

# The Good, the Bad, and the Missing: Neural Code Generation for Machine Learning Tasks

JIHO SHIN, York University, Canada

MOSHI WEI, York University, Canada

JUNJIE WANG, Institute of Software, Chinese Academy of Sciences, China

LIN SHI, Institute of Software, Chinese Academy of Sciences, China

SONG WANG, York University, Canada

Machine learning (ML) has been increasingly used in a variety of domains, while solving ML programming tasks poses unique challenges because of the fundamentally different nature and construction from general programming tasks, especially for developers who do not have ML backgrounds. Automatic code generation that produces a code snippet from a natural language description can be a promising technique to accelerate ML programming tasks. In recent years, although many deep learning-based neural code generation models have been proposed with high accuracy, the fact that most of them are mainly evaluated on general programming tasks calls into question their effectiveness and usefulness in ML programming tasks. In this paper, we set out to investigate the effectiveness of existing neural code generation models on ML programming tasks. For our analysis, we select six state-of-the-art neural code generation models, and evaluate their performance on four widely used ML libraries, with newly-created 83K pairs of natural-language described ML programming tasks. Our empirical study reveals some good, bad, and missing aspects of neural code generation models on ML tasks, with a few major ones listed below. **(Good)** Neural code generation models perform significantly better on ML tasks than on non-ML tasks. **(Bad)** Most of the generated code is semantically incorrect. **(Bad)** Code generation models cannot significantly improve developers' completion time. **(Good)** The generated code can help developers write more correct code by providing developers with clues for using correct APIs. **(Missing)** The observation from our user study reveals the missing aspects of code generation for ML tasks, e.g., decomposing code generation for divide-and-conquer into two tasks: API sequence identification and API usage generation.

CCS Concepts: • **Software and its engineering** → **Open source model**; • **General and reference** → **Empirical studies**.

Additional Key Words and Phrases: Neural code generation, machine learning tasks, empirical analysis

## ACM Reference Format:

Jiho Shin, Moshi Wei, Junjie Wang, Lin Shi, and Song Wang. 2022. The Good, the Bad, and the Missing: Neural Code Generation for Machine Learning Tasks. 1, 1 (May 2022), 20 pages. <https://doi.org/XXXXXXX.XXXXXXX>

---

Authors' addresses: Jiho Shin, jihoshin@yorku.ca, York University, 4700 Keele St., North York, Ontario, Canada, M3J 1P3; Moshi Wei, York University, 4700 Keele St., North York, Canada, moshiwei@yorku.ca; Junjie Wang, Institute of Software, Chinese Academy of Sciences, Beijing, China, junjie@iscas.ac.cn; Lin Shi, Institute of Software, Chinese Academy of Sciences, Beijing, China, shilin@iscas.ac.cn; Song Wang, York University, 4700 Keele St., North York, Canada, wangsong@yorku.ca.

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

© 2022 Association for Computing Machinery.

Manuscript submitted to ACM

Manuscript submitted to ACM

1

## 1 INTRODUCTION

Recently, with the advances in deep learning techniques [15, 49], many neural code generation models have been proposed and extensively studied [16, 20, 38, 47, 56]. Most of these models treat code generation as a machine translation task and are built in an end-to-end manner, i.e., a corpus of source code and comment pairs are used to train the models, and given a natural language (NL) described programming task, the models will generate a source code snippet.

Although many neural code generation models have exerted high accuracy in generating source code from NL described programming tasks, most of which are mainly evaluated on general programming tasks, little is known about their effectiveness and usefulness for domain-specific tasks. In this study, we investigate Machine Learning (ML), which has significantly different paradigms from traditional general programming tasks (relatively more deterministic and less statistically orientated) [58]. In addition, ML programming tasks often require many complex algorithms, mathematical operations, and data process operations [37, 50].

To investigate the effectiveness of existing neural code generation models on ML programming tasks, We use six state-of-the-art neural code generation models as the baselines (details are in Section 3.2) and we select four widely used ML libraries, i.e., Scikit-Learn<sup>1</sup>, Keras<sup>2</sup>, TensorFlow<sup>3</sup>, and PyTorch<sup>4</sup> for ML programming task collection. In addition, inspired by existing studies [50], we categorize the ML programming tasks according to the utilized APIs into three different categories based on their purposes in the ML pipeline, i.e., *data process*, *model building*, and *evaluation* (details are in Section 2.2).

For our analysis, we need a new dataset of ML programming tasks inclusively as most of the existing widely used code generation benchmark datasets, e.g., StaQC [53], CoNaLa [54], Django [39], and ReCa [31] mainly contain general-purpose programming tasks. Specifically, we create the new ML task-related dataset by reusing data from JuCe [1], i.e., a refined human-curated code generation dataset including 1.5 million examples mined from over 659K publicly available Jupyter notebooks from GitHub. We use the API information from the four studied ML libraries to extract ML-related programming tasks. Note that, we also have collected a non-ML task-related dataset from JuCe [1] as a comparison dataset. We then evaluate the six selected state-of-the-art neural code generation models on the two datasets (ML vs non-ML) regarding the accuracy, syntactic and semantic correctness, and usefulness of the generated code. Our empirical study reveals some good, bad, and missing aspects of state-of-the-art neural code generation models on ML tasks, with a few major ones listed below. **(Good)** Neural code generation models perform significantly better on ML tasks than on non-ML tasks. **(Good)** The generated code can help developers write more correct code by providing developers with clues for using correct APIs. **(Bad)** Most of the generated code is semantically incorrect. **(Bad)** Code generation models cannot significantly improve developers' completion time. **(Missing)** The observation from our user study reveals the missing aspects of code generation for ML tasks, e.g., decomposing code generation for divide-and-conquer into two tasks: API sequence identification and API usage generation.

This paper makes the following contributions:

- The first empirical study for evaluating six state-of-the-art neural code generation models on ML programming tasks regarding the accuracy, syntactic and semantic correctness, and usefulness of the generated code.
- The missing viewpoints between existing automatic neural code generation models and practical ML programming tasks, as well as future research directions for improving code generation.

<sup>1</sup><https://scikit-learn.org/stable/>

<sup>2</sup><https://keras.io/>

<sup>3</sup><https://www.tensorflow.org/>

<sup>4</sup><https://pytorch.org/>

- A new dataset that is composed of 83K pairs of NL descriptions and the corresponding source code that are implemented by four ML libraries, which facilitate the follow-up studies in this direction.
- The released source code of our tool and the dataset of our experiments to help other researchers replicate and extend our study<sup>5</sup>.

The rest of this paper is organized as follows. Section 2 presents the background of this study. Section 3 describes the methodology and the study design of our work. Section 4 presents the evaluation results. Section 5 discusses open questions and the threats to the validity of this work. Section 6 presents the related studies. Section 7 concludes this paper.

## 2 BACKGROUND

This section introduces the background of this study, i.e., the neural source code generation model and ML programming tasks.

### 2.1 Neural Code Generation

Neural code generation techniques exploit deep neural generation models to learn the distribution or the density estimation of source code with different representations such as the sequence of tokens or tree structures such as abstract syntax tree (AST) [14, 33]. Due to the similarity of the data representation, the source code generation model inherit many of the techniques from NL generation models. In NL generation, many tasks can be categorized such as content determination, text structuring sentence aggregation, lexicalization, referring expression generation, linguistic realization, and so on [18]. There are also different tasks in the source code generation such as code completion [30], code search [19], program synthesis via rules or examples [13], domain-specific [17] or general-purpose code generation [8], code to code translation [3], API recommendation [11], and so on. The generation model that this paper focuses on is the multi(bi) modal modeling of source code and NL, which usually uses an end-to-end encoder-decoder architecture of the model that is widely used in neural machine translations. The encoder takes in the sequence of an NL description of a code and maps them in a latent space and then the decoder decodes them into a source code. The encoder-decoder layers learn to map similar data points of each modal to be closely mapped in the latent space so that it generates similar source codes.

To evaluate code generation models, many different benchmark datasets have been proposed. Django [39] dataset was first mined to generate pseudo-code from source code for the Django framework (Python-to-English). The data contains 18K pairs of python statements and their corresponding English pseudo-code. CoNaLa [54] is a dataset mined from Stack Overflow posts which consists of 3K pairs of human-annotated python code snippets and its NL intent. StaQC [53] is another source code and NL description pair that are mined from Stack Overflow. It consists of 1.4K Python and 1.2K SQL queries which were mined using a neural network to incorporate textual similarities. ReCa [31] is a large-scale dataset of NL requirements and its programming language. ReCa consists of data for multiple general languages, i.e., C, C++, Java, and Python. It also has multiple programs for the same requirement. JuIce [1] is a new large-scale open-domain dataset composed of 1.5M pairs of Python code examples and the corresponding NL descriptions collected from 659K publicly available Jupyter notebooks on GitHub. Table 1 shows more detailed statistics of these existing datasets.

---

<sup>5</sup><https://zenodo.org/record/7036255>

Table 1. Statistics of existing benchmark datasets. **PL** denotes the program languages, **Loc** denotes the average lines of code, and **Length** denotes the average length of NL tasks in token.

Dataset	Domain	PL	#Pairs	Loc	Length
Django	general	Python	18.8K	1	14
CoNaLa	general	Python	2.9K	1.1	14
StaQC	general	Python/SQL	2.6k	10.1	10
ReCa	general	C/Java/Python	101K	37.2	185
JuIce	general	Python	1.5M	10.0	39.7

## 2.2 Machine Learning Tasks

Machine learning-based tasks often contain different steps, which are also known as the pipeline of machine learning [4].

Specifically, constructing an ML pipeline contains the following three steps. The first step is mainly about the processing of the dataset that is needed to train a model. It involves pre-processing tasks such as data loading, noise filtering, format converting, data imputation, scaling or normalization, and feature engineering (e.g., feature selection, elimination, word embedding) [26]. After processing the data, one needs to further specify the learning algorithm to infer relations that capture the hidden patterns in the dataset. In the specified algorithm, there is information regarding the parameters that are learned and fit the input data, the loss function to be minimized, the optimizer and the learning rate to lead the model in better learning, and the architecture of the model that specifies the construction of the network’s layers and nodes [27]. And finally, one needs to evaluate the trained model. This step includes choosing the evaluation metrics to assess the performance of trained models, e.g., accuracy, precision, and recall [22]. Evaluations are applied to an unseen dataset that is separate from the training data to assess the generalizability of the trained model. In this work, to understand the effectiveness of existing neural code generation models on different types of ML programming tasks, we categorize ML programming tasks into different steps in the ML pipeline [50].

- **Data processing:** tasks that are related to data processing, e.g., data loading, format transformation, normalization, feature engineering, and data transformation.
- **Model building:** tasks that are related to building the machine/deep learning models, which can be conventional ML models such as classification models, regression models, clustering models, or neural network models with specific neural architecture regarding the type of layers and activation function used.
- **Evaluation:** tasks that are related to evaluating the performance of the trained machine/deep learning models, e.g., getting the values of different evaluation metrics such as accuracy, f-measure, AUC, or getting the probability of instances from the model.

Note that while we focus on three types of ML programming tasks in the ML pipeline (i.e., *data process*, *model building*, and *evaluation*), there are other tasks in the ML pipeline, e.g., model deployment and performance monitoring [4]. As finishing these tasks mainly requires auxiliary functions or scripts rather than the APIs from the ML libraries, thus we exclude these tasks in this study.

## 3 STUDY DESIGN

In this section, we discuss the design of our empirical study on evaluating the performance of code generation models on ML programming tasks.

Table 2. The studied ML libraries in this work.

Library	Description	#API
Scikit-learn	A library for ML algorithms.	1.2k
Keras	An interface for artificial neural networks.	1.4k
TensorFlow	A library for DL developed by Google.	8.0k
PyTorch	A library for ML and DL tasks.	0.4k

Table 3. Statistics of the experimental datasets. #API is the average number of APIs involved.

ML task	#Training	#Testing	Loc	Length	#API
data	30.9K	5.5K	3.93	12.67	1.48
model	26.3K	5.5K	4.18	12.44	1.28
evaluation	13K	2.6K	3.28	11.59	1.13
non-ml	34K	5.5K	3.23	13.26	1.63

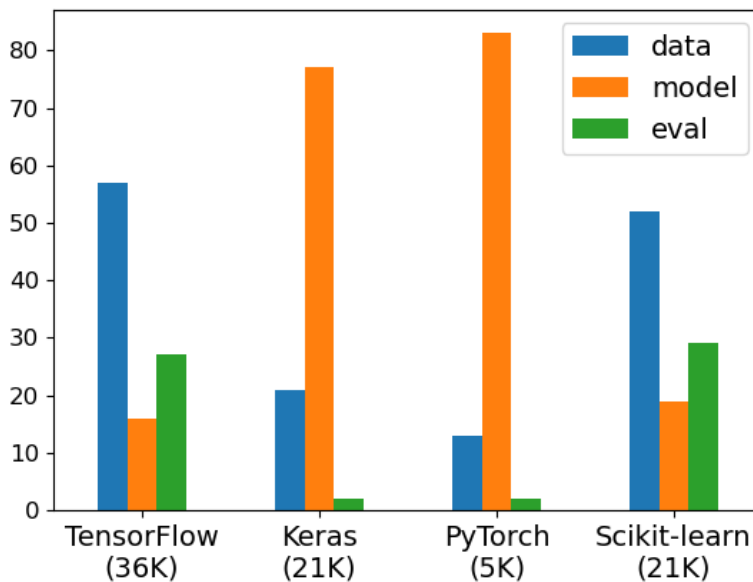


Fig. 1. Distribution of experiment data in each ML library.

### 3.1 Data Collection

In this work, we prepare a new dataset that contains the three types of ML programming tasks (i.e., *data process*, *model building*, and *evaluation*) as most of the existing widely used code generation benchmark datasets, e.g., StaQC [53], CoNaLa [54], Django [39], and ReCa [31] mainly contains general-purpose programming tasks. Specifically, we create our dataset from JuICe [1], i.e., a refined human-curated code generation dataset including 1.5 million Python code examples mined from over 659K publicly available Jupyter notebooks from GitHub.

**ML API selection:** The original JuCe dataset [1] contains open-domain programming tasks. For our study, we focus on ML programming tasks that are developed by four widely used ML libraries, i.e., Scikit-learn, Keras, TensorFlow, and PyTorch. The list of studied libraries is organized in Table 2. Specifically, three of the authors independently use the following two steps to manually identify all the APIs in each of the four ML libraries that are related to the three types of ML programming tasks (i.e., *data process*, *model building*, and *evaluation*) based on the documentation of these APIs: 1). We read the documentation of the API to understand its functionality. A category will be assigned to the API based on its major intent. 2). If a decision cannot be made based on the documentation of the API as the description might not contain enough information, we further check the source code following the instructions in Step 1 to label the API. The value of Fleiss’ kappa during this process is 0.90, which shows an almost perfect agreement. In total, 128, 244, and 51 APIs are identified for the three types of ML programming tasks, i.e., *data process*, *model building*, and *evaluation* respectively.

**ML-task data construction:** After we identify the APIs in an ML library for each of the three ML programming tasks, we further use the corresponding APIs as the filter to collect ML programming tasks from the JuCe dataset [1], e.g., we scan the source code of each programming task in JuCe, a programming task will be labeled as *data process* task if it only contains *data process* related APIs from each of the four ML libraries. We discard any instances that use multiple ML libraries to distinguish programming tasks that use different libraries.

**Data cleaning:** We also remove duplicated data instances, meaningless source code elements, i.e., ‘>>>’, single and multi-line comments, lines that start with ‘!’ and ‘%’ for commands, code snippets with just imports, and none ASCII code. We also apply additional filtering to further denoise the dataset and make the problem simpler by discarding instances with NL description lengths below 20 and over 150, and source code with more than 10 LOC as suggested by existing work [25]. To assess the quality of the collected data, the first three authors performed a manual analysis on 100 random samples from each split, i.e. *data processing*, *evaluation*, and *model building*. The authors first checked if the source code of the instance is relevant to the ML programming task. Then we check if the NL description is relevant to the source code. We found that 64%-81% of the samples have both source code and NL relevant to the ML programming task and its description. We also carefully split the instances on a project level, i.e. instances from the same project go to the same split, to avoid any information leakage.

The new dataset of ML programming tasks is organized in Table 3. We also show the distribution of the different ML programming tasks in each ML library in Figure 1.

### 3.2 Studied Neural Code Generation Models

In this work, we select six state-of-the-art neural code generation models proposed in the recent three years as our experiment subjects. These models are selected mainly based on the following inclusion criteria:

- The model should not stick to a specific programming language, which makes it practical and generalizable for it to work on our dataset.
- The model’s implementation should be publicly available and replicable, which can remove potential implementation bias and facilitates the evaluation process.
- The model should be proposed in recent years and represents the state-of-the-art, i.e., has a high performance in generation in BLEU score.

As a result, six neural code generation models are selected from 29 candidate papers published on ACL, ICLR, AAAI, TSE, EMNLP, etc. Details of these models are as follows:

**tranX** [56]: proposed a transition-based neural semantic parser that maps NL utterances into formal meaning representations such as executable programs. Specifically, it develops a transition system that decomposes the generation procedure of an AST into a sequence of tree-constructing actions.

**External-Knowledge-Codegen (EK Codegen)** [52]: proposed a model-agnostic approach based on data augmentation, retrieval, and data re-sampling, to incorporate external knowledge into code generation models. Specifically, the mined NL-code pairs from the online programming QA forum Stack Overflow and API documentation for pre-training. The basic architecture of this model uses **tranX** as its base.

**CG-RL** [24]: proposed an extended Seq2Tree model equipped with a context-based branch selector, which is capable of dynamically determining optimal branch expansion orders for multi-branch nodes. Specifically, it adopts reinforcement learning to train the whole model with an elaborate reward that measures the loss difference between different branch expansion orders.

**Codegen-TAE** [38]: exploited the use of a large amount of monolingual programming language corpus to fit a generic transformer-based Seq2Seq model with minimal code-generation-specific inductive bias.

**TreeCodeGen** [14]: proposed a Transformer-based structure-aware tree-to-tree model. They adopted a Tree Transformer, an attention-based tree-to-tree model with a hierarchical accumulation for code generation.

**PyCodeGPT** [57]: proposed a large pre-trained language model based on the GPT-Neo [6] with 110M parameters, which is comparable to the powerful GPT-3 model [7]. They mined 1.2M Python repositories in GitHub and filtered/pre-processed high-quality Python code resulting in 96GB of training data.

In our experiment, to remove potential bias, we use the implementation of their publicly available code with the settings that have the best performance reported in their papers or used in their replication packages.

### 3.3 Research Questions

We answer the following research questions to evaluate the performance of the studied neural code generation models on ML programming tasks.

- **RQ1 (Accuracy):** How accurate are the code generation models on different ML programming tasks?
- **RQ2 (Syntactic Correctness):** How often does the generated code for ML programming tasks pass syntactic checking?
- **RQ3 (Semantic Correctness):** How often is the generated code for ML programming tasks pass semantic checking?
- **RQ4 (Usefulness):** Is the generated code for ML programming tasks useful for developers?

In **RQ1**, we set out to investigate the performance of state-of-the-art neural code generation models on the three different types of ML programming tasks. In **RQ2** and **RQ3**, we assess the soundness of the generated code by checking its syntactic and semantic correctness, respectively. In **RQ4**, we explore the usefulness of the generated code to developers in practice via a user case study.

### 3.4 Experiment Setup

The selected six neural code generation models are required to apply specific data pre-processing steps, i.e., building AST from the source code used in **tranX** [56]. Thus to rebuild these models, we first apply the data pre-processing scripts provided in their replication packages. For training each model, we select 80% of our training data to train the model and the left 20% are used as validation data.

To maximize the potential of the evaluated approaches, we perform hyper-parameter tuning for each of the evaluated approaches. For **tranX**, we use 0.3 for the dropout, a hidden size of 256 and an embedding size of 128, a learning rate of 0.001, batch size of 64. We train the model for a maximum of 20 epochs and use the beam size of 15 when testing. For **EK-codegen**, we use the same value of hyper-parameter used in **tranX**. For **CG-RL**, we used the same dropout, network size, and learning rate for the training. However, we use 10 epochs to pre-train the branch selector following the original paper of **CG-RL** [24]. For **Code-gen-TAE**, we use 1e-05 for encoder learning rate and 7.5e-05 for decoder learning rate, the hidden size of the encoder is 768 with 12 layers, the decoder size is 256 with 4 layers, and the batch size used is 16. We also trained them for 20 epochs like the rest of the model and used a beam number of 10 when testing. For **TreeCodeGen**, 0.1 is used for the dropout, and the hidden and embedding size used is the same as **tranX**. The learning rate used is 1e-4. The max epoch used for training is 20 and the beam size used for testing is 30. For **PyCodeGPT**, the model is trained with 32K vocabularies, 768 for hidden size, 12 for the number of heads and layers, a window size of 256, and an initializer range of 0.02, which is the same settings reported from GPT-Neo and PyCodeGPT. The model was trained for 100K steps as suggested in the original paper.

All the aforementioned settings used for the training are from the papers of the six models and any details missing from the paper are set as the default settings provided by their replication packages. We used the GPUs in the Google Colab Pro<sup>6</sup>, which dedicates either NVIDIA Tesla P100 or T4, for training models. For training PyCodeGPT, we used NVIDIA A100 40GB from google computing platform<sup>7</sup>. The time cost of model training of these models ranges from 3 hours (i.e., **tranX**) to 16 hours (i.e., **PyCodeGPT**).

### 3.5 Evaluation Metrics

To assess the performance of the models in generating high-quality source code, we use the following three evaluation measures, i.e., BLEU [40], METEOR [5], and ROUGE [28].

BLEU score [40] is an evaluation metric commonly used in assessing the generation models in natural language processing. It measures the similarity of the model-generated text to the ground truth text by computing the overlapping n-grams of the two texts. A value of 0 indicates that the generated source code has no overlap of n-grams with the ground truth source code, while the value of 1 indicates that it is a perfect overlap with the ground truth source code. There are different ways to calculate the BLEU score. We can set a different number of n-grams we want to assess in overlapping, i.e. unigram, bigram, trigram, and 4-gram. This is also referred as to BLEU-1, BLEU-2, BLEU-3, and BLEU-4 respectively. METEOR metric [5] is also commonly used to assess machine language generation models. The metric is based on the harmonic mean of unigram precision and recall but with a higher weight on recall. ROUGE [28] is a metric used for evaluating natural language generation models to compare the generated sequence to the reference. There are also multiple sets of assessing different ROUGE values. In this study, we report the ROUGE-L value of each model which uses the Longest Common Sub-sequence (LCS) in assessing the similarity of the two texts.

## 4 RESULTS AND ANALYSIS

This section presents the experimental results and answers the research questions discussed in Section 3.3.

<sup>6</sup><https://colab.research.google.com/>

<sup>7</sup><https://cloud.google.com/>



Table 4. Performance of the studied six neural code generation models on different ML tasks and non-ML tasks. The best performance of a model on different tasks is shown in bold, while the overall best model on each ML task is shown with asterisk (\*).

Models	Tasks	BLEU-1	BLEU-2	BLEU-3	BLEU-4	ROUGE-L	METEOR
tranX	<i>data process</i>	0.3606	0.2670*	0.2237*	0.1905	0.3380	0.3600*
	<i>model building</i>	<b>0.3646</b>	<b>0.2717</b>	<b>0.2277</b>	<b>0.1940</b>	0.3167	0.3598
	<i>evaluation</i>	0.3621	0.2677	0.2233	0.1886	<b>0.3532</b>	<b>0.3712</b>
	<i>non-ml</i>	0.2855	0.1714	0.1236	0.0929	0.1636	0.2282
EK Codegen	<i>data process</i>	0.3391	0.2541	0.2154	0.1853	0.3441	0.3592
	<i>model building</i>	0.3549	0.2690	<b>0.2289</b>	<b>0.1978</b>	0.3241	0.3633
	<i>evaluation</i>	<b>0.3593</b>	<b>0.2698*</b>	0.2282	0.1953	<b>0.3616</b>	<b>0.3759*</b>
	<i>non-ml</i>	0.2969*	0.1838*	0.1348*	0.1028*	0.1705	0.2365*
CG-RL	<i>data process</i>	<b>0.2503</b>	<b>0.1943</b>	<b>0.1678</b>	<b>0.1471</b>	0.3397	0.3333
	<i>model building</i>	0.1923	0.1471	0.1257	0.1089	0.3160	0.3057
	<i>evaluation</i>	0.2404	0.1840	0.1575	0.1366	<b>0.3620</b>	<b>0.3480</b>
	<i>non-ml</i>	0.1213	0.0776	0.0578	0.0450	0.1723	0.1916
TreeCodeGen	<i>data process</i>	0.3465	0.2540	0.2125	0.1810	<b>0.3530*</b>	0.3476
	<i>model building</i>	0.3483	0.2559	0.2134	0.1812	0.3122	0.3377
	<i>evaluation</i>	<b>0.3644*</b>	<b>0.2681</b>	<b>0.2235</b>	<b>0.1883</b>	<b>0.3530</b>	<b>0.3619</b>
	<i>non-ml</i>	0.1997	0.1255	0.0932	0.0723	0.1929*	0.1979
Codegen-TAE	<i>data process</i>	<b>0.3775*</b>	<b>0.2444</b>	<b>0.1857</b>	<b>0.1441</b>	<b>0.3145</b>	<b>0.3233</b>
	<i>model building</i>	0.3279	0.1993	0.1438	0.1052	0.2339	0.2623
	<i>evaluation</i>	0.2919	0.1743	0.1260	0.0914	0.2357	0.2358
	<i>non-ml</i>	0.2620	0.1451	0.0985	0.0703	0.1217	0.1944
PyCodeGPT	<i>data process</i>	0.3347	0.2562	0.2219	0.1958*	0.3333	0.3462
	<i>model building</i>	<b>0.4036*</b>	<b>0.3286*</b>	<b>0.2925*</b>	<b>0.2634*</b>	<b>0.3701*</b>	<b>0.3893*</b>
	<i>evaluation</i>	0.3397	0.2693	0.2361*	0.2094*	0.3667*	0.3654
	<i>non-ml</i>	0.0789	0.0352	0.0217	0.0141	0.0619	0.0938

#### 4.1 RQ1: Accuracy

**Approach:** To answer this RQ, we follow our experiment setups (described in Section 3.4) to re-train each of the six neural code generation models with data from different ML programming tasks, i.e., *data process*, *model building*, and *evaluation* datasets. We also evaluate the performance of the six neural code generation models on the non-ML programming task dataset for comparison. For creating the non-ML programming task dataset, we randomly select samples from JuIce [1] with ML programming tasks removed. To make a fair comparison, we set the size of the non-ML task dataset to our biggest ML task dataset, i.e. *data process* dataset. To remove potential bias, we repeat the data creation for non-ML tasks 10 times and re-train the models accordingly. We use the average performance of each studied neural code generation model for comparison. To assess the model’s capability of generating source code on the different ML libraries, we label each testing instance with the studied ML library using the ML API name.

**Result:** The performance of the studied six neural code generation models on different types of ML tasks and non-ML tasks are organized in Table 4. Overall, **PyCodeGPT** performs best, especially on ML-related tasks. For example, **PyCodeGPT** achieves the largest values regarding all the metrics on *model building*. However, on non-ML tasks, **EK Codegen** performs best with 5/6 of the metrics. A possible reason for these two models to have better performance is that they exploit external data for pre-training. Specifically, **PyCodeGPT** uses corpus mined from GitHub, and **EK**

**Codegen** uses corpus mined from Python Standard Library<sup>8</sup> and StackOverflow<sup>9</sup>. Due to the abundant number of ML projects on GitHub, **PyCodeGPT** benefits in generating ML-related code. Similarly, non-ML tasks are composed of short standalone code snippets, which are very similar to the code snippets answered in StackOverflow. This could be the reason why **EK Codegen** has benefited in generating non-ML programming tasks. Regarding the performance on a specific type of ML task, we can see that the best performance tasks for each neural code generation model vary. For example, **tranX** performed best on *model building* tasks over other tasks, while **CG-RL** and **Codegen-TAE** have better performance for *data process* tasks than the performance on other tasks regarding BLEU scores. **EK Codegen** and **TreeCodeGen** perform better on *evaluation*, while **PyCodeGPT** performs better on *model building* related ML tasks than the other tasks.

In addition, we can also see that all the six neural code generation models perform significantly better on ML tasks compared to non-ML tasks regarding all the different metrics, i.e., BLEU-1/2/3/4, ROUGE-L, and METEOR. Possible reasons for this can be as follows. First, the non-ML tasks are much more diverse than ML tasks as we limited the ML tasks to only four libraries, i.e., TensorFlow, Keras, Pytorch, and Scikit-learn. The variance of programs from ML tasks is less than the non-ML dataset. Second, the natural language descriptions of the non-ML-related tasks were also more varied and context-free since the domain does not have a constraint compared to the descriptions of ML tasks. This finding is interesting, as previous studies state that traditional programs are more deterministic and less statistically orientated [58]. However, from the generation point of view, it is the opposite. Even though the ML libraries are more dynamic and statistically orientated, programs built upon them are often not.

**Good:** Neural code generation models generate significantly better performance on ML tasks than non-ML tasks.

**Bad:** Neural code generation models perform not stable on different ML programming tasks.

## 4.2 RQ2: Syntactic Checking

**Approach:** To answer this RQ, we conduct syntactic checking on the generated programs for each instance in the testing dataset. Following existing work [31], we conduct a static syntactic checking on the generated programs with the state-of-the-practice tool Pylint<sup>10</sup>, a Python static code analysis tool that looks for programming errors. Note that, **codegen-TAE** generates all code statements in one line for a given programming task, while Python programs have strict style requirements, e.g., Python uses indentation to indicate a block of code. To avoid any style errors, we first parse the generated source code and output it with proper indentation. During this step, we do not change any content for avoiding potential bias. We then use Pylint to check the syntactic correctness of each instance in the testing dataset, we consider an instance syntactically correct if it passes the checking.

**Result:** Table 5 shows the results of the syntactic checking for programs generated by different neural code generation models in different ML tasks and non-ML tasks. We can observe that in general, all the state-of-the-art models have a high accuracy in generating syntactically correct programs. However, we observed a significant decline in generating syntactically correct code using **PyCodeGPT** compared to other models. One of the possible reasons for this is that these five models are AST-based approaches, given a programming task, they first generate the corresponding AST and then transfer them into a code snippet. On the other hand, **PyCodeGPT** uses a token-based seq2seq Transformer, giving the model the freedom to generate any token sequences, which can be easily syntactic incorrect. In addition, we

<sup>8</sup><https://docs.python.org/3/library/index.html>

<sup>9</sup><https://stackoverflow.com/>

<sup>10</sup><https://pylint.pycqa.org/>

Table 5. Syntactical correctness (in percentage) of the generated code by different neural code generation models on different types of machine learning tasks.

Methods	data	model	evaluation	non-ml
<b>tranX</b>	98.40	98.68	99.36	97.06
<b>EK codegen</b>	96.39	97.92	97.68	95.16
<b>CG-RL</b>	97.37	98.07	97.97	94.70
<b>TreeCodeGen</b>	92.24	95.46	97.00	82.85
<b>Codege-TAE</b>	92.35	93.99	90.70	93.22
<b>PyCodeGPT</b>	71.36	70.53	72.88	67.16

Table 6. Semantic correctness (in percentage) of the generated code by different neural code generation models on different types of machine learning tasks.

Methods	data	model	evaluation	non-ml
<b>tranX</b>	6%	14%	34%	0%
<b>EK codegen</b>	16%	12%	30%	10%
<b>PyCodeGPT</b>	34%	14%	30%	6%

Table 7. Correctness of identified APIs on the generated code by different neural code generation models on different types of machine learning tasks.

Methods	Metrics	data	model	evaluation	non-ml
<b>tranX</b>	Precision	0.2129	0.2740	0.4348	0.1214
	Recall	0.2215	0.2729	0.5105	0.1364
<b>EK codegen</b>	Precision	0.3458	0.2104	0.4234	0.2278
	Recall	0.3400	0.2693	0.4452	0.2078
<b>PyCodeGPT</b>	Precision	0.6133	0.4085	0.5500	0.0867
	Recall	0.5811	0.3943	0.4649	0.0783

can also see that all the six neural code generation models perform better on most of the ML tasks compared to non-ML tasks, which is consistent with our findings in RQ1.

**Good:** The neural code generation models generate highly accurate syntactically correct programs for ML tasks.  
**Good:** Our results confirm the advantage of AST-based code generation used in current neural code generation models compared to the token-based model.

### 4.3 RQ3: Semantic Checking

**Approach:** To further analyze the soundness of the generated programs by these state-of-the-art neural code generation models, we also check the semantics of the generated programs manually as most of the generated programs require specific inputs to be executed, which cannot be generated automatically. For this experiment, we examine the semantic correctness of the generated programs from the best three models, i.e., **tranX**, **EK codegen**, and **PyCodeGPT** on the TensorFlow library. Specifically, we first randomly select 50 examples from each type of ML programming task, thus in total, we have 450 programming tasks to be checked, i.e., 50 samples  $\times$  3 types of ML programming tasks  $\times$  3 models. We consider a generated program of a given programming task is semantically equivalent to the ground-truth program if

the generated program shares the same logic with the ground-truth program and can solve the given programming task. For checking the semantic correctness of each sample, all the authors work together and make the decision together.

To further illustrate the effectiveness of the generated programs for ML programming tasks, we also evaluate the correctness of a generated program regarding the APIs called in the program. Specifically, given an ML programming task, we first parse its corresponding ground truth source code and extract the APIs from these four ML libraries that are used to solve the task. Then we follow the same process to extract APIs in the generated program for this task by a neural code generation model. Note that, for the non-ML tasks, we also follow the same process to collect APIs used in the ground-truth and generated programs. For measuring the performance of neural code models regarding identifying the correct APIs, we use Precision and Recall, where Precision measures the percentage of correct identified APIs among all the used APIs in the generated programs and Recall is the percentage of correct identified APIs among the ground-truth APIs.

---

Listing 1. An example of generated code by **EK codegen** for a given ML programming task

---

```
# NL description
# using a feed_dict, same graph, but without
# hardcoding inputs when building session

# ground truth
a = tf.placeholder(dtype=tf.int32, shape=(None,))
b = tf.placeholder(dtype=tf.int32, shape=(None,))
c = tf.add(a, b)
with tf.Session() as sess:
    result = sess.run(c, feed_dict={
        a: [3, 4, 5], b: [-1, 2, 3]})
    print(result)

# generated
x = tf.placeholder(tf.float32)
with tf.Session() as sess:
    print(sess.run(x))
```

---

**Result:** Table 6 shows the semantic correctness of the generated programs of the three neural code generation models. We can observe that only a small number of samples of the generated source code are semantically correct with an average of 17% out of the 450 samples. **PyCodeGPT** outperforms **EK Codegen** followed by **tranX** with 13%, 17%, and 21% of average percentage, respectively, which is consistent with the results from RQ1. Similar to the trends revealed in RQ1 and RQ2, the examined models also have better performance on ML programming tasks than non-ML programming tasks.

Despite the low values in semantic correctness, we observe that these neural code generation models have better performance in suggesting the correct APIs for the given programming tasks compared to directly generating source code. Table 7 shows the performance of these models in identifying the correct APIs. As we can see, the precision and recall values for identifying correct APIs are higher across all different ML programming tasks, which suggests they

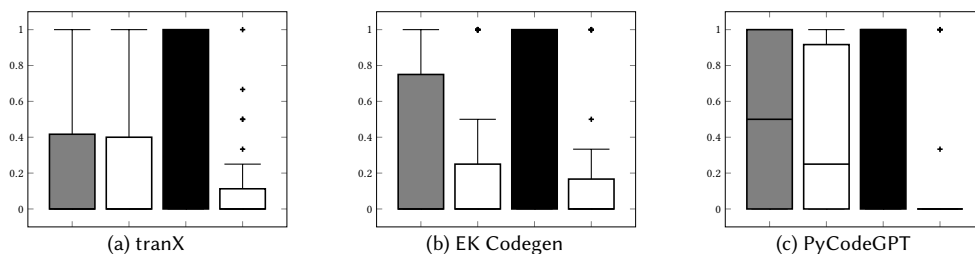


Fig. 2. Precision of identifying correct APIs of **tranX** and **EK Codegen** **PyCodeGPT**, on *data*, *model*, *eval*, *nonml*, respectively.

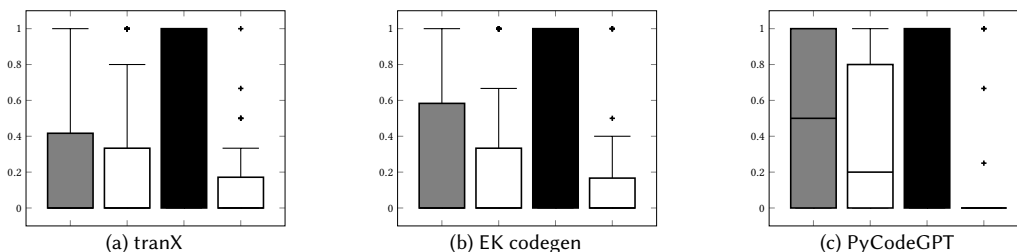


Fig. 3. Recall of identifying correct APIs of **tranX** and **EK codegen** **PyCodeGPT**, on *data*, *model*, *eval*, *nonml*, respectively.

can help identify an average of 39% of correct APIs for solving ML tasks. We further show the box plots of precision and recall values regarding identifying correct APIs of the three models in Figure 2 and Figure 3 respectively.

We further show an example ML task that cannot be solved by existing neural code generation models while some of the required APIs can be identified in Listing 1. The task is labeled as a *model building* task finished with the TensorFlow library, which is about building and running a session by loading a dictionary object. The ground truth code instantiates two placeholders with a *tf.int32* type and adds them before running a session. When they run the session, they feed a dictionary object before running them. While the generated source code failed to generate the dictionary part, it successfully generates code that instantiates a *tf.int32* typed placeholder and then builds and runs a session successfully. The precision and recall regarding identifying the correct APIs are 100% and 57% respectively.

**Bad:** All the examined neural code generation models are weak in generating semantically correct programs. **Good:** The generated programs have the potential to be used for identifying correct APIs for ML programming tasks.

#### 4.4 RQ4: Usefulness

**Approach:** To explore the practical value of neural code generation models, we further conduct a user case study to investigate whether programs generated by these neural code models can help developers finish given programming tasks efficiently and accurately. In this experiment, we examine the programs generated by the best two models, i.e., **PyCodeGPT** and **EK codegen** on the TensorFlow library. We randomly selected 20 programming tasks from the test dataset of TensorFlow. We invited 12 students (i.e., six Ph.D. students and six MS students) who are familiar with the TensorFlow framework to complete the 20 programming tasks. Their experience in developing ML tasks based on TensorFlow varied from two to four years, with an average of three years. We then divided the participants into two groups (Group1 and Group2) based on their experience. The 20 tasks were also randomly divided into two groups (Task1

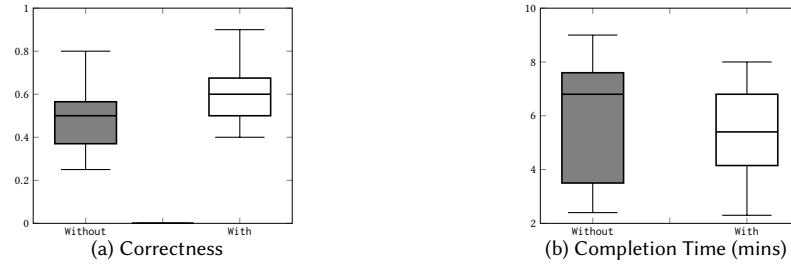


Fig. 4. Performance of participants with (denoted as ‘**With**’ in the figure) and without (denoted as ‘**Without**’ in the figure) the help of code generation models.

and Task2). The experiment was conducted in two steps. In the first step, Group1 and Group2 were asked to work on Task1 while Group1 will be given the generated program from both **PyCodeGPT** and **EK codegen**, in the second step, Group1 and Group2 were asked to work on Task2 while Group2 will be given the generated program from both **PyCodeGPT** and **EK codegen**. Each participant was asked to record his/her screen during the experiment for getting the time he/she spent on solving each programming task. Following existing studies [31, 51], we use correctness and completion time to measure the performance of the participants in solving ML tasks. Specifically, correctness evaluates whether a participant can write the correct code for a given task. Note that, as developers might solve the same task with different logic, thus we measure the proportion of correct APIs submitted by a participant among all APIs in the ground truth answer of the task. Completion time evaluates how quickly a participant can solve a given task. For each task, we recorded the correctness and completion time of each participant.

**Result:** Figure 4 shows the distribution of correctness and completion time of participants to finish the given 20 ML tasks. Overall, with the generated code from the two code generation models, participants completed the programming tasks more accurately and took slightly less time than participants without any clues. On average, the correctness and completion time of participants with and without the generated code are 0.62 and 5.2, versus 0.51 and 5.6 respectively. We further used Wilcoxon signed-rank test for verifying the statistical significance of the differences. The p-value for correctness is small than 0.05 while the p-value for completion time is larger than 0.05. The result suggests that the generated code cannot significantly reduce the time cost of ML programming tasks, while it can help developers write more correct code. One possible reason for this is that most of the generated programs are semantic incorrect, modifying the generated programs is not easy and requires a great amount of time. Our discussion with the participants confirmed that the examined code generation models can help find a part of the correct APIs for ML tasks, which provides developers with useful clues, thus further improving developers’ correctness.

**Bad:** Code generation models cannot significantly improve developers’ completion time as most of the generated programs are semantic incorrect, which requires a non-trivial time for modifications. **Good:** The generated code can help developers write more correct code by providing developers with clues for using correct APIs.

## 5 DISCUSSION

### 5.1 The Missing of Code Generation on ML Tasks

The above results and analysis have revealed several good and bad aspects of state-of-the-art neural code generation models on ML programming tasks. This section further summarizes the missing aspect that can serve as the practical guidelines for improving code generation tasks for ML tasks.

**Decompose code generation for divide-and-conquer.** Even powered with state-of-the-art deep neural techniques, code generation for ML programming tasks is still a challenge. Directly generating code from a given NL-described programming task often produces incorrect and incompetent programs that cannot be used by developers. Nevertheless, the findings of our user case study reveal that incorrectly generated code could still be useful for developers as far as it contains essential APIs to solve the tasks. Meanwhile, identifying the correct APIs is easier than generating the correct programs directly. Taken in this sense, future code generation techniques for ML tasks can be decomposed into two sequential tasks, i.e., API sequence identification and API usage generation. Crowd-sourced knowledge from Stack Overflow can be used to help generate the context of the identified API sequences, which has been proved to be useful in mining API usage scenarios [23, 48].

**Human-machine collaborated code generation.** The examined state-of-the-art neural code generation models treat code generation as a machine translation task and generate statements/blocks from a given NL-described programming task sequentially, during which any incorrect statement could mislead the generation of the sequential code snippets. Meanwhile, during our case study, similar to the findings of Murali et al. [36], we also find that, despite lacking the whole knowledge for certain programming tasks, developers usually know whether it is correct for some generated statements, or whether a specific API should be utilized based on their knowledge of an ML library. If human knowledge could be integrated into the process of automatic code generation, e.g., in the form of simple feedback from end-users to assess the correctness of the intermediate uncertain statements, the code generation could be significantly improved.

Such practice has another advantage, i.e. personalize the generated code that fit the knowledge of different end-users, by integrating developers' knowledge into the code generation process.

**Focus on domain-specific tasks.** Compared to the general non-ML programming tasks, all of the six neural code generation models exert better performance regarding syntactic correctness, semantic correctness, and identifying essential APIs, on domain-specific tasks, i.e., ML programming tasks. We assume this might be because the domain-specific task only involves a smaller set of APIs (indicating little knowledge needed to be learned), thus training an effective model is relatively easier compared to the model training on a dataset with diversified tasks. We also notice that previous practices on code generation were usually conducted on general tasks [1, 39, 53, 54]. Thus, future code generation should focus on training or fine-tuning models with data from specific domains to improve the performance of generating programs for domain-specific tasks, which have much more potential than general programming tasks. We also encourage the need for the design of domain-specific models by incorporating the characteristics of the domain, which might further improve the performance of specific tasks.

### 5.2 Threats to Validity

**Internal Validity.** The main threat to the internal validity of our study is the limited number of models that are compared. Due to this limitation, we can not generalize our findings from this paper to all the source code generation models. However, we have used the most recently published and the state-of-the-art performing ones to conduct our

experiments, we also described a detailed methodology and the setup of the study and the data used, which will allow future researchers to replicate our study and further explore other source code generation models.

**External Validity.** The main threat to external validity is the labeling of ground-truth data explored in the study. Although we have put much effort to maintain high-quality data by using real-world programs contributed by developers on GitHub, it could not be enough to validate the implementation of the source code. Also, programming or implementing a programming task can lead to many different forms and logic of source code. Thus, these programs in our dataset might not represent all the ways to implement a given programming task.

**Construct Validity.** The main threat to the construct validity can be the evaluation metrics we used. BLEU, ROUGE-L, and METEOR scores can assess program generation by calculating how the generated programs of neural code generation models are similar to the ground-truth programs of given tasks in our dataset. However as mentioned from the external validity, other implementations can be a reasonable answer rather than using similarity to the ground truth. In our future study, we plan to examine a different set of evaluation metrics to assess the generation of the program.

## 6 RELATED WORK

### 6.1 Automatic Code Generation

Besides the six neural code generation models used in this paper, there are many other methods for automatic code generation from natural language descriptions proposed in the recent decade [29, 42, 55]. More specifically, deep neural networks in the natural language processing field have been increasingly adopted to encapsulate the complexity of generating general-purpose programming languages such as Java and Python. Dong and Lapata [17] proposed *Coarse2Fine* which uses a structure-aware neural architecture that applies two decoding steps of semantic parsing. First, the approach decodes a rough sketch of the meaning representation with a high level where it omits the details such as identifiers. Then they apply another mechanism that fills the missing details by conditioning the natural language description together with the sketch of the general meaning representation. Hayati et al. [21] proposed *ReCode* which exploits sub-tree retrieval for explicitly referencing code examples within a neural code generation model. This approach adopts a retrieval-based neural machine translation approach in the context of source code generation. This is the first approach that introduced the retrieval technique to code generation. Murali et al. [36] proposed an approach that combines neural network and combinatorial search to generate source code from sketches, which is an abstract style of source code that masks user-defined constants and identifiers. Sun et al. [46] proposed a grammar-based structural convolutional neural network (CNN) that utilizes tree-based convolution, pre-order convolution, and attentive pooling layers for a more compact prediction than seq2seq models. The tree-based convolution applies a sliding window on the AST structures. Pre-order convolution that traverses the nodes of the partial AST. These two modules of CNN help capture the neighbor information of the trees. The attentive pooling is designed to aggregate the CNN features to predict better identifier names. Sun et al. [47] proposed *TreeGen* which is a tree-based neural architecture that uses the attention mechanism of Transformers to alleviate the long dependency issues and utilizes an AST encoder to incorporate grammar rules and AST structures into the network. They utilized a convolution sub-layer only on the first Transformer decoder blocks to prevent the Transformer to mix information from other nodes which may leak the original information. Lyu et al. [33] proposed a generation model that exploits the dependencies of the API method as an API dependency graph and utilizes the graph embedding into a seq2seq model. They incorporate an encoder-decoder module that captures both the global structural dependencies and textual program descriptions when generating the source code. Ahmad et al. [2] proposed *PLBART*, which is an auto-regressive transformer model that pre-trains on



an extensive collection of unlabeled Java and Python functions that are paired with natural language texts through denoising auto-encoders. This approach helps reduce the effort of mining highly human-curated annotations when fine-tuning specific tasks. They have evaluated *PLBART* on several tasks, i.e. code summarization, code generation, code translation, program repair, clone detection, and vulnerability detection.

API recommendation can be seen as a kind of code generation, which targets to generate a specific API set for a given query from developers [9–11, 32, 35, 44, 45, 51]. Liu et al. [32] proposed a method called *RecRank* that applies a novel ranking-based discriminative approach that leverages API usage path features to improve their performance on the recommendation. Xie et al. [51] studied the relationship between verb phrase patterns that describes the functionality of APIs and the user queries for finding those APIs. Chen et al. [11] proposed a novel API recommendation method that combines API usage with the text information in the source code based on an API context graph network. Chen et al. [10] propose COOK, which is an API sequence recommendation that combines deep learning models and post-processing strategies. They have enhanced the beam search with code-specific heuristics that helped them improve the quality of the recommendation.

## 6.2 Analysis of Model Effectiveness

There also have been numerous empirical studies that investigated the effectiveness of source code generation models in different contexts. Liu et al. [31] investigated the effectiveness of source code generation models when natural language requirement texts of general programming are given. They have built a large-scale dataset, *ReCa*, to assess five source code generation models in handling longer sequences of requirement texts. Chirkova and Troshin [12] conducted an empirical study on approaches that apply Transformers [49] to source code in the software engineering domain. They investigated how Transformers utilize the syntactic information of source code in modeling code completion, function naming, and bug fixing tasks. Antonio et al. [34] empirically assessed the effect of T5 (Text-to-Text Transfer Transformer) [43] architecture when applied to source code. They found that their T5 model outperforms four existing baselines when bigger data is used in pre-training. Peng et al. [41] did a study to unify the tasks and benchmark datasets of API recommendations to fully facilitate future studies. They grouped the different approaches concerning the input they handle (i.e. query or code). They propose a general benchmark dataset, *APIBENCH*, to assess 11 existing approaches in both query and code-based API recommendations.

These empirical studies assess the effectiveness of source code generation models in generating general programming tasks with respect to different levels of NL complexity [31] and different architecture [12, 34]. The difference in our work is that we investigate the effectiveness of generation models in generating machine/deep learning-related source code using ML libraries.

## 7 CONCLUSION

This paper conducts an empirical study to explore the performance of six state-of-the-art neural code generation models on ML programming tasks. A new benchmark dataset that contains 83K pairs of natural language described programming tasks and the corresponding programs implemented by four ML libraries is introduced. Our empirical study reveals some good, bad, and missing aspects, with a few major ones listed below. **(Good)** Neural code generation models perform significantly better on ML tasks than on non-ML tasks. **(Bad)** Most of the generated code is semantically incorrect. **(Bad)** Code generation models cannot significantly improve developers' completion time. **(Good)** The generated code can help developers write more correct code by providing developers with clues for using correct APIs. **(Missing)** The

observation from our user study reveals the missing aspects of code generation for ML tasks, e.g., decomposing code generation for divide-and-conquer into two tasks: API sequence identification and API usage generation.

## REFERENCES

- [1] Rajas Agashe, Srinivasan Iyer, and Luke Zettlemoyer. 2019. Juice: A large scale distantly supervised dataset for open domain context-based code generation. *arXiv preprint arXiv:1910.02216* (2019).
- [2] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 2655–2668.
- [3] Wasi Uddin Ahmad, Md Golam Rahman Tushar, Saikat Chakraborty, and Kai-Wei Chang. 2021. AVATAR: A Parallel Corpus for Java-Python Program Translation. *arXiv preprint arXiv:2108.11590* (2021).
- [4] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. 2019. Software engineering for machine learning: A case study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 291–300.
- [5] Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*. 65–72.
- [6] Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. 2021. GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow. *If you use this software, please cite it using these metadata* 58 (2021).
- [7] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.
- [8] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2021. Self-Supervised Contrastive Learning for Code Retrieval and Summarization via Semantic-Preserving Transformations. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 511–521.
- [9] Wing-Kwan Chan, Hong Cheng, and David Lo. 2012. Searching connected API subgraph via text phrases. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 1–11.
- [10] Chi Chen, Xin Peng, Bihuan Chen, Jun Sun, Zhenchang Xing, Xin Wang, and Wenyun Zhao. 2022. “More Than Deep Learning”: post-processing for API sequence recommendation. *Empirical Software Engineering* 27, 1 (2022), 1–32.
- [11] Chi Chen, Xin Peng, Zhenchang Xing, Jun Sun, Xin Wang, Yifan Zhao, and Wenyun Zhao. 2021. Holistic combination of structural and textual code information for context based API recommendation. *IEEE Transactions on Software Engineering* (2021).
- [12] Nadezhda Chirkova and Sergey Troshin. 2021. Empirical study of transformers for source code. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 703–715.
- [13] Anthony E Cozzie and Samuel King. 2012. Macho: Writing programs with natural language and examples. (2012).
- [14] Samip Dahal, Adyasha Maharana, and Mohit Bansal. 2021. Analysis of Tree-Structured Architectures for Code Generation. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*. 4382–4391.
- [15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).
- [16] Li Dong and Mirella Lapata. 2016. Language to logical form with neural attention. *arXiv preprint arXiv:1601.01280* (2016).
- [17] Li Dong and Mirella Lapata. 2018. Coarse-to-Fine Decoding for Neural Semantic Parsing. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 731–742.
- [18] Albert Gatt and Emiel Kraemer. 2018. Survey of the state of the art in natural language generation: Core tasks, applications and evaluation. *Journal of Artificial Intelligence Research* 61 (2018), 65–170.
- [19] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 933–944.
- [20] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).
- [21] Shirley Anugrah Hayati, Raphael Olivier, Pravalika Avvaru, Pengcheng Yin, Anthony Tomasic, and Graham Neubig. 2018. Retrieval-Based Neural Code Generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*.
- [22] Jin Huang and Charles X Ling. 2005. Using AUC and accuracy in evaluating learning algorithms. *IEEE Transactions on knowledge and Data Engineering* 17, 3 (2005), 299–310.
- [23] Qiao Huang, Xin Xia, Zhenchang Xing, David Lo, and Xinyu Wang. 2018. API method recommendation without worrying about the task-API knowledge gap. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 293–304.
- [24] Hui Jiang, Chulun Zhou, Fandong Meng, Biao Zhang, Jie Zhou, Degen Huang, Qingqiang Wu, and Jinsong Su. 2021. Exploring Dynamic Selection of Branch Expansion Orders for Code Generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. 5076–5085.

- [25] Rafal Jozefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. 2016. Exploring the limits of language modeling. *arXiv preprint arXiv:1602.02410* (2016).
- [26] Samina Khalid, Tehmina Khalil, and Shamila Nasreen. 2014. A survey of feature selection and feature extraction techniques in machine learning. In *2014 science and information conference*. IEEE, 372–378.
- [27] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436–444.
- [28] Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*. 74–81.
- [29] Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiský, Fumin Wang, and Andrew Senior. 2016. Latent Predictor Networks for Code Generation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 599–609.
- [30] Chang Liu, Xin Wang, Richard Shin, Joseph E Gonzalez, and Dawn Song. 2016. Neural code completion. (2016).
- [31] Hui Liu, Mingzhu Shen, Jiaqi Zhu, Nan Niu, Ge Li, and Lu Zhang. 2020. Deep learning based program generation from requirements text: Are we there yet? *IEEE Transactions on Software Engineering* (2020).
- [32] Xiaoyu Liu, LiGuo Huang, and Vincent Ng. 2018. Effective API recommendation without historical software repositories. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 282–292.
- [33] Chen Lyu, Ruyun Wang, Hongyu Zhang, Hanwen Zhang, and Songlin Hu. 2021. Embedding API dependency graph for neural code generation. *Empirical Software Engineering* 26, 4 (2021), 1–51.
- [34] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2021. Studying the usage of text-to-text transfer transformer to support code-related tasks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 336–347.
- [35] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. 2011. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*. 111–120.
- [36] Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. 2018. Neural Sketch Learning for Conditional Program Generation. In *International Conference on Learning Representations*.
- [37] Giang Nguyen, Stefan Dlugolinsky, Martin Bobák, Viet Tran, Alvaro Lopez Garcia, Ignacio Heredia, Peter Malík, and Ladislav Hluchý. 2019. Machine learning and deep learning frameworks and libraries for large-scale data mining: a survey. *Artificial Intelligence Review* 52, 1 (2019), 77–124.
- [38] Sajad Norouzi, Keyi Tang, and Yanshuai Cao. 2021. Code Generation from Natural Language with Less Prior Knowledge and More Monolingual Data. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*. 776–785.
- [39] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 574–584.
- [40] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.
- [41] Yun Peng, Shuqing Li, Wenwei Gu, Yichen Li, Wenxuan Wang, Cuiyun Gao, and Michael Lyu. 2022. Revisiting, Benchmarking and Exploring API Recommendation: How Far Are We? *IEEE Transactions on Software Engineering* (2022).
- [42] Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract Syntax Networks for Code Generation and Semantic Parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 1139–1149.
- [43] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, Peter J Liu, et al. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.* 21, 140 (2020), 1–67.
- [44] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. 2016. Swim: Synthesizing what i mean-code search and idiomatic snippet synthesis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 357–367.
- [45] Mohammad Masudur Rahman, Chanchal K Roy, and David Lo. 2016. Rack: Automatic api recommendation using crowdsourced knowledge. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 349–359.
- [46] Zeyu Sun, Qihao Zhu, Lili Mou, Yingfei Xiong, Ge Li, and Lu Zhang. 2019. A grammar-based structural cnn decoder for code generation. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 33. 7055–7062.
- [47] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2020. Treegen: A tree-based transformer architecture for code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 8984–8991.
- [48] Gias Uddin, Foutse Khomh, and Chanchal K Roy. 2020. Mining API usage scenarios from stack overflow. *Information and Software Technology* 122 (2020), 106277.
- [49] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.
- [50] Song Wang, Nishtha Shrestha, Abarna Kucheri Subburaman, Junjie Wang, Moshi Wei, and Nachiappan Nagappan. 2021. Automatic Unit Test Generation for Machine Learning Libraries: How Far Are We?. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1548–1560.
- [51] Wenkai Xie, Xin Peng, Mingwei Liu, Christoph Treude, Zhenchang Xing, Xiaoxin Zhang, and Wenyun Zhao. 2020. API method recommendation via explicit matching of functionality verb phrases. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and*

- Symposium on the Foundations of Software Engineering*. 1015–1026.
- [52] Frank F Xu, Zhengbao Jiang, Pengcheng Yin, Bogdan Vasilescu, and Graham Neubig. 2020. Incorporating External Knowledge through Pre-training for Natural Language to Code Generation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*.
  - [53] Ziyu Yao, Daniel S Weld, Wei-Peng Chen, and Huan Sun. 2018. Staqc: A systematically mined question-code dataset from stack overflow. In *Proceedings of the 2018 World Wide Web Conference*. 1693–1703.
  - [54] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *2018 IEEE/ACM 15th international conference on mining software repositories (MSR)*. IEEE, 476–486.
  - [55] Pengcheng Yin and Graham Neubig. 2017. A Syntactic Neural Model for General-Purpose Code Generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 440–450.
  - [56] Pengcheng Yin and Graham Neubig. 2018. TRANX: A Transition-based Neural Abstract Syntax Parser for Semantic Parsing and Code Generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. 7–12.
  - [57] Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. 2022. CERT: Continual Pre-Training on Sketches for Library-Oriented Code Generation. *arXiv preprint arXiv:2206.06888* (2022).
  - [58] Jie M Zhang, Mark Harman, Lei Ma, and Yang Liu. 2020. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering* (2020).

Received 20 February 2022; revised 12 March 2022; accepted 5 June 2022