample input for execution and a sample output for validation. All such extra information is part of CodeNet but absent in GCJ and POJ-104.

Clean samples. For effective machine learning, the data samples are expected to be independent and identically distributed (iid); otherwise, the resulting performance metric could be significantly inflated [20]. The existence of duplicate and/or near duplicate code samples makes the iid assumption dubious. Hence, it is crucial to identify the near duplicates. The presence of identical problems in the dataset poses even a bigger issue. In CodeNet, we analyzed the code samples for (near) duplication and used clustering to find identical problems. This information is made available as part of the dataset release but it is absent in GCJ and POJ-104.

5 Construction of CodeNet

5.1 Collection of Code Samples

The CodeNet dataset contains problems, submissions, and metadata, scraped from the AIZU and AtCoder online judging systems. For AIZU, we used the REST APIs provided to download all the metadata. For AtCoder, due to the absence of a REST API, we scraped the problems, submissions, and metadata directly from the web pages. We considered only public and non-empty submissions that did not contain errors or inconsistencies in the metadata. We manually merged the information from the two sources and adopted a unified format to create a single dataset

5.2 Cleansing

Because data were collected from different sources, we apply a consistent character encoding (UTF-8) on all raw data files. Additionally, we remove byte-order marks and use Unix-style line-feeds as line ending.

As indicated in section 4, we identify near duplicates. We follow Allamanis [20] and use Jaccard similarity [21] as a metric to score code pairs. Each code sample is tokenized and stored as a bag-of-tokens multiset. In our case, we keep all tokens except comments and preprocessor directives. We compute the set and multiset Jaccard indices and respectively use 0.9 and 0.8 as the near-duplicate thresholds.

The problems are gathered over many decades, so identical problems are likely. We go through the problem description files (in HTML format) and apply `fdupes` to extract identical problem pairs. Additionally, using the near-duplicate information calculated for code samples, we consider a problem pair to be a potential duplicate when the number of near-duplicate code pairs exceeds a threshold. clustering of duplicate problems is illustrated by the graphs in Figure 2, where each node denotes a problem and an edge between two nodes is labeled by the number of near-duplicate code pairs. Each connected graph is then a cluster of potential duplicate problems and we manually inspect the problem descriptions to verify the correctness of this duplicate detection.

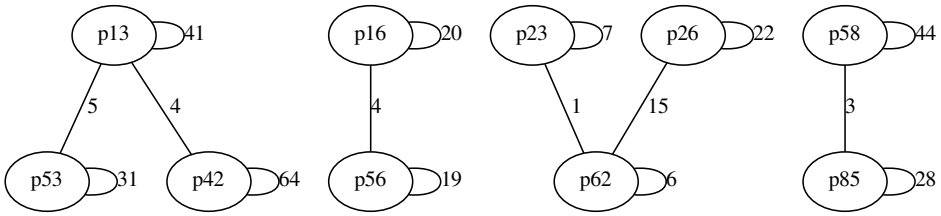


Figure 2: An example of near-duplicate problem graph.

5.3 Benchmarks

Project CodeNet has a rich set of code samples, and the user can assemble a customized benchmark according to his/her need. Following POJ-104, we extracted benchmark datasets from Project CodeNet in C++, Python, and Java. The benchmark characteristics are shown in Table 2. For the C++ benchmarks, the number of problems and their solutions are chosen to make the benchmark challenging. The benchmarks are filtered in the following ways:

1. Each code sample is "unique" in the sense that it is not a near duplicate of another code sample.
2. Each problem is "unique" in the sense that it is not a near duplicate of another problem.
3. Samples with a large fraction of dead code are excluded.
4. Each code sample has successfully gone through the tokenizer, the SPT generator, and the graph generator, all described in the next section. This step is to ensure that proper processing can be done to convert a code sample to a machine learning model input.

6 Code Representation and Tools

Machine learning with source codes requires proper abstractions of the codes. The abstractions are instantiated as representations in specific formats. As a usability feature, we provide several pre-processing tools to transform source codes into representations that can be readily used as inputs into machine learning models. They are described as follows.

Tokenizer. We offer fast C implementations of tokenizers for C, C++, Java, Python, and JavaScript. Additionally, the parse-tree generator described next can also produce token streams for C, C++, Java, and Python and can easily be extended for more languages.

Simplified Parse Tree (SPT) Simplified parse trees are derived from parse trees, which we generate using ANTLR4[22]. We traverse the ANTLR4 parse tree and remove internal nodes that only have one child while maintaining two key invariants: (1) the out-degree of any internal node is greater than 1; and (2) the out-degree of any internal node remains the same as in the parse tree. By doing so, we maintain the essential structure of the parse tree while pruning out unnecessary parser production rules. Finally, we adopt Aroma's [23] naming convention: leaf nodes are named by their literal strings and internal nodes are named by a concatenation of their children's names (only reserved words are kept while others are replaced by a hash mark #). We produce features for each node: (1) node type (token or parsing rule); (2) token type (e.g., an identifier), when applicable; (3) parsing rule type (e.g., an expression), when applicable; and (4) whether it is a reserved word. We adopt an extensible JSON graph schema so that edges can be augmented with types when needed. Currently, we support generating SPTs for four languages: C, C++, Java, and Python. Table 2 summarizes the SPT statistics for the four benchmarks.

Table 2: Benchmark statistics.

	C++1000	C++1400	Python800	Java250
#classes	1,000	1,400	800	250
#samples	500,000	420,000	240,000	75,000
#SPT-nodes	188,449,294	198,258,050	55,744,550	25,449,640
#SPT-edges	187,949,294	197,838,050	55,504,550	25,374,640

Code Graphs. We augment the tool chain with a code graph generator by using WALA[24], a general framework for program analysis. The backbone of a code graph is a System Dependence Graph, which is an interprocedural graph of program instructions (e.g. call, read), expressing control flow and data flow information as edges. We also generate inter-procedural control flow graphs (IPCFG), which are control flow graphs of all the methods in the program, stitched together to connect call sites with target methods. Our code graph tool currently supports only Java, but we plan to support more languages such as Python and Javascript.

7 Experiments with the CodeNet dataset

In this section, we report the results of code classification and code similarity experiments performed on benchmarks extracted from the CodeNet dataset. These experiments produce a set of baselines that other authors can use to gauge their models with. Our experiments cover code representations at the token level, parse tree level, and graph level. We also employ neural networks with different complexities, from convolution neural networks (CNNs) to graph neural networks (GNNs).

7.1 Code Classification

In code classification, each problem is one class. A code sample belongs to a class if it is a submission to the corresponding problem. Four benchmarks are extracted from Project CodeNet for experiments: Java, Python, C++ (1000 classes), and C++ (1400 classes). In each benchmark, 20% of the code samples are used for testing, while the rest are split in 4:1 for training and validation, respectively.

We experiment with a diverse set of machine learning methods, ranging from simple techniques (e.g., bag of tokens) to advanced models (e.g., GNNs). Details of each model, along with the experiment environment, are given in the appendix.

1. **MLP with bag of tokens.** A code sample is represented by a vector of relative frequencies of token occurrences. Only operator and keyword tokens are used. The model is a 3-layer MLP.
2. **CNN with token sequence.** We use the same set of tokens as above but retain their order to form a sequence. All sequences have the same length under zero padding. The classification model is a CNN with an initial token embedding layer.
3. **C-BERT with token sequence.** Treating a code sample as a piece of natural language text, we build a C-BERT model [25] through pretraining on 10K top starred Github projects written in C. We use the Clang C tokenizer and Sentencepiece to tokenize each code sample. The pretrained model is fine-tuned on each benchmark.
4. **GNN with SPT.** Based on the parse tree representation, we use graph convolutional networks (GCN) and graph isomorphism networks (GIN) as well as their variants as the prediction model [26]. The variant adds a virtual node to the graph to enhance graph message passing.

Table 3: Classification accuracy (in %).

	Java250	Python800	C++1000	C++1400
MLP w/ bag of tokens	71.87	67.84	68.23	64.73
CNN w/ token sequence	90.96	89.49	94.27	94.10
C-BERT	97.60	97.30	93.00	90.00
GNN (GCN)	88.92	91.64	91.49	90.18
GNN (GCN-V)	90.79	92.37	92.29	91.18
GNN (GIN)	89.12	91.35	93.04	92.05
GNN (GIN-V)	90.27	92.17	93.61	92.85

Table 3 summarizes the classification accuracy for all models on all benchmarks. Despite the simplicity of bag of tokens, its performance is already reasonably good. For example, on the Python benchmark with 800 classes, a random guess is only accurate 0.125% of the time, but this model achieves 67.84% accuracy. With ordering retained (CNN with token sequence), the performance is significantly improved, reaching approximately 90% across all benchmarks.

More complex neural models sometimes further improve the prediction performance, as witnessed by C-BERT, which reaches approximately 97% for both Java and Python. It is interesting to note that even though C-BERT is pre-trained with C programs, its performance on the two C++ benchmarks is less impressive. We speculate that such a lower performance is related to programming practices. For C++, it is common to have identical program construction, such as declaration of constants (e.g., pi and epsilon) and data structures, appear across C++ submissions to different problems, but such a practice is rare in Java and Python.

Overall, the GNN models exhibit good performance. They are consistently the top performers, if not the best. The -V variant consistently improves the performance by approximately 1%.

7.2 Code Similarity

In code similarity, two pieces of code samples are considered similar if they solve the same problem (so called type-4 similarity in [27]). Note that similarity in text does not necessarily mean they are similar in functionality. For example, programs that differ by only one token might behave differently; hence, they are not considered similar. We treat the problem as binary classification.

We use the same training, validation and testing split as in classification. Code pairs are randomly sampled within each subset. The number of similar pairs is the same as dissimilar ones. We experiment with the following models and methods.

1. **MLP with bag of tokens.** The model is the same as the one for code classification, except that the input is a concatenation of the two bag-of-tokens vectors from each program.
2. **Siamese network with token sequence.** The token sequence is the same as the one for code classification. The model is a Siamese network with two CNNs with shared weights.
3. **SPT based methods: AROMA and MISIM.** Based on the SPT code representation, a few off-the-shelf methods are available for predicting similarity. AROMA [23] normalizes SPT node names and uses them to extract rule-based features. Then, the similarity is computed as a dot

product. MISIM [18], on the other hand, converts the source code into a context aware semantic structure, uses a neural network (in our case, GNN) to extract high-level features, and uses the cosine similarity of the extracted features as the prediction.

Table 4: Similarity accuracy (in %).

	Java250	Python800	C++1000	C++1400
MLP w/ bag of tokens	82.48	86.64	85.69	86.47
Siamese w/ token sequence	89.39	94.86	96.70	96.36

Table 4 summarizes the classification accuracy for the first two models. The performance of bag of tokens is modest, considering that the problem is a binary classification with perfectly balanced classes. On the other hand, the Siamese model significantly outperforms bag of tokens, as expected.

Table 5: Similarity MAP@R score.

	C++1000	C++1400
AROMA	0.17	0.15
MISIM	0.75	0.75

Table 5 summarizes the MAP@R score for the two SPT-based approaches. The AROMA method has a relatively low performance metric because the feature extraction is rule based and no model is learned, whereas the MISIM method uses a neural network to extract features through supervised training.

7.3 A Masked Language Model for Token Inference

A task such as code completion relies on the ability to predict a token at a certain position in a sequence. To accomplish this we can build a Masked Language Model (MLM) using a technique that randomly masks out tokens in an input sequence and aims to correctly predict them in an as-yet-unseen test set. We train a popular BERT-like attention model on the C++ 1000 CodeNet benchmark after tokenization to a vocabulary of over 400 tokens and obtain a top-1 prediction accuracy of 0.9068 and a top-5 accuracy of 0.9935. More details can be found in Appendix D.

8 Further Uses of Project CodeNet

The rich metadata and language diversity open Project CodeNet to a plethora of uses cases. The problem-submission relationship in CodeNet corresponds to type-4 similarity [27] and can be used for code search and clone detection. The code samples in Project CodeNet are labeled with their acceptance status and we can explore AI techniques to distinguish correct codes from problematic ones. Project CodeNet's metadata also enables the tracking of how a submission evolves from problematic to accepted, which could be used for exploring automatic code correction. Each code sample is labeled with CPU run time and memory footprint, which can be used for regression studies and prediction.

Project CodeNet may also be used for program translation, given its wealthy collection of programs written in a multitude of languages. Translation between two programming languages is born out of a practical need to port legacy codebases to modern languages in order to increase accessibility and lower maintenance costs. With the help of neural networks, machine translation models developed for natural languages [28] were adapted to programming languages, producing pivotal success [3]. One considerable challenge of neural machine translation is that model training depends on large, parallel corpora that are expensive to curate [29], especially for low-resource languages (e.g., legacy code). Recently, monolingual approaches [30, 3] were developed to mitigate the reliance on parallel data, paving ways to build models for languages with little translation. Compared with current popular data sets (e.g., [3, 31]), Project CodeNet covers a much richer set of languages with ample training instances.

9 Conclusion

Artificial intelligence has made great strides in understanding human language. Computer scientists have been fascinated by the possibility and tantalized by the vision of computers (AI) programming computers. In this paper, we presented "Project CodeNet", a first-of-its-kind very large-scale, diverse and high-quality dataset to accelerate the algorithmic advances in AI for Code. This dataset is not only unique in its scale, but also in the diversity of coding tasks it can help benchmark: from code similarity and classification for advances in code recommendation algorithms, and code translation between a large variety programming languages, to advances in code performance improvement techniques.

10 References

[1] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37, 2018.

[2] Yanming Yang, Xin Xia, David Lo, and John Grundy. A survey on deep learning for software engineering. *arXiv preprint arXiv:2011.14597*, 2020.

[3] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. In *NeurIPS*, 2020.

[4] Zheng Wang and Michael O'Boyle. Machine learning in compiler optimization. *Proceedings of the IEEE*, 106(11):1879–1901, 2018.

[5] Aizu. `https://onlinejudge.u-aizu.ac.jp/introduction`.

[6] Atcoder. `https://atcoder.jp/`.

[7] Yunhui Zheng, Saurabh Pujar, Burn Lewis, Luca Buratti, Edward Epstein, Bo Yang, Jim Laredo, Alessandro Morari, and Zhong Su. D2a: A dataset built for ai-based vulnerability detection methods using differential analysis. In *Proceedings of the ACM/IEEE 43rd International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '21, New York, NY, USA, 2021. Association for Computing Machinery.

[8] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive programsemantics via graph neural networks. In *Advances in Neural Information Processing Systems*, pages 10197–10207. NeurIPS Foundation, 2019.

[9] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gonga, Linjun Shou, Bing Qin, Ting Liu, and Daxin Jiang. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155v4*, 2020.

[10] Miltiadis Allamanis and Charles Sutton. Mining source code repositories at massive scale using language modeling. In *10th Working Conference on Mining Software Repositories (MSR)*, page 207–216. IEEE, 2013.

[11] Veselin Raychev, Pavol Bielik, and Martin Vechev. Probabilistic model for code with decision trees. *ACM SIGPLAN Notices*, 2016.

[12] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. In *ACM Transactions on Software Engineering and Methodology (TOSEM)*, pages 1–29, 2019.

[13] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436v3*, 2019.

[14] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Mapping language to code in programmatic context. *arXiv preprint arXiv:1808.09588*, 2018.

[15] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation, 2021.

[16] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. Neural code comprehension: A learnable representation of code semantics. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 3588–3600. Curran Associates, Inc., 2018.

[17] Farhan Ullah, Hamad Naeem, Sohail Jabbar, Shehzad Khalid, Muhammad Ahsan Latif, Fadi Al-turjman, and Leonardo Mostarda. Cyber security threats detection in internet of things using deep learning approach. *IEEE Access*, 7:124379–124389, 2019.

[18] Fangke Ye, Shengtian Zhou, Anand Venkat, Ryan Marcus, Nesime Tatbul, Jesmin Jahan Tithi, Niranjan Hasabnis, Paul Petersen, Mattson. Timothy, Tim Kraska, Pradeep Dubey, Vivek Sarkar, and Justin Gottschlich. Misim: A novel code similarity system, 2021.

[19] `https://sites.google.com/site/treebasedcnn/home/problemdescription`.

[20] Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2019, page 143–153, New York, NY, USA, 2019. Association for Computing Machinery.

[21] Wikipedia. Jaccard index — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Jaccard_index`, 2020.

[22] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

[23] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. Aroma: code recommendation via structural code search. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–28, Oct 2019.

[24] IBM T.J. Watson Research Center. Wala. `https://github.com/wala/WALA`, 2021.

[25] Luca Buratti, Saurabh Pujar, Mihaela Bornea, Scott McCarley, Yunhui Zheng, Gaetano Rossiello, Alessandro Morari, Jim Laredo, Veronika Thost, Yufan Zhuang, and Giacomo Domeniconi. Exploring software naturalness through neural language models, 2020.

[26] Veronika Thost and Jie Chen. Directed acyclic graph neural networks. In *ICLR*, 2021.

[27] Hitesh Sajnani. *Large-Scale Code Clone Detection*. PhD thesis, University of California, Irvine, 2016.

[28] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. Preprint arXiv:1609.08144, 2016.

[29] Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree neural networks for program translation. In *NeurIPS*, 2018.

[30] Guillaume Lample, Alexis Conneau, Ludovic Denoyer, and Marc'Aurelio Ranzato. Unsupervised machine translation using monolingual corpora only. In *ICLR*, 2018.

[31] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. Preprint arXiv:2102.04664, 2021.

[32] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664, 2021.

[33] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2017.

[34] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *ICLR*, 2019.

[35] Lili Mou, Ge Li, Zhi Jin, Lu Zhang, and Tao Wang. TBCNN: A tree-based convolutional neural network for programming language processing. *CoRR*, abs/1409.5718, 2014.

[36] gcj-dataset. https://openreview.net/attachment?id=AZ4vmLoJft&name=supplementary_material.

[37] Kevin Musgrave, Serge J. Belongie, and Ser-Nam Lim. A metric learning reality check. *CoRR*, abs/2003.08505, 2020.

[38] Ankur Singh. "end-to-end masked language modeling with bert". https://keras.io/examples/nlp/masked_language_modeling.

A Further information of CodeNet

Table 6 summarizes the metadata available for each code submission to a problem. Figure 3 gives the distributions of problems based on number of submissions received.

Table 6: Submission metadata.

column	unit/example	description
submission_id	s[0-9]{9}	anonymized id of submission
problem_id	p[0-9]{5}	anonymized id of problem
user_id	u[0-9]{9}	anonymized user id
date	seconds	date and time of submission
language	C++	consolidated programming language
original_language	C++14	original language
filename_ext	.cpp	filename extension
status	Accepted	acceptance status, or error type
cpu_time	millisecond	execution time
memory	kilobytes	memory used
code_size	bytes	source file size
accuracy	4/4	passed tests (AIZU only)

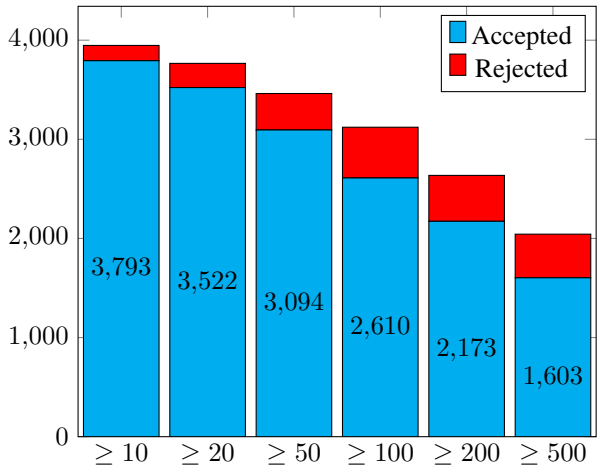


Figure 3: Number of problems providing at least X submissions. The bars show both the numbers of accepted submissions (blue) and rejected submissions (red).

B Details of Experiments on Code Classification

B.1 MLP with Bag of Tokens

One of the simplest representations of a code sample is a bag of tokens. Here, the code sample is represented by a vector of relative frequencies of token occurrences in the source code. The vector is computed by the following steps:

1. Convert a given source code into a sequence of tokens using a tokenizer (i.e., lexical analyzer).
2. From this sequence, remove the tokens considered not useful for code classification.
3. Count the number of each token type in the reduced sequence and form a vector of counts.
4. Normalize the vector with respect to L2 norm.

We do not use all tokens available in the grammar of the programming language. Only some operators and keywords are used. All identifiers, comments and literals are ignored. We also ignore some operators and many keywords that in our opinion provide no significant information on the algorithm the source code implements.

The vector representing a bag of tokens has the same length for every code sample, which makes it convenient for processing with a neural network. The vector is usually short, which makes training of a neural network fast. However, in a bag-of-tokens representation, information about the number of occurrences and position of each token is lost. Hence, the accuracy of a classifier using a bag-of-tokens representation is rather limited.

Table 7 provides results of code classification of all four benchmarks. The columns give the benchmark name, the test accuracy, the number of training epochs, the run time of each epoch, and the number of token types considered. All networks are implemented using Keras API. Training is performed on a single V100 GPU.

Table 7: Code classification by MLP with bag of tokens.

Benchmark dataset	Accuracy %%	Number epochs	Run time sec/epoch	Number tokens
Java	71.87	30	2	81
python	67.84	22	7	71
C++ 1000	68.23	20	14	56
C++ 1400	64,73	17	12	56

Figure 4 shows the neural network used for solving the classification problem for the C++ 1400 benchmark. The neural network has only three dense layers, two of which have ReLU activation. The last dense layer does not have any activation since its output goes directly to a softmax layer.

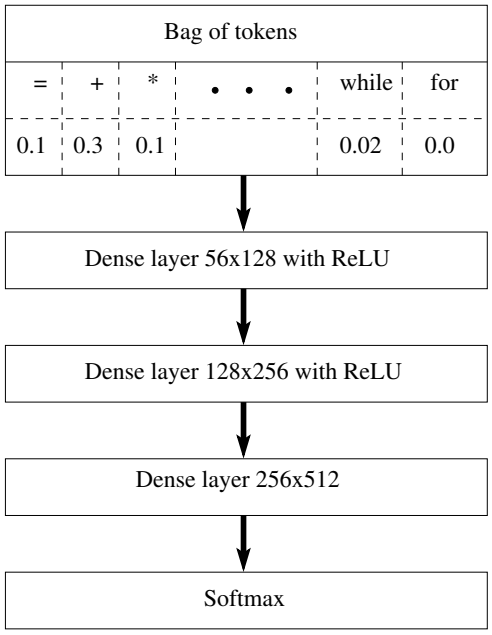


Figure 4: MLP architecture for code classification.

From Table 7 we see that training is rather fast, the reason being that the network is simple. In spite of simplicity, this neural network performs very well. The 64.73% test accuracy is significantly better than the potential 0.071% accuracy of random guess. It indicates that the relative frequencies of source code tokens provides sufficient information for classifying code.

The neural networks used for classification of other benchmarks are similar to the one shown in Figure 4. As we see in Table 7 their performance is quite similar.

B.2 CNN with Token Sequence

The sequence-of-tokens representation retains more information of a code sample than the bag-of-tokens representation. For our experiments on code classification, we use the same set of tokens that is used in the above bag-of-tokens approach. Similarly, we omit all comments and identifiers.

Table 8: Code classification by CNN with token sequence.

Benchmark dataset	Accuracy %%	Number epochs	Run time sec/epoch	Number tokens
Java	90.96	810	10	81
python	89.49	504	26	71
C++1000	94.27	235	59	56
C++1400	94.10	334	60	56

Table 8 shows results of code classification on all four benchmarks by using the sequence-of-tokens representation. The columns give the benchmark name, the test accuracy, the number of training epochs, the run time of each epoch, and the number of token types considered. All networks are implemented using Keras API. The training is performed on four V100 GPUs.

We have experimented with several types of neural networks. Figure 5 shows the neural network we choose for the C++ 1400 benchmark. It is a multi-layer convolutional neural network. It uses categorical encoding of source code tokens. For batching, the sequences of tokens are padded with zeros.

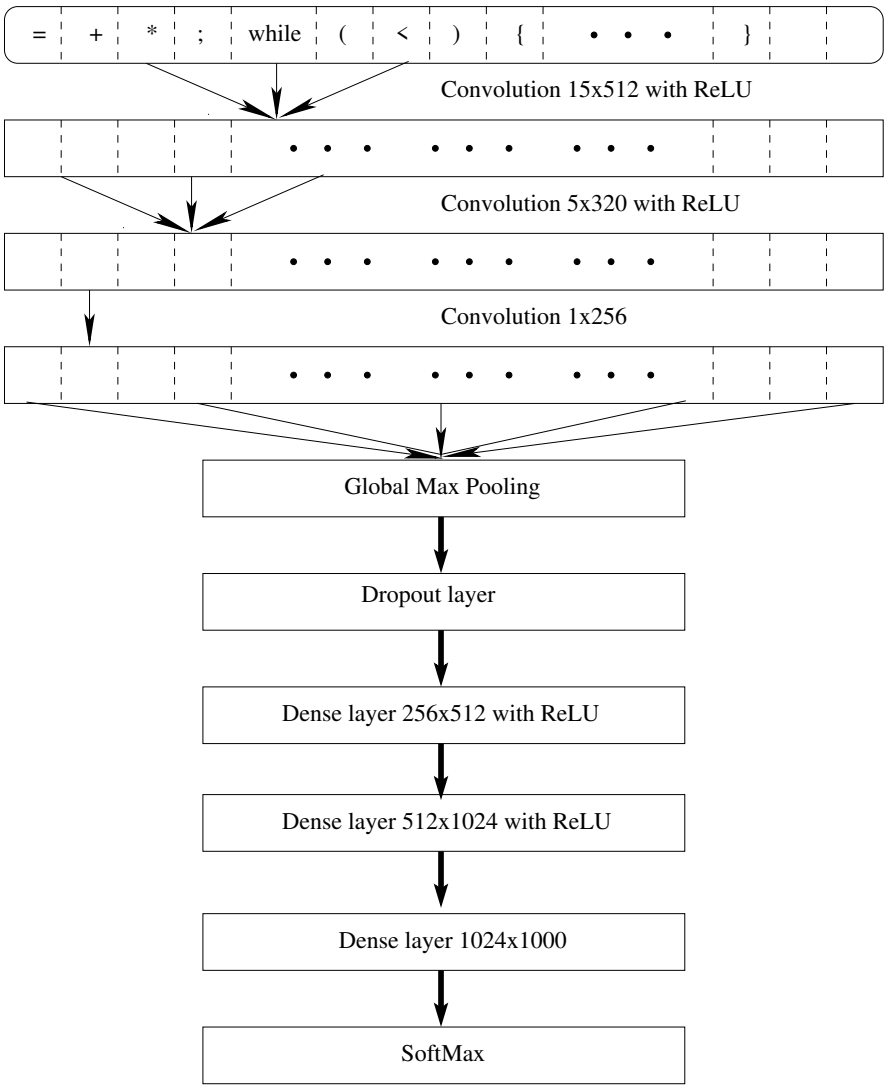


Figure 5: CNN architecture for code classification.

The first layer of the network is a trainable embedding layer, which converts tokens into embedding vectors. The embedding vectors are initialized randomly and learned.

The next three layers of the network are convolutions. The role of these layers is to extract the most important features of the source code. Two of the convolution layers have ReLU activation. The last convolution layer has no activation since its output goes through a max-pooling layer. The max-pooling layer selects dominant occurrences of the features extracted by the convolution layers. The output of the max-pooling layer is supplied to a stack of two dense layers. One of them has a ReLU activation. The last dense layer has no activation since its output goes through a softmax layer to perform actual classification.

Using this network we get a test accuracy 94.1%, which is significantly better than the accuracy shown by the bag-of-tokens approach.

The neural networks used for classification of other benchmarks are similar to the one shown in Figure 5. As we see in Table 8, their performance is similar.

B.3 C-BERT with Token Sequence

The sequence-of-tokens representation can be used with other neural networks of increasing capacity. We build a C-BERT model (a transformer model introduced in [25]) by pre-training on 10,000 top starred GitHub open source projects written in C, where we use Clang C tokenizer and Sentencepiece to tokenize the pre-training data. The C-BERT model is then fine tuned on each classification benchmark. Additionally, we experiment with the POJ-104 dataset, which contains code examples in C and C++.

C-BERT achieves appealing results on binary classification and vulnerability detection with C source code [7, 32]. However, it has not been used on multiclass classification tasks or with other languages such as C++, Java, and Python. Because we use sub-word tokenization and different programming languages share common tokens, we could apply the C-BERT model directly on the benchmarks.

After pretraining, we fine tune the model for five epochs on each benchmark, with a batch size 32 and learning rate 2e-5. The fine-tuning was done on two V100 GPUs and it took 30 minutes to four hours, depending on the size of the dataset. The sub-word vocabulary size is 5,000. Contexts larger than 512 tokens were truncated.

Table 9 summarizes the accuracies C-BERT achives on the four CodeNet benchmarks as well as the POJ-104 dataset. C-BERT achieves high accuracy and performs the best on Java and Python.

Table 9: C-BERT results for code classification.

	POJ-104	C++1000	C++1400	Java250	Python800
C-BERT	98.4%	93.0%	90.0%	97.6%	97.3%

The relatively low performance on C++ benchmarks is possibly related to the idiosyncrasies of the dataset and certain programming practices. Manual inspection suggests that lack of detailed variable names in C++ hurts the performance of the model, in problems appearing similar and having similar solutions. Removing one of the similar problems improves the model performance on the other problem. Moreover, one programming practice which could potentially confuse the models is that certain C++ users copied common constants (e.g., pi and epsilon) and data structures (e.g., enums) to all solutions they submitted. In many cases, these duplicate contents were not even used. We did not observe such practices in Python and Java.

B.4 GNN with SPT

We experiment with four GNNs: the graph convolutional network (GCN) [33], the graph isomorphism network (GIN) [34], and a variant for each (denoted by -V). The variant adds a virtual node to the graph to enhance graph message passing [26]. The experiments were conducted on one NVIDIA V100 GPU. All GNN models have five layers. We use the Adam optimizer with learning rate 1e-3 for training.

Table 10: GNN results for code classification.

	C++1000	C++1400	Java250	Python
GCN	91.49%	90.18%	88.92%	91.64%
GCN-V	92.29%	91.18%	90.79%	92.37%
GIN	93.04%	92.05%	89.12%	91.35%
GIN-V	93.61%	92.85%	90.27%	92.17%

Table 10 compares the performance of GNNs on four benchmarks. GCN-V works best for JAVA and Python benchmarks and GIN-V works best for the C++ benchmarks. Overall, adding a virtual node improves GNN performance. GIN variants work better when the number of classes is large.

C Details of Experiments on Code Similarity

C.1 MLP with Bag of Tokens

For experiments on code similarity analysis, we use the same bag of tokens for code classification. The input to the neural network is constructed by concatenating two bags of tokens, one for each source code file.

Table 11 provides results of code similarity analysis on all four benchmarks. The columns give the benchmark name, the test accuracy, the number of training epochs, the number of samples in each epoch, the run time of each epoch, the number of token types considered, and the number of test samples. All networks are implemented using Keras API. The training is performed on a single V100 GPU.

Table 11: Similarity analysis by MLP with bag of tokens.

Benchmark dataset	Accuracy %%	Number epochs	Size of epoch	Run time sec/epoch	Number tokens	N test samples
Java	82.48	20	4,096,000	21	81	512,000
python	86.64	94	4,096,000	24	71	512,000
C++1000	85.69	64	4,096,000	21	56	512,000
C++1400	86.47	64	4,096,000	22	56	512,000

Figure 6 shows the neural network used for code similarity analysis on the C++ 1400 benchmark. It has three dense layers, two of which have ReLU activation. The last dense layer has no activation since its output goes directly to a sigmoid layer, predicting the similarity score.

As we see in Table 11, the model accuracy is modest 86.47%, which is not very high for a binary classification problem of a fully balanced dataset. Obviously, the bag of tokens is too primitive and misses many important details necessary for identifying similarity.

The neural networks used for code similarity analysis on other benchmarks are similar to the one shown in Figure 6. As we see in Table 11, their accuracy is similar.

C.2 Siamese Network with Token Sequence

For experiments on code similarity, we use the same sequence of tokens for code classification. The neural network has two inputs, one for each source code file. After experimenting with various neural network architectures, we select the siamese network for its good performance.

Table 12 provides results of code similarity analysis on all four benchmarks. The columns give the benchmark name, the test accuracy, the number of training epochs, the number of samples in each epoch, the run time of each epoch, the number of token types considered, and the number of test samples. All networks are implemented using Keras API. The training is performed on four V100 GPUs. The training epochs are much fewer than the code classification case.

The neural network for the C++ 1400 benchmark is depicted in Figure 7. The siamese parts of the network have the same structure and share all their weights. If the inputs are identical, so are the outputs. Therefore, by construction, the network guarantees detecting similarity of identical source code samples.

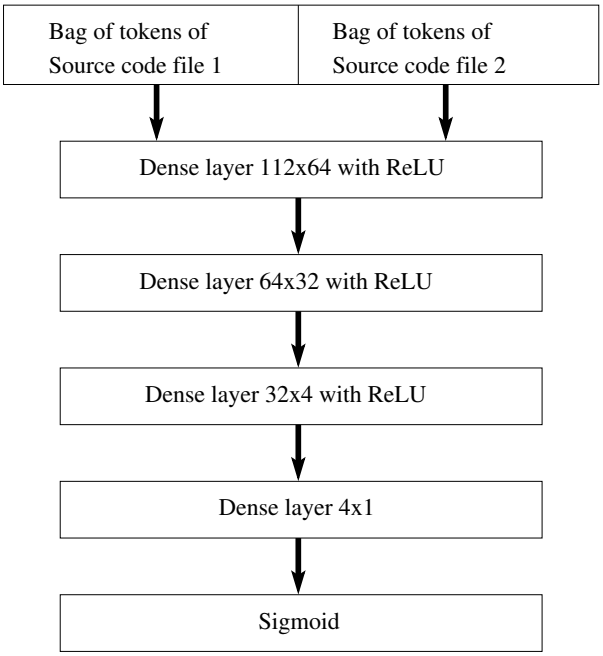


Figure 6: MLP architecture for similarity analysis.

Table 12: Similarity analysis by Siamese network with token sequence.

Benchmark dataset	Accuracy %%	Number epochs	Size of epoch	Run time sec/epoch	Number tokens	N test samples
Java	89.39	29	51,200	114	75	512,000
python	94.86	110	64,000	89	71	512,000
C++1000	96.70	123	64,000	89	56	512,000
C++1400	96.36	144	64,000	96	56	512,000

The siamese parts of the network are similar to the neural network used for code classification (Figure 5). Their first layer is a trainable embedding layer. It converts tokens into embedding vectors. The embedding vectors are initialized randomly and learned.

The next three layers of the siamese part are convolution layers. Their kernels extract the most distinctive features of the source code. Two of these layers have ReLU activation. The last convolution layer has no activation since its output goes to a max-pooling layer. The max-pooling layer selects dominant occurrences of the features. The output of the max-pooling layer is processed with a dropout layer, which is the last layer of the siamese part of the network. We inserted a dropout layer to mitigate over-fitting.

The outputs of the siamese parts are compared by computing the absolute difference. The result is supplied to a stack of two dense layers. One of them has ReLU activation. The last dense layer has no activation since its output goes directly to a sigmoid layer, computing the similarity score.

The network shows 96.36% test accuracy. We consider this a good result, especially considering that the token sequence ignores all identifiers, comments, and many keywords.

The neural networks used for code similarity analysis of other benchmarks are similar to the one shown in Figure 7. As we see in Table 12, their accuracy is quite similar.

C.3 SPT Based Methods

We use a rule-based model and a learning-based model, both with the SPT code representation, to perform code similarity analysis. The rule-based model i) normalizes SPT node names to eliminate

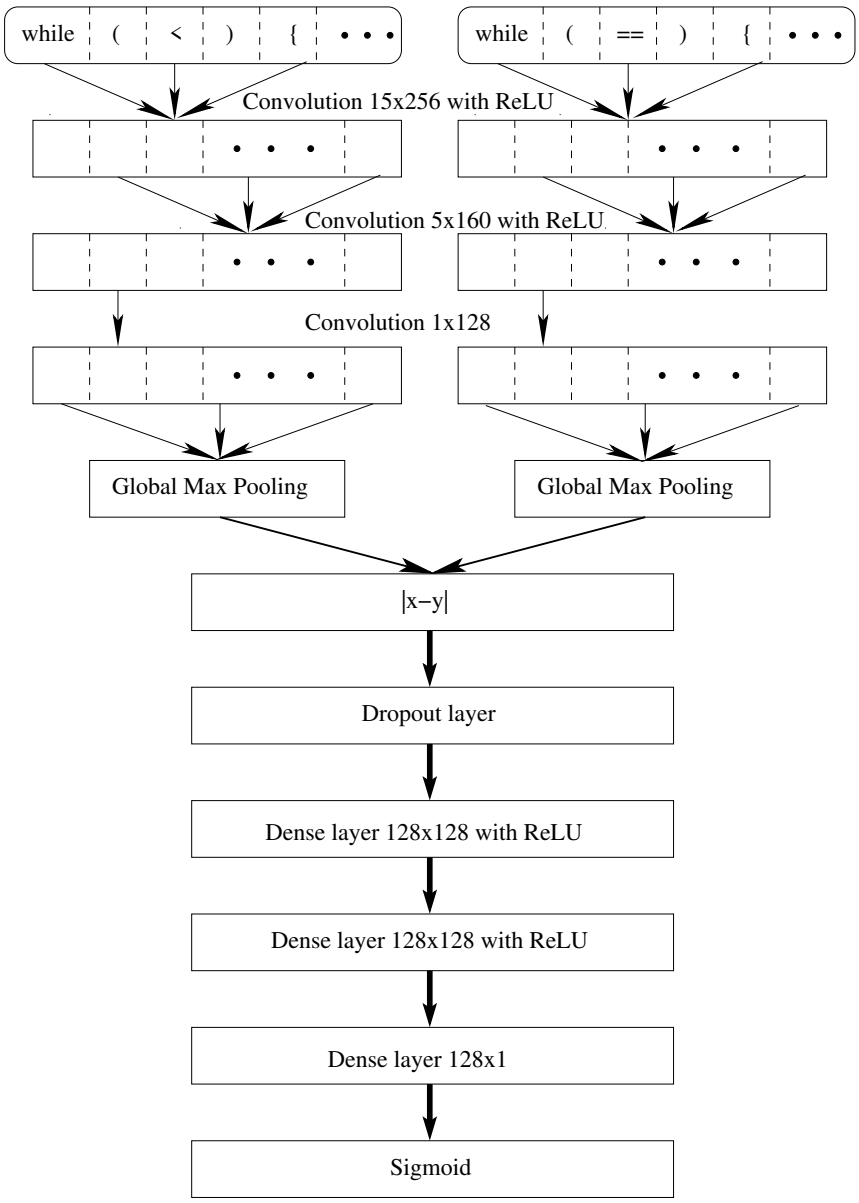


Figure 7: Siamese architecture for similarity analysis.

code-specific details such as variable names and local function names, and ii) uses the normalized node names and manually specified rules to extract features. These features are then used to create a feature vector for the code. The similarity of two code samples is then computed using distance metrics such as dot product or cosine distance. AROMA [23] is one such rule-based technique that we use.

The learning-based model also normalizes SPT node names as the first step, but instead of manually specifying rules to create a feature vector, it uses a trainable model to create a feature vector. MISIM [18] i) converts the source code into a context aware semantic structure (CASS), ii) builds and trains neural network models such as recurrent neural networks (RNN) or graph neural networks (GNN), and iii) uses cosine distance of the vectors in the final layer to compute similarity. In our experiments, we use a GNN model in step ii, since it is demonstrated to achieve good results for code similarity [18] on POJ-104 [35] and GCJ-297 [17, 36] datasets.

We use the mean average precision at R (MAP@R) to compare the quality of the two models. We refer the reader to [37] for details on how MAP@R is computed.

Table 13 reports the results. Due to the scale and variety of the problems, the rule-based technique results in significantly lower quality compared to the learning-based technique. Note that due to the high memory requirements of computing MAP@R over the test set, we need to modify AROMA to use a smaller vocabulary. We estimate that this modification reduces MAP@R score of AROMA by about 15%.

Table 13: MAP@R score for similarity analysis with SPT based methods.

Dataset/Model	AROMA	MISIM
C++ 1000	0.17	0.75
C++ 1400	0.15	0.75

C.4 Discussions

Models trained on the CodeNet benchmarks can benefit greatly due to the high quality of the dataset. To demonstrate this, we compare CodeNet-1K (the C++ 1000 benchmark) to one of the largest publicly available dataset, GCJ-297. For the purpose of this comparison, we train the same MISIM model on CodeNet-1K and GCJ-297 and test the two trained models on a third, independent dataset - POJ-104. The result of this comparison is plotted in Figure 8.

The x-axis of this plot is the number of training epochs used and the y-axis is the MAP@R score. The MISIM model for both datasets is trained for 500 epochs and the MAP@R score for validation and test is computed after every other ten epochs. There are a total of four curves - a validation and a test curve for GCJ-297 and a validation and a test curve for CodeNet-1K.

The four curves show that a 10% higher validation score can be achieved with GCJ-297 as compared to CodeNet-1K. However, when tested on POJ-104, the model trained on GCJ-297 achieves a 12% lower score as compared to the model trained on CodeNet-1K.

CodeNet-1K has better generalization than GCJ-297 mainly for two reasons: i) high data bias in GCJ-297 (as shown in Figure 9) and ii) cleaning and de-duplication of submissions in CodeNet-1K (as described in Section 5.2).

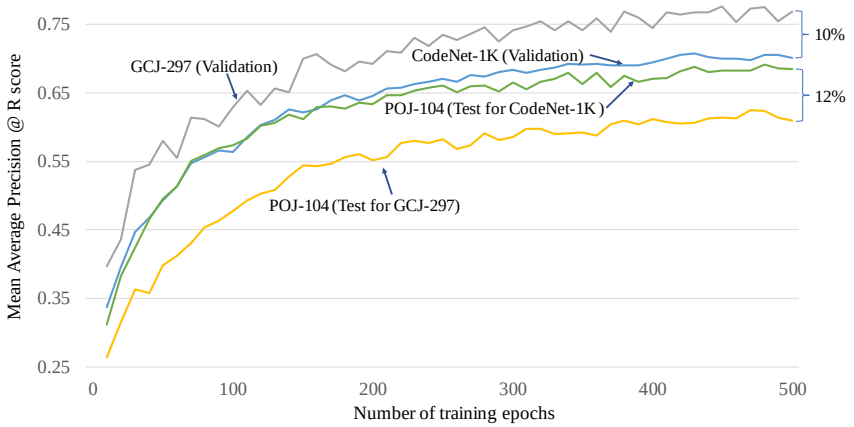


Figure 8: Test score on POJ-104 is 12% higher when a model is trained on CodeNet-1K than when it is trained on GCJ-297, even though validation score on GCJ-297 is 10% higher than validation score on CodeNet-1K.

Figure 9 shows a bar chart of the number of submissions per problem in GCJ-297, with problems being sorted in descending order of number of submissions. The chart suggests that the top 20 problems with most number of submissions account for 50% of all submissions. This data bias in

GCJ-297 can be detrimental to model quality, because the model is trained mainly on features from a small set of problems.

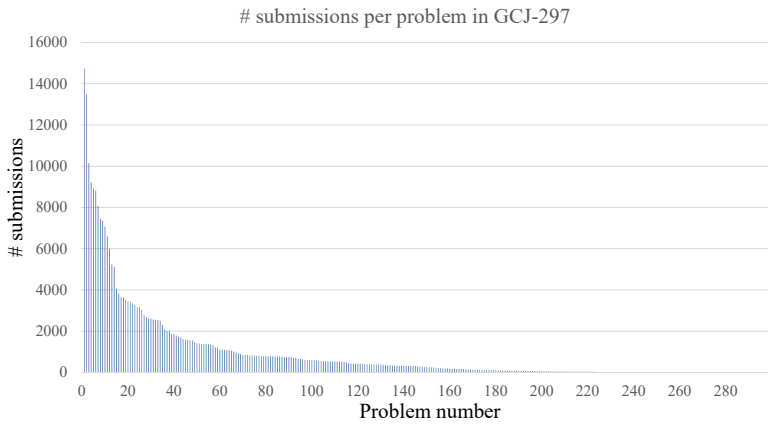


Figure 9: Number of submissions for each of the 297 problems in GCJ-297. The 20 problems with most number of submissions account for 50% of all submissions.

D Details of MLM Experiment

Here we show how a masked language model (MLM) can be trained with CodeNet. We closely follow the approach by Ankur Singh, documented in the blog [38]. The goal of the model is to infer the correct token for an arbitrary masked-out location in the source text. We assume that in every text, precisely one token is randomly masked. The original token at such position is then the golden label.

From each of the 1000 CodeNet-1K problems, we randomly select 100 samples for training and another 100 for testing. Each C++ source file is tokenized into a vocabulary of 442 distinct tokens as categorized in Table 14. For example, `while` is a keyword and `strlen` is a function literal.

Table 14: Token categories used for MLM.

Type	Count	Description
the keyword	95	all C++20 reserved words
the function	280	function names in common header files
the identifier	42	standard identifiers, like stderr, etc.
the punctuator	16	small set of punctuation symbols
# or ##	2	the C pre-processor symbols
0, 1	2	special case for these frequent constants
the token class	5	identifier, number, operator, character, string

This code snippet:

```
for (i = 0; i < strlen(s); i++) {}
```

will be tokenized to:

```
for ( id = 0 ; id < strlen ( id ) ; id operator ) { }
```

The tokenized source files are read into a pandas dataframe and processed by the Keras Text Vectorization layer, to extract a vocabulary and encode all token lines into vocabulary indices, including the special "[mask]" token. Each sample has a fixed token length of 256. The average number of tokens per sample across the training set is 474. Short samples are padded with 0 and those that are too large are simply truncated.

The model is trained with 100,000 samples in batches of 32 over five epochs, with a learning rate 0.001 using the Adam optimizer. We evaluate the trained model on a test set of 100,000 samples. Each sample is pre-processed in the same way as the training samples and one token (never a padding) is arbitrarily replaced by the "[mask]" symbol. Then, a prediction is generated and the top 1 and top 5 results are compared with the expected value. The achieved accuracies are top-1: 0.9068 and top-5: 0.9935.