Representation Learning for Source Code Modeling: Challenges, Progress, and Prospects

Zhimin Zhao
Student number: 20311734
School of Computing, Queen's University
Kingston, ON, Canada
z.zhao@queensu.ca

***Abstract*—Representation learning has been proven to play an significant role in the unprecedented success of a series of application, such as speech recognition, natural language processing, and computer vision. In recent years, researchers have made a lot of progress in applying such a learning paradigm in modeling source code and deploy it in the real-world tasks such as code auto-completion. In this paper, we conduct a literature review of representation learning for source code modeling. Our study shows that such a research domain is evolving fast and have much potential to push forward the frontier of both artificial intelligence and software engineering.**

I. Introduction

"Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do. [1]"

—— Donald Knuth, American computer scientist

Like natural language, programming is a form of communication among developers. Such inspiration is first validated and explored by Naturalness Hypothesis [2] which holds that software programs share similar statistical properties to natural language corpus and these properties can be learned and exploited to better assist software developers. In such a context, representation learning, aiming to learn compact, numerical representations for raw data, has been applied in the modeling of source code [1].

With the recent success of representational learning, moving from natural language words, sentences, and paragraphs to programming language tokens, statements, and blocks is natural. In the past few years, researchers have achieved encouraging experimental results [3]–[72] and even deployed their models in the real-world settings [2] [3] [4]. This paper presents a literature review of representation learning for source code modeling, aka Big Code [5]. Our contributions are:

- A review of word embedding techniques for source code modeling and their systematic categorization
- A review of benchmarks, datasets, and evaluation metrics for source code modeling
- A summary of current challenges and proposal for future research directions

The remainder of this paper is structured as follows. Section II discusses the preliminaries of representation learning and programming language and related surveys of representational learning of source code. Section III presents our review protocol and the bibliometrics of our findings. Section IV talks about the benchmarks, datasets, and measurements for source code modeling. Section V compares the modeling techniques and categorizes them systematically. Section VI summarizes the major study implications to practitioners, researchers, and educators. Section VII discusses some important threats to the study validity and the means we used for coping with these threats whenever possible. Finally, Section VIII concludes this paper and suggests future work.

II. Background and Related Works

In this section, we would talk about the background of our study to make it self-contained. Section II-A details the preliminaries of representation learning. Section II-B presents and the traditional source code modeling techniques. Section II-C compares the previous works on big code.

A. Preliminaries of Representation Learning

This section reviews fundamental terminology in representation learning so as make the review self-contained.

Representation learning is a subdivision of artificial intelligence that focuses on extracting compact, numerical representations (i.e. vector of real numbers) for sources of signal [73]. Basically, representation learning covers the domain of deep learning (DL) since the implementation of the former can be either shallow or deep. A representation learning system is typically composed of the following elements:

- **Data instance**: A piece of digitalized signal encoding information about an object.
- **Feature**: An observable and measurable property of data instances.
- **Neural network**: A multi-layered architecture of non-linear processing units for feature extraction and pattern recognition.

[1] We interchangeably use source code and computer program here. Their trivial differences are out of scope in our paper.
[2] kyte.ai
[3] www.deepcode.ai
[4] www.aixcoder.com
[5] ml4code.github.io

- **Model**: The neural network artifact that encodes prediction logic for a target task.
- **Embedding**: A mapping from semantic units to a vector of real numbers. Embeddings are the artifacts of big code models, which are expected to preserve the semantics of the represented units.
- **Encoder-decoder framework**: As shown in Figure 1, it is a model that learns the latent representation to reconstruct code or text as the output for a target task.

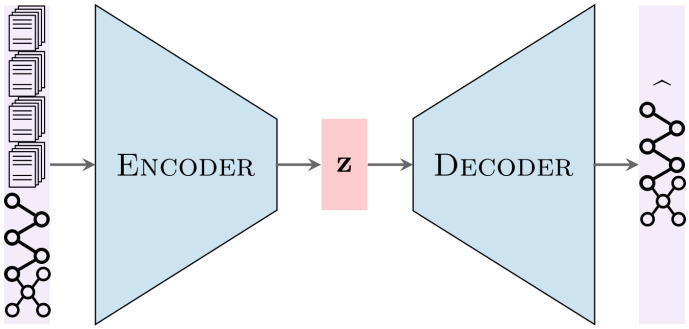

Fig. 1: Encoder-Decoder Framework [74]

- **Dataset**: A set of data instances for learning a representation model. The data could be categorized as:
  - **Training data**: The data used to learn the neural network parameters
  - **Validation data**: The data used to tune the neural network hyper-parameters
  - **Test data**: The data used to validate the neural network performance

There are different types of representation learning. Based on the training data characteristics, representation learning covers:

- **Supervised learning**: an approach that learns models on labeled training data as learning targets.
- **Supervised learning**: an approach that learns models on unlabeled training data and aims to understand the data itself.
- **Reinforcement learning**: an approach that learns models on sequential training data and aims to interact in a specific environment to maximize the rewards.

B. Traditional Modeling Techniques

Source code modeling aims to model the syntax or semantics of source code and apply the learned models in the specific programming task. In the pre-DL era, researchers resort to techniques such as probabilistic grammar to complete the static analysis. The rest of this section would brief a bit on each style of modeling technique.

a) Domain-specific language (DSL)-guided models: defines a set of parametric program template for common code statements in the DSL and use them to generate the structure of the program as needed. Such models are widely used in program induction, wherein the induction is commonly converted as a Constraint Satisfaction Problem (CSP). CSP as a search problem is NP-hard and usually solved by heuristic techniques such as MAC and FC algorithms [75].

b) Context Free Grammar (CFG)-guided models: define the production rules of a context-free language in a probabilistic manner. Such models allow the discovery of more complex structures compared to DSL-guided models. Algorithms such as Tree Substitution Grammar (TSG) [76] and Probabilistic Context-free Grammar (PCFG) [77] belong to this category.

c) n-gram language models: is a probabilistic model which assumes each token is conditionally dependent on the previous $n-1$ tokens (n is the length of truncation). The difference between the n-gram and Markov model is that the former discards positional information in the truncation.

With deep domain knowledge incorporated, DSL-guided models and CFG-guided models can effectively capture the dependencies among sequential symbols. However, the strictly defined rules make the models generalize poorly in new areas. Though n-gram is easier to generalize, it is poor in capturing long-term dependencies between tokens. Their strengths and weaknesses can be found in Table I.

TABLE I: Strengths and Weaknesses of Traditional Source Code Modeling Techniques

Technique	Strength	Weakness
DSL-guided models	• small grammar size • human interpretable • structure capturing	• poor scalability • deep domain knowledge dependent • intolerant of noisy, erroneous, or ambiguous data
CFG-guided models	• automatic model selection • high perfomance	• poor scalability • manual design dependent
n-gram language models	• easy to generalize • simple to implement • high scalability	• poor in modeling high-level programming paradigms • poor in capturing long-term dependency • high sparsity of token vector

C. Related Works

Chen and Monperrus [9] collect and categorize code embeddings and their applications in big code 5. other embeddings. Wright and Wiklicky [78] compare the effectiveness of natural language models and static analysis graph-based models in representing source code. Wang [79] presents an evaluation of several language models on software defect datasets. Karampatsis et al. [61] discuss how different source code modelling techniques impact the resulting vocabulary on large-scale software projects. Similar works can be found in Wainakh et al. [80]. Le et al. [81] provide a literature review to elaborate the existing deep learning techniques for source code modelling. Compared with the previous studies, our work also provides a more comprehensive overview of the techniques, benchmarks, datasets, and measurements for big code.

III. Collection Overview

We apply the following inclusion criteria when collecting papers. If a paper satisfy any one or more of the following criteria, we will include it.

1) The paper discuss the general idea or related aspects of Big Code.

2) The paper proposes a representation learning technique that aims to model source code.
3) The paper presents a dataset or benchmark specifically designed for the purpose of Big Code.

To collect the papers as many as possible, we started by using combinatorial keyword search string on popular scientific databases which include: 1) Google Scholar, 2) Semantic Scholar, and 3) DBLP. The keywords used for searching are listed below:

- big code
- neural program embedding
- representation learning + code | program | software
- deep learning + code | program | software
- neural network + code | program | software

To ensure a more comprehensive and accurate literature review, we conduct both forward and backward snowballing and include as many as papers possible in our collection. We collected more than 200 papers in our initial retrieval. After filtering, we keep only 70 papers as these are the most relevant papers on the topic of big code [6]. Figure 2 shows the number of annual publications since the invention of big code until Mar. 2022. We can see that around 63% of papers have appeared in the last three years, testifying to the emergence of a new source code modeling domain of interest: *big code*.

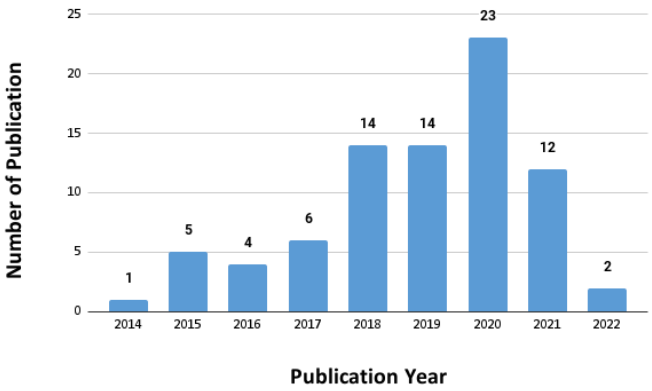

Fig. 2: Big Code Publication with time until Mar. 2022

Figure 3 shows the distribution of paper collection for different publication purposes in big code. Among all the papers, 69.1% papers are published for presenting a new technique. Benchmark papers account for 11.1%, while dataset papers take around 12.3%. The rest of the papers only account for about 7.4%.

Figure 4 shows the distribution of paper collection for different tasks in big code. Generative tasks account for 95.2% among all the papers, while discriminative tasks only take 16.7%. We can see that their sum is more than 100% since a great proportion of the papers aim to solve both tasks in their experiments. This phenomenon somewhat showcases the good generalization of big code models.

[6]Our paper catalog can be found in bit.ly/3NqaGon

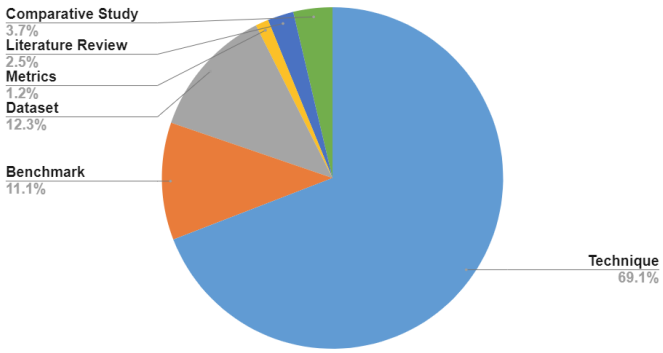

Fig. 3: Big Code Publication based on Purposes

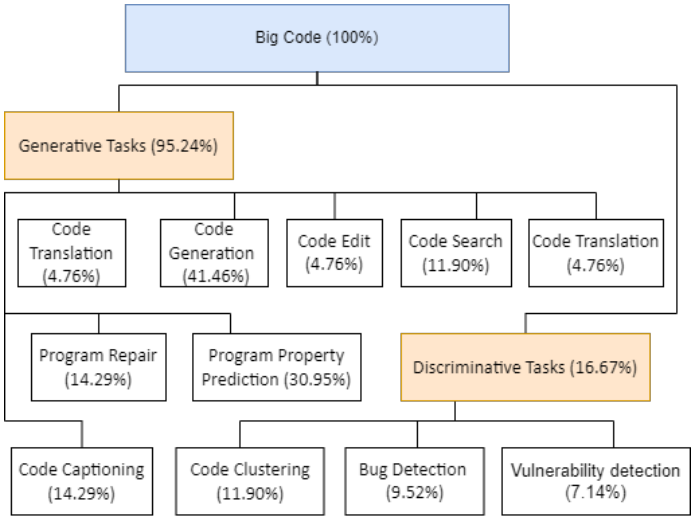

Fig. 4: Big Code Publication based on Tasks

IV. Benchmarks, Datasets, and Measurements

1) Benchmark: In this section, we would elaborate the benchmarks for big code.

- **IdBench** [80] is the first benchmark for evaluating to what extent token embedding represents semantic similarity and relatedness. The benchmark is based on thousands of ratings gathered by surveying 500 software developers.
- **COSET** [16] is a benchmark for evaluating program semantics and characteristics. COSET consists of a dataset of nearly eighty-five thousand programs and labeled with a number of characteristics of interest (e.g. running time, pre- and post-condition, loop invariant).
- **BLANCA** is a benchmarks for language models on coding artifacts. The dataset describes code artifacts in Python and links 1.3 million programs to associated posts and class-documentation. It also include five tasks related to Python usages.
- **APPS** is a benchmark for code generation. It intends to measure the ability of models to take an arbitrary natural language specification and generate satisfactory Python code. The dataset includes ten thousand triplets of problem, generated code and test cases.
- **CodeXGLUE** [15] is a general language understanding

evaluation benchmark for source code developed by Microsoft Research Asia. As shown in Figure 5, it includes a collection of 10 tasks across 14 datasets and a platform for model evaluation and comparison.

- **BigCloneBench** [18] is a clone detection benchmark of known clones in the IJaDataset source repository. The dataset covers ten functionalities (a.k.a. cases) including 6 million true clone pairs and 260 thousand false clone pairs which are mined from the big data inter-project repository IJaDataset 2.0.
- **CodeSearchNet** [29] is a collection of datasets and benchmarks that explore the problem of code retrieval using natural language. It consists of 2 million code-comment pairs from open source libraries which support Python, Javascript, Ruby, Go, Java, and PHP.

Category	Task	Dataset Name	Language	Train/Dev/Test Size	Baselines	Task definition
Code-Code	Clone Detection	BigCloneBench	Java	900K/416K/416K	CodeBERT	Predict semantic equivalence for a pair of codes.
		POJ-104	C/C++	32K/8K/12K		Retrieve semantically similar codes.
	Defect Detection	Devign	C	21k/2.7k/2.7k		Identify whether a function is vulnerable.
	Cloze Test	CT-all	Python, Java, PHP, JavaScript, Ruby, Go	-/-/176k		Tokens to be predicted come from the entire vocab.
		CT-max/min	Python, Java, PHP, JavaScript, Ruby, Go	-/-/2.6k		Tokens to be predicted come from {max, min}.
	Code Completion	PY150	Python	100k/5k/50k	CodeGPT	Predict following tokens given contexts of codes.
		GitHub Java Corpus	Java	13k/7k/8k		
	Code Repair	Bugs2Fix	Java	98K/12K/12K	Encoder-Decoder	Automatically refine codes by fixing bugs.
	Code Translation	CodeTrans	Java-C#	10K/0.5K/1K		Translate the codes from one programming language to another programming language.
Text-Code	NL Code Search	CodeSearchNet, AdvTest	Python	251K/9.6K/19K	CodeBERT	Given a natural language query as input, find semantically similar codes.
		CodeSearchNet, WebQueryTest	Python	251K/9.6K/1k		Given a pair of natural language and code, predict whether they are relevant or not.
	Text-to-Code Generation	CONCODE	Java	100K/2K/2K	CodeGPT	Given a natural language docstring/comment as input, generate a code.
Code-Text	Code Summarization	CodeSearchNet	Python, Java, PHP, JavaScript, Ruby, Go	908K/45K/53K	Encoder-Decoder	Given a code, generate its natural language docstring/comment.
Text-Text	Documentation Translation	Microsoft Docs	English-Latvian/Danish/Norwegian/Chinese	156K/4K/4K		Translate code documentation between human languages (e.g. En-Zh), intended to test low-resource multi-lingual translation.

Fig. 5: A brief summary of CodeXGLUE, including tasks, datasets, language, sizes in various states, baseline systems, providers, and short definitions of each task.

2) Datasets: In this section, we would detail the datasets for big code.

- **WikiSQL** [21] is a dataset of 80,654 hand-annotated natural language questions and SQL queries across 24,241 HTML tables from Wikipedia. On WikiSQL, big code models generate a SQL query from a natural language question and table schema.
- **CoNaLa** [24] is a dataset of code/natural language challenge jointly developed by the NeuLab and Strudel labs in Carnegie Mellon University. It consists of 2,879 annotated examples and more than 600k raw examples. On CoNaLa, big code models can test the generation of code snippets from natural language.
- **CoNaLa-Ext** [19] is an enhanced version of CoNaLa, in which every example has the full question body from its respective StackOverflow Question. It has ten thousand annotated examples wherein big code models can generate code snippets more accurately based on both the intent and question body.
- **Django** [14] is a dataset for generating pseudo-code from source code. It consists of six thousand training, one thousand development and eighteen hundred test annotations. Each data point consists of a line of Python code together with a manually created natural language description.
- **CONCODE** [7] is a dataset with over a hundred thousand examples consisting of Java class from online code repositories. On CONCODE, researchers can develop a new encoder-decoder architecture that models the interaction between the method documentation and the class environment.
- **Devign** [27] is a manually labeled dataset built on 4 diversified large-scale open-source C projects that incorporate high complexity and variety of real source code. It has totally 48,687 commits with confirmed labels, 12811 from Linux, 13962 from FFmpeg, 11910 from Qemu, and 10004 from Wireshark.
- **ETH Py150** [8] is a deduplicated dataset of Python files from Github. It is released to create new kinds of programming tools and techniques based on machine learning and statistical models learned over massive codebases.
- **PyTorrent** [13] is a dataset of Python library corpus for large-scale language models. It contains 218,814 Python package libraries from PyPI and Anaconda environment.
- **notebookcdg** [28] is a dataset mined from Top 10% highly-voted notebooks from the top 20 popular competitions on Kaggle. It contains approximately 28k processed code-documentation pairs extracted from 2,476 highly-ranked notebooks.
- **ParallelCorpus-Python** [31] is a parallel corpus of 150,370 triples of function declarations, function docstrings and function bodies assembled from open source GitHub repositories. The Python code is preprocessed to normalize the syntax, extract top-level functions, remove comments and semantically irrelevant whitespaces, and separate declarations, docstrings (if present) and bodies.
- **DeepCom-Java** [12] is a large-scale Java corpus built from 9,714 open source projects from GitHub. Each pair of data contains a Java method and code comment.
- **Hybrid-DeepCom-Java** [11] is a further version of DeepCom-Java, commonly used to evaluate automated code summarization. It contains a hundred thousands Python functions with their documentation strings ("docstrings") generated by scraping open source repositories on GitHub.

3) Measurements: In this section, we would elaborate the evaluation metrics for big code. Since automatic metrics do not always agree with the actual quality of the results [82], human evaluation studies are sometimes necessary to evaluate the performance of the trained model. Compared with automatic evaluation, human evaluation are often based on a numeric rating scale, e.g. Likert scale [83].

a) Automatic Evaluation:

- **Accuracy** is the proportion of correct predictions (both true positives and true negatives) among the total number of cases examined. It measures how close or far a given set of observations is to their true value.
- **Precision** is the fraction of retrieved documents that are relevant to the query. It can be measured as of the

[7] github.com/sriniiyer/concode
[8] www.sri.inf.ethz.ch/py150

total actual positive cases, i.e. how many positive class predictions belong to the positive class.
- **Recall** is the fraction of the relevant documents that are successfully retrieved. It can be measured as the positive predictive value (PPV), i.e. how many positives were predicted correctly.
- F_1**-score** is the harmonic mean of precision and recall.
- **Confusion matrix** is a table to describe the performance of a classifier on a set of test data for which we know the true values.
- **AUC** stands for the area under the curve. It measures the ability of a classifier to distinguish between classes. The higher the AUC, the better the classification performance of the model.
- **METEOR** [84] is short for Metric for Evaluation of Translation with Explicit ORdering. It is used for the evaluation of the machine translation output. The metric is based on the harmonic mean of unigram precision and recall, with recall weighted higher than precision.
- **BLEU** [85] is short for bilingual evaluation understudy. It is used for evaluating the quality of the translated text. The metric is based on the n-gram overlap between the machine translation output and reference translation.
- **CodeBLEU** [15] is a method for automatic evaluation of program synthesis. It absorbs the strength of BLEU in the n-gram match and further injects code syntax via data-flow graphs.
- **Exact match** [86] measures the percentage of predictions that match any one of the ground truth answers exactly.
- k**-adversaries** [10] measures the *k-robustness* of a big code model. When an input program undergoes a pre-specified set of k (e.g. 2) semantics-preserving transformations, the adversary succeeds if it manages to change the inference of the model.
- **MRR** [87] is short for Mean Reciprocal Rank. It is a measure to evaluate any process that produces a list of possible responses to a sample of queries, ordered by probability of correctness.
- **MAP** [88] is short for Mean Average Precision. Average Precision (AP) is a popular metric in measuring the accuracy of object detectors like Faster R-CNN, SSD, etc. MAP is the average of AP for each class.
- **NDCG** [89] is short for Normalized Discounted Cumulative Gain. It is a metric of ranking quality that is commonly used to measure effectiveness of recommendation systems.

b) Human Evaluation:

- **Naturalness** [14] measures grammaticality and fluency of the output artifacts (code, natural language summary etc.).
- **Informativeness** [53] measures amount of content carried over from the input code to the natural language summary, or vice versa.

V. Modeling Techniques

In this section, we would present the techniques behind big code.

A. Embedding

The task of embedding commonly involves encoding as much information about a program text as possible, and then the ability to regenerate a program from that information as correctly as possible. Literally, any semantic unit can be encoded in the big code model:

- **Variable**: It can be the name or type.
- **Function**: It can be a call or definition.
- **Statement**: It can be a loop, conditional, or sequential control directive.
- **Snippet**: It can be a code block within a defined scope.
- **Module**: It can be an independent functionality of a program.
- **File**: It can be a source file.
- **Binary**: It can an executable program.
- **Exception**: It can be a warning or error message.
- **IR**: It can be an intermediate representation instruction.
- **Edit**: It can be a change to the code.
- **Trace**: It can be a execution trace.
- **IPC**: It can be an inter-process communication call.
- **Algebraic expression**: It can be a mathematical formula.

B. Pipeline

Illustrated in Figure 6, the general pipeline to learn the embeddings is usually an encoder-decoder framework. In the training phase, program text will be fed, processed, and encoded as embeddings. In the inference phase, embeddings will be fed, processed, and decoded back to program text. Sometimes, the model does not directly predict the next token. Instead, it utilizes the predicted type to assist the token prediction [32].

C. I/O

We can further categorize the models based on the modality of input/output (I/O) program text. Figure 7 demonstrates the Top 3 commonly used I/O modalities in big code models. For example, *Tree2Graph* indicates that we should feed tree fragments (e.g. abstract syntax tree) to the model and expect to get graph fragments (e.g. control flow graph) as the output. Sequence structure is commonly used in natural language processing since we can treat a code sequence as a natural language sentence. Tree structure (parse tree etc.) comes from the preprocessing of program text via a tree-walking interpreter or counterpart. Graph structure (program dependence graph, etc.) comes from further preprocessing of tree structure via static program analysis or other modelling tools.

D. Architecture

Theoretically, any neural network that fits with the encoder-decoder framework can be used for learning the embeddings of the program text. Based on the efficiency of data usage, we can categorize the architecture of neural networks into two categories: 1) multi-task, 2) single-task. Multi-task models are usually pre-trained models in which multiple learning tasks are solved at the same time while exploiting commonalities and differences across tasks [90]. In contrast, single-task

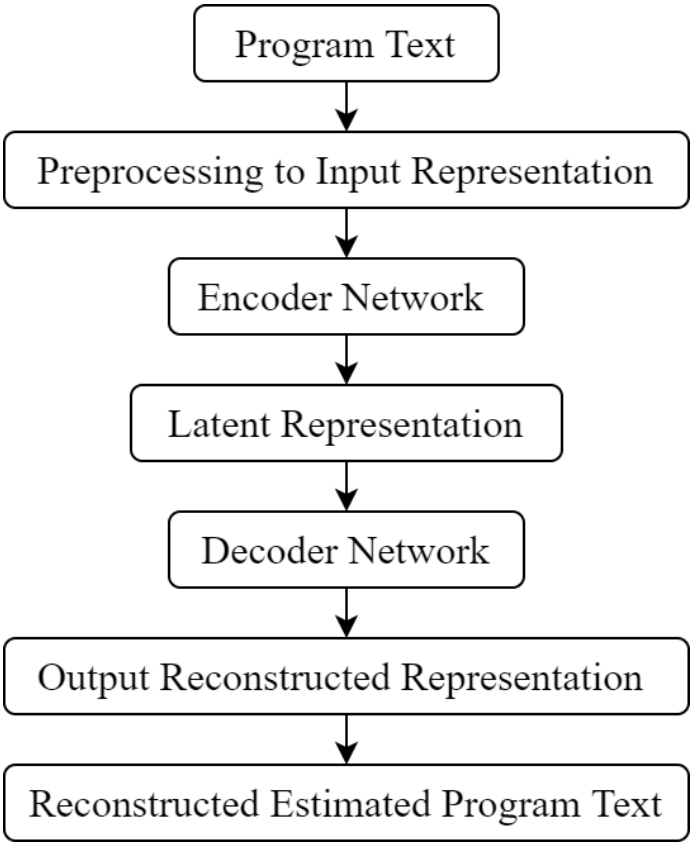

Fig. 6: General Embedding Pipeline [78]

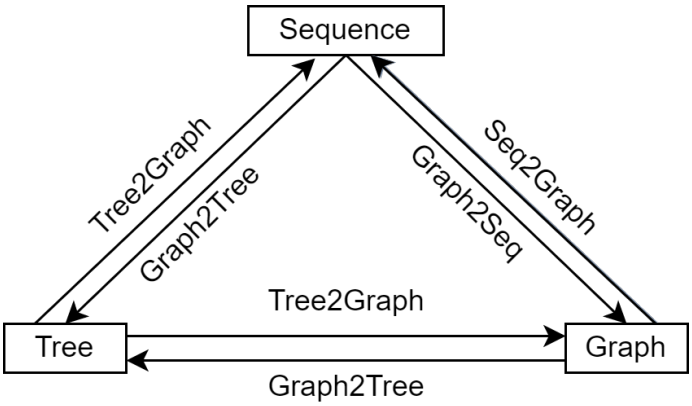

Fig. 7: Model I/O Modality

models are trained with a single purpose. However, when two tasks share a lot of commonalities. Single-task models can sometimes work pretty well for the new task since the embeddings are expected to be unchanged or little-changed in both tasks.

Figure 8 shows a recent example of AlphaCode [17]. AlphaCode can read the whole problem statement and produces code. It achieved an estimated rank within the top 54% of participants in Codeforces [9] by solving new problems that require a combination of critical thinking, logic, algorithms, coding, and natural language understanding. Here, AlphaCode uses Transformer-based language models (which belong to pre-trained models) to generate code and then filters to a small set of candidate programs. Other multi-task models are found in related works [7], [8], [11], [12], [18], [31], [36], [38], [50], [54]–[56], [72].

[9]A website that hosts competitive programming contests worldwide

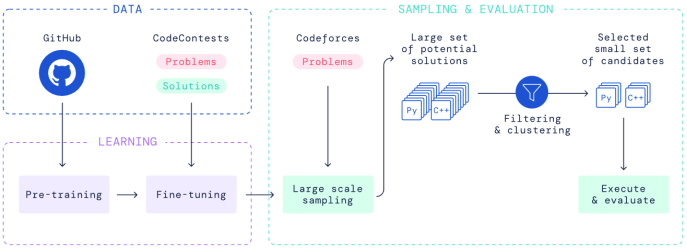

Fig. 8: AlphaCode Pipeline

Single-task models are often neural networks with limited learnability compared with multi-task ones. Learnability of a model is dependent on the characteristics of network architecture such as the topology (e.g. GNN, CNN, RNN), the size (e.g. number of hidden layers,), the hyper-parameters, the optimizer. The implementation details of each network topology are out of the scope of this paper. Thus, we only categorize the modeling techniques in the literature based on their network topology as shown in Table II.

TABLE II: Literature Categorization based on Network Topology

Network Topology	Literature
Graph Neural Network	[5], [6], [16], [27], [33], [37], [40], [47], [79]
Recurrent Neural Network	[10], [20]–[22], [45], [53], [62], [63], [70]
Transformer	[24], [42], [43], [48]
Convolutional Neural Network	[21], [44]
Reinforcement Learning Network	[4], [21]
Neuro-symbolic network	[3], [20]

VI. Discussions

This section discusses the challenges VI-A and research opportunities VI-B in big code.

A. Challenges

Based on our readings, we find a series of challenges related with big code:

- Existing models learn static embeddings regardless of program context. However, the differences in the meaning of a token in varying contexts are lost when each token is associated with a single representation. Particularly, most techniques are usually formed from recognized natural events and therefore, are incapable of suggesting zero-shot (unseen) code tokens.
- Type information is ignored or unrestricted in most of the existing models, which, however, is a significant attribute in the traditional rule-based models.
- Existing models lack the learnability of formal methods. Combinatorial concepts found in formal methods, such as sets and lattices lack direct analogues in deep learning [91]. That also leads to poor explainability of model inference.

- Existing models are vulnerable to adversarial attacks. Traditional program analyses offer explicit guarantees about a program's behavior even within adversarial settings. Representation learning-based program analyses relax many of those guarantees towards reducing false positives or aiming to provide some value beyond the one offered by formal methods (e.g. use ambiguous, noisy, erroneous information).
- Data efficiency is low in downstream task learning. Existing pre-training models are often trained on datasets labeled for specific downstream tasks, while the code representations may not be suitable for other tasks.
- Existing models are poor in capturing long-term context dependencies. This is an innate pain point of gradient descent when temporal contingencies present in the input/output sequences span long intervals [92].
- Most existing representation learning-based program analysis technique either use unsupervised/self-supervised proxy objectives or make use of relatively large datasets of annotated code. However, a number of the desired program analyses do not fit such frameworks and would require at least some form of weak supervision.
- Current techniques lack to illustrate, how they can assist programmers, in a real-world software development environment.

B. Opportunities

Based on our readings, we find a series of promising research directions following the aforementioned challenges:

- Develop more effective contextual embedding schemas. How to learn the contextual information effectively remains a hot topic in natural language processing for the recent years.
- Infuse formal knowledge into the learning process. Most specification inference researches treat the formal analyses as a separate pre- or post-processing step. However, learning richer combinatorial — and possibly non-parametric — representations will provide valuable alternatives for big code.
- Improve the explainability of the inferences. Explainability is important to programmers who should understand the reason behind the inference and either mark them as false positives or address them appropriately.
- Improve adversarial robustness for learned models. We should take effective countermeasures to improve the robustness of representation learning models, with a particular focus on adversarial training in safety-critical static analysis tools.
- Integrate models into real-world production environment and iteratively improve them. The existing works present little enlightenment how existing models can be integrated into an integrated development environment.
- Develop innovative pre-trained models which can deal with:
  - Multi-source heterogeneous data
  - Self or weak supervision
  - Multi-modality

  Current pre-training models are trained on homogeneous data, such as specific code patterns, which do not generalize well to heterogeneous data used in program analysis.

VII. Threats to Validity

Construct validity: We carefully refined the study methodology, including the study scope, goal, and protocol. We look into the literature to find out more about big code, but the problem is that we have scattered sources of information in academia. Thus, we only focus on the most relevant papers and include them in our review. This may bias readers' perspectives on what big code actually is. We mitigate this construct threat by reading even more papers until we reach a consensus to move on to the next step.

Internal validity: One threat to the internal validity might be that the collection and inclusion/exclusion are subjective to individual opinion since there is only one author in this paper. We mitigate the internal threat by carefully double-checking until the results are good enough to be accepted.

Conclusion validity: We still have limited knowledge of big code since such a domain evolving fast. Thus, our agreement on the categorization might not be accurate. We alleviate the conclusion threat by comparing our results with previous studies.

External validity: There are a number of sub-domains in big code, but we only select a few related works. We mitigate external threats via picking the most representative research after reading extensively on big code.

VIII. Conclusion

This paper studies the challenges, progress, and prospects in the area of big code, i.e. representation learning for source code modeling. We also want to provide some final thoughts about responsible AI for big code. Responsible AI allows companies to engender trust and scale AI with confidence. That is crucial when we deploy the big code models in the production environment. For instance, a bug is detected and then fixed by an automatic program repair model. However, no one knows why the bug gets fixed in this manner since the model explainability is poor. This is dangerous since someday the fixed system might malfunction in another application scenario due to the false-positive bug fix. Therefore, besides aiming for high performance, researchers in this area should also be aware of the trustworthiness of a big code model in real-world settings.

References

[1] D. E. Knuth, "Literate programming," in *Comput. J.*, 1984.

[2] P. T. Devanbu, "On the naturalness of software," *2012 34th International Conference on Software Engineering (ICSE)*, pp. 837–847, 2012.

[3] S. Bhatia, P. Kohli, and R. Singh, "Neuro-symbolic program corrector for introductory programming assignments," *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 60–70, 2018.

[4] U. Z. Ahmed, P. Kumar, A. Karkare, P. Kar, and S. Gulwani, "Compilation error repair: For the student programs, from the student programs," *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, pp. 78–87, 2018.

[5] X. Yu, Q. Huang, Z. Wang, Y. Feng, and D. Zhao, "Towards context-aware code comment generation," in *FINDINGS*, 2020.

[6] Y. Hussain, Z. Huang, Y. Zhou, and S. Wang, "Deep transfer learning for source code modeling," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 30, pp. 649–668, 2020.

[7] P. Jain, A. Jain, T. Zhang, P. Abbeel, J. Gonzalez, and I. Stoica, "Contrastive code representation learning," in *EMNLP*, 2021.

[8] H. Babii, A. Janes, and R. Robbes, "Modeling vocabulary for big code machine learning," *ArXiv*, vol. abs/1904.01873, 2019.

[9] Z. Chen and M. Martin, "A literature study of embeddings on source code," *ArXiv*, vol. abs/1904.03061, 2019.

[10] G. Ramakrishnan, J. Henkel, Z. Wang, A. Albarghouthi, S. Jha, and T. Reps, "Semantic robustness of models of source code," *ArXiv*, vol. abs/2002.03043, 2020.

[11] B. Li, M. Yan, X. Xia, X. Hu, G. Li, and D. Lo, "Deepcommenter: a deep code comment generation tool with hybrid lexical and syntactical information," *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020.

[12] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pp. 200–20 010, 2018.

[13] M. Bahrami, S. N. C., S. Ruangwan, L. Liu, Y. Mizobuchi, M. Fukuyori, W.-P. Chen, K. Munakata, and T. Menzies, "Pytorrent: A python library corpus for large-scale language models," *ArXiv*, vol. abs/2110.01710, 2021.

[14] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura, "Learning to generate pseudo-code from source code using statistical machine translation (t)," *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 574–584, 2015.

[15] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *ArXiv*, vol. abs/2102.04664, 2021.

[16] K. Wang and M. Christodorescu, "Coset: A benchmark for evaluating neural program embeddings," *ArXiv*, vol. abs/1905.11445, 2019.

[17] Y. Li, D. H. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, Tom, Eccles, J. Keeling, F. Gimeno, A. D. Lago, T. Hubert, P. Choy, C. de, M. d'Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Gowal, Alexey, Cherepanov, J. Molloy, D. J. Mankowitz, E. S. Robson, P. Kohli, N. de, Freitas, K. Kavukcuoglu, and O. Vinyals, "Competition-level code generation with alphacode," 2022.

[18] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," *2014 IEEE International Conference on Software Maintenance and Evolution*, pp. 476–480, 2014.

[19] G. Orlanski and A. Gittens, "Reading stackoverflow encourages cheating: Adding question text improves extractive code generation," *ArXiv*, vol. abs/2106.04447, 2021.

[20] P. Yin and G. Neubig, "Tranx: A transition-based neural abstract syntax parser for semantic parsing and code generation," in *EMNLP*, 2018.

[21] V. Zhong, C. Xiong, and R. Socher, "Seq2sql: Generating structured queries from natural language using reinforcement learning," *ArXiv*, vol. abs/1709.00103, 2017.

[22] W. Ling, P. Blunsom, E. Grefenstette, K. M. Hermann, T. Kociský, F. Wang, and A. W. Senior, "Latent predictor networks for code generation," *ArXiv*, vol. abs/1603.06744, 2016.

[23] F. F. Xu, Z. Jiang, P. Yin, B. Vasilescu, and G. Neubig, "Incorporating external knowledge through pre-training for natural language to code generation," in *ACL*, 2020.

[24] P. Yin, B. Deng, E. Chen, B. Vasilescu, and G. Neubig, "Learning to mine aligned code and natural language pairs from stack overflow," in *International Conference on Mining Software Repositories*, ser. MSR. ACM, 2018, pp. 476–486.

[25] A. M. Mir, E. Latoskinas, and G. Gousios, "Manytypes4py: A benchmark python dataset for machine learning-based type inference," *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pp. 585–589, 2021.

[26] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. X. Song, and J. Steinhardt, "Measuring coding challenge competence with apps," *ArXiv*, vol. abs/2105.09938, 2021.

[27] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *ArXiv*, vol. abs/1909.03496, 2019.

[28] X. Liu, D. Wang, A. Y. Wang, and L. Wu, "Haconvgnn: Hierarchical attention based convolutional graph neural network for code documentation generation in jupyter notebooks," in *EMNLP*, 2021.

[29] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[30] J. Gu, P. Salza, and H. C. Gall, "Assemble foundation models for automatic code summarization," 2022.

[31] A. V. M. Barone and R. Sennrich, "A parallel corpus of python functions and documentation strings for automated code documentation and code generation," in *IJCNLP*, 2017.

[32] F. Liu, G. Li, Y. Zhao, and Z. Jin, "Multi-task learning based pre-trained language model for code completion," *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 473–485, 2020.

[33] Y. Hussain, Z. Huang, Y. Zhou, and S. Wang, "Deepvs: An efficient and generic approach for source code modeling usage," *ArXiv*, vol. abs/1910.06500, 2020.

[34] N. D. Q. Bui, Y. Yu, and L. Jiang, "Infercode: Self-supervised learning of code representations by predicting subtrees," *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 1186–1197, 2021.

[35] S. Suneja, Y. Zheng, Y. Zhuang, J. Laredo, and A. Morari, "Learning to map source code to software vulnerability using code-as-a-graph," *ArXiv*, vol. abs/2006.08614, 2020.

[36] D. Tarlow, S. Moitra, A. C. Rice, Z. Chen, P.-A. Manzagol, C. Sutton, and E. Aftandilian, "Learning to fix build errors with graph2diff neural networks," *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 2020.

[37] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pre-training code representations with data flow," *ArXiv*, vol. abs/2009.08366, 2021.

[38] M. Yasunaga and P. Liang, "Graph-based, self-supervised program repair from diagnostic feedback," *ArXiv*, vol. abs/2005.10636, 2020.

[39] M. Allamanis and M. Reserch, "Graph neural networks in program analysis," *Graph Neural Networks: Foundations, Frontiers, and Applications*, 2022.

[40] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *ArXiv*, vol. abs/1711.00740, 2018.

[41] M. Brockschmidt, M. Allamanis, A. L. Gaunt, and O. Polozov, "Generative code modeling with graphs," *ArXiv*, vol. abs/1805.08490, 2019.

[42] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Learning and evaluating contextual embedding of source code," in *ICML*, 2020.

[43] T. D. Dowdell and H. Zhang, "Language modelling for source code with transformer-xl," *ArXiv*, vol. abs/2007.15813, 2020.

[44] P. Yin, G. Neubig, M. Allamanis, M. Brockschmidt, and A. L. Gaunt, "Learning to represent edits," *ArXiv*, vol. abs/1810.13337, 2019.

[45] A. Stehnii, "Generation of code from text description with syntactic parsing and tree2tree model," 2018.

[46] J. Devlin, J. Uesato, R. Singh, and P. Kohli, "Semantic code repair using neuro-symbolic transformation networks," *ArXiv*, vol. abs/1710.11054, 2017.

[47] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, and H. Leather, "Programl: Graph-based deep learning for program optimization and analysis," *ArXiv*, vol. abs/2003.10536, 2020.

[48] U. Kusupati, V. Ravi, and T. Ailavarapu, "Natural language to code using transformers," 2018.

[49] S. Chakraborty, Y. Ding, M. Allamanis, and B. Ray, "Codit: Code editing with tree-based neural models," *IEEE Transactions on Software Engineering*, pp. 1–1, 2020.

[50] Z. Sun, Q. Zhu, Y. Xiong, Y. Sun, L. Mou, and L. Zhang, "Treegen: A tree-based transformer architecture for code generation," in *AAAI*, 2020.

[51] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," *ArXiv*, vol. abs/1602.03001, 2016.

[52] Y. Hussain, Z. Huang, S. Wang, and Y. Zhou, "Codegru: Context-aware deep learning with gated recurrent unit for source code modeling," *ArXiv*, vol. abs/1903.00884, 2020.

[53] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *ACL*, 2016.

[54] N. D. Q. Bui, Y. Yu, and L. Jiang, "Treecaps: Tree-based capsule networks for source code processing," in *AAAI*, 2021.

[55] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *ArXiv*, vol. abs/1808.01400, 2019.

[56] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, pp. 1 – 29, 2019.

[57] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," *ArXiv*, vol. abs/2002.08155, 2020.

[58] I. Abdelaziz, J. Dolby, J. P. McCusker, and K. Srinivas, "Can machines read coding manuals yet? - a benchmark for building better language models for code understanding," *ArXiv*, vol. abs/2109.07452, 2021.

[59] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," *ArXiv*, vol. abs/1508.07909, 2016.

[60] S. Proksch, J. Lerch, and M. Mezini, "Intelligent code completion with bayesian networks," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, pp. 1 – 31, 2015.

[61] R.-M. Karampatsis, H. Babii, R. Robbes, C. Sutton, and A. Janes, "Big code != big vocabulary: Open-vocabulary models for source code," *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pp. 1073–1085, 2020.

[62] J. Li, Y. Wang, M. R. Lyu, and I. King, "Code completion with neural attention and pointer networks," *ArXiv*, vol. abs/1711.09573, 2018.

[63] R. Tiwang, T. Oladunni, and W. Xu, "A deep learning model for source code generation," *2019 SoutheastCon*, pp. 1–7, 2019.

[64] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015.

[65] M. Allamanis, E. T. Barr, and C. Sutton, "Learning natural coding conventions," *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014.

[66] V. J. Hellendoorn and P. T. Devanbu, "Are deep neural networks the best choice for modeling source code?" *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017.

[67] J. Cruz-Benito, S. Vishwakarma, F. Martín-Fernández, I. F. I. Quantum, Electrical, and C. E. C. M. University, "Automated source code generation and auto-completion using deep learning: Comparing and discussing current language-model-related approaches," *ArXiv*, vol. abs/2009.07740, 2020.

[68] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," *ArXiv*, vol. abs/2103.06333, 2021.

[69] A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, pp. 858–868, 2015.

[70] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "A general path-based representation for predicting program properties," *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2018.

[71] P. Bielik, V. Raychev, and M. T. Vechev, "Phog: Probabilistic model for code," in *ICML*, 2016.

[72] J. Cruz-Benito, I. Faro, F. Martín-Fernández, R. Therón, and F. J. García-Peñalvo, "A deep-learning-based proposal to aid users in quantum computing programming," in *HCI*, 2018.

[73] Y. Bengio, A. C. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, pp. 1798–1828, 2013.

[74] D. Kozlowski, J. Dusdal, J. Pang, and A. Zilian, "Semantic and relational spaces in science of science: Deep learning models for article vectorisation," *Scientometrics*, vol. 126, pp. 5881–5910, 2021.

[75] V. Kumar, "Algorithms for constraint-satisfaction problems: A survey," *AI Mag.*, vol. 13, pp. 32–44, 1992.

[76] T. Cohn, P. Blunsom, and S. Goldwater, "Inducing tree-substitution grammars," *J. Mach. Learn. Res.*, vol. 11, pp. 3053–3096, 2010.

[77] Z. Chi, "Statistical properties of probabilistic context-free grammars," *Comput. Linguistics*, vol. 25, pp. 131–160, 1999.

[78] A. P. Wright and H. Wiklicky, "Comparison of syntactic and semantic representations of programs in neural embeddings," *ArXiv*, vol. abs/2001.09201, 2020.

[79] K. Wang, "An evalutation of programming language models' performance on software defect detection," *ArXiv*, vol. abs/1909.10309, 2019.

[80] Y. Wainakh, M. Rauf, and M. Pradel, "Evaluating semantic representations of source code," *ArXiv*, vol. abs/1910.05177, 2019.

[81] T. H. M. Le, H. Chen, and M. A. Babar, "Deep learning for source code modeling and generation," *ACM Computing Surveys (CSUR)*, vol. 53, pp. 1 – 38, 2020.

[82] S. Gupta and A. Stent, "Automatic evaluation of referring expression generation using corpora," 2005.

[83] R. Likert, "A technique for the measurement of attitude scales," 1932.

[84] S. Banerjee and A. Lavie, "Meteor: An automatic metric for mt evaluation with improved correlation with human judgments," in *IEEvaluation@ACL*, 2005.

[85] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *ACL*, 2002.

[86] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "Squad: 100,000+ questions for machine comprehension of text," in *EMNLP*, 2016.

[87] D. R. Radev, H. Qi, H. Wu, and W. Fan, "Evaluating web-based question answering systems," in *LREC*, 2002.

[88] J.-L. Gauvain, Y. de Kercadio, L. Lamel, and G. Adda, "The limsi sdr system for trec-8," in *TREC*, 1999.

[89] K. Kishida, "Property of average precision and its generalization: An examination of evaluation indicator for information retrieval experiments," 2005.

[90] R. Caruana, "Multitask learning," in *Encyclopedia of Machine Learning and Data Mining*, 1998.

[91] M. K. Sarker, L. Zhou, A. Eberhart, and P. Hitzler, "Neuro-symbolic artificial intelligence," *AI Commun.*, vol. 34, pp. 197–209, 2021.

[92] Y. Bengio, P. Y. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE transactions on neural networks*, vol. 5 2, pp. 157–66, 1994.