On the Effectiveness of Pretrained Models for API Learning

Mohammad Abdul Hadi
University of British Columbia
Kelowna, Canada
mohammad.hadi@ubc.ca

Imam Nur Bani Yusuf
Singapore Management University
Singapore
imamy.2020@phdcs.smu.edu.sg

Ferdian Thung
Singapore Management University
Singapore
ferdianthung@smu.edu.sg

Kien Gia Luong
Singapore Management University
Singapore
kiengialuong@smu.edu.sg

Jiang Lingxiao
Singapore Management University
Singapore
lxjiang@smu.edu.sg

Fatemeh H. Fard
University of British Columbia
Kelowna, Canada
fatemeh.fard@ubc.ca

David Lo
Singapore Management University
Singapore
davidlo@smu.edu.sg

ABSTRACT

Developers frequently use APIs to implement certain functionalities, such as parsing Excel Files, reading and writing text files line by line, etc. Developers can greatly benefit from automatic API usage sequence generation based on natural language queries for building applications in a faster and cleaner manner. Existing approaches utilize information retrieval models to search for matching API sequences given a query or use RNN-based encoder-decoder to generate API sequences. As it stands, the first approach treats queries and API names as bags of words. It lacks deep comprehension of the semantics of the queries. The latter approach adapts a neural language model to encode a user query into a fixed-length context vector and generate API sequences from the context vector.

We want to understand the effectiveness of recent Pre-trained Transformer based Models (PTMs) for the API learning task. These PTMs are trained on large natural language corpora in an unsupervised manner to retain contextual knowledge about the language and have found success in solving similar Natural Language Processing (NLP) problems. However, the applicability of PTMs has not yet been explored for the API sequence generation task. We use a dataset that contains 7 million annotations collected from GitHub to evaluate the PTMs empirically. This dataset was also used to assess previous approaches. Based on our results, PTMs generate more accurate API sequences and outperform other related methods by ~11%. We have also identified two different tokenization approaches that can contribute to a significant boost in PTMs' performance for the API sequence generation task.

CCS CONCEPTS

• **Software and its engineering → Reusability**.

KEYWORDS

API, Deep Leaning, Transformers, code search, API sequence, API usage

1 INTRODUCTION

Developers frequently rely on existing class libraries to implement particular functionalities by invoking APIs that correspond to the libraries. It is extremely helpful to identify which APIs to use and in what order developers should invoke the API-related methods. For developers, who may need to face a steep learning curve, learning and becoming acquainted with an unfamiliar library can be challenging. It is not uncommon for a library as large as JDK to have hundreds or thousands of APIs, but they are not well documented regarding the usage patterns of API methods. Our literature study reveals that inadequate or absent resources cause obstacles for learning APIs, and a major challenge for API users is to discover the subset of the APIs that can help complete a task.

Search engines, such as Google and Bing, are commonplace to discover APIs and their usage sequences. However, the general web search engines are not designed to cater specifically to programming or code-related tasks, so they are often deemed ineffective when it comes to assisting with such tasks. Currently, there is no easy way for developers to learn about APIs and their usage sequences from the web pages returned by the search engines, which the search engines retrieved based on keyword matching without considering the semantics of natural language queries, making it more difficult to locate relevant code snippets and APIs associated with them.

Using a statistical word alignment model, Raghothaman et al. recently proposed SWIM, to translate a natural language query into a list of possible APIs [35]. To retrieve relevant API sequences, SWIM applies UP-Miner [43], a utility pattern mining toolbox, to a list of APIs. This method has a critical problem; it ignores the ordering of the natural language queries as these queries are treated as a bag of words. Therefore, it cannot recognize a deep semantic meaning within the natural language queries. This critical problem was addressed by DeepAPI [13], a deep-learning-based method that translates a natural language query into relevant API usage sequences. API learning has been formulated as a machine translation problem: given a natural language query, the goal is to translate it into an API sequence. DeepAPI leverages RNN-based encoder-decoder architecture, and also adapts attention mechanism cite(attention paper). It has demonstrated a deep understanding of natural language queries in two aspects: learning the semantics of words by embedding them in context-vector representations of context and understanding the sequence of words in a natural language query and the associated APIs instead of focusing on word-to-word alignment.

DeepAPI is trained on a corpus containing pairs of API sequences and the corresponding natural language queries. The encoder learns how to encode each query into a fixed-length context vector for the decoder to use for decoding the API sequence. In the training phase, the DeepAPI learns the semantics of the queries and the corresponding API sequences and accordingly updates its parameters. After the training phase, DeepAPI can generate the API sequences given a query describing the functionality of the intended API.

During the training phase , the encoder and the decoder in DeepAPI is focused on learning: 1) the semantics of the queries in the input and the corresponding API sequences in the output, and 2) the correlations between the input and the output sequences. We want to separate these two tasks and propose to leverage a Pre-trained Transformer model (PTM). PTM is a model that leverages Transformer encoder-decoder architecture(cite transformer paper) that has been pre-trained using certain techniques on a data-rich corpora in the pre-training phase. Because a PTM has been pre-trained on a data-rich corpora beforehand, it has a sense of understanding the semantic meaning of the queries and the corresponding API sequences. Intuitively, the learning burden of the model can be reduced if the model already has such a knowledge in the beginning of its training. Hence, the model can focus to learn the correlation between the input and the output sequences in the training phase.

In our study, we aim to answer the following research questions:

RQ1: Can PTMs yield better performance than the prior approaches for API learning task?

RQ2: Can different tokenization approaches help the PTMs perform better?

In the first research question, we explore how the current existing PTMs perform compared to the other existing tools, such as SWIM and DeepAPI. The existing tools are based on the curated code-search and pattern mining algorithms, and also deep learning-based approaches. Our results shows that PTMs can outperform the prior approaches by ~3% to ~7% in terms of BLEU score.

The second research question deals with whether we could boost the performance of PTMs by introducing different tokenization techniques. Tokenization is a technique to split a piece of text or

sentence into a smaller unit called token <insert citation here>. Tokenization may affect the performance of a model because it affects how the input is represented to the model. We have proposed two tokenization techniques. Leveraging our customized tokenization approach, our results demonstrate that the PTMs can outperform the prior state of the art by ~11% in terms of BLEU Score.

The main contribution of our paper is stated below:

- To the best of our knowledge, we are the first who adapt PTMs for the API sequence generation task.
- We propose two novel tokenization techniques to help improve the performance of the PTMs in the API sequence generation task.
- We evaluate the existing PTMs against existing approaches using a corpus containing 7 million pairs of Java API sequences and the corresponding queries used in the prior study <cite the deep api learning paper here>. Our results demonstrate that PTMs can outperform the prior approaches by ~3% to ~7% in terms of BLEU score.Leveraging our tokenization technique, PTM can outperforms the prior best approach by ~11% in terms of BLEU Score.
- We provide a replication package and open source our codes and dataset to help fellow researchers and practitioners to reproduce our results. [1]

Below is an outline of the remainder of this paper. Section 2 introduces background knowledge. Sections 3 and 4 describe the methodology of our approach followed by the results. Section 5 discusses the interpretations and implications of our findings, along with the threats to the validity. Section 6 presents the related works. Finally, we conclude our study and present future work in Section 7.

2 PRELIMINERIES

In this section, we will discuss three different approaches used for API sequence generation from natural language queries: Information Retrieval Models, Deep Learning Approaches without Pretrained Knowledge, and Deep Learning Approaches with Pretrained Knowledge.

2.1 Information Retrieval Models

2.1.1 Combination of Code Search and Pattern Mining for API Sequence generation. Researchers have used code search using information retrieval techniques [16, 21, 24, 26] to identify API sequences in a code corpus that correspond to a given query, followed by the use of an API usage pattern miner [10, 46, 49] to determine the best API sequences from the returned code snippets. Following previous literature, we used Lucene [2] to perform a code search using a natural language query, while UP-Miner is used to analyze API usage patterns. A text retrieval engine such as Lucene treats source code as plain text documents, which is how it builds a source code index and performs text retrieval. Pattern mining tool UP-Miner [46] analyzes snippets of codes based on the results of a Lucene search and finds API sequence patterns . In this procedure, API sequences extracted from code snippets are clustered and then frequent patterns are identified. Lastly, it clusters the frequently

---

occurring patterns to reduce redundancy. For this experiment, we used the same code corpus as those used for evaluating PTMs and compared BLEU scores to those from PTMs.

2.1.2 SWIM. In [35], Raghothaman et al. transformed natural language queries into a list of possible APIs using statistical word alignment models [2]. Based on the predefined API list, SWIM retrieves relevant APIs. Its statistical word alignment model, however, is based on a bag-of-words assumption that does not consider how words are arranged in natural language queries and APIs are positioned in the result. Also, the model does not take contextual information into account when producing API sequences from natural language input, and therefore cannot recognize deep semantics of natural language queries. As an example, the model does not distinguish between the query 'convert txt to CSV' and 'convert CSV to txt.'

SWIM uses the statistical word alignment model to extract a list of relevant APIs based on the keywords in the given natural language query [2] and then leverages Lucene to search for related API sequences from the extracted API list. During our experiment, we used the same dataset for evaluating SWIM as we did for evaluating PTMs.

2.2 Deep Learning Approaches without Pretrained Knowledge

DeepAPI [13] generates API usage sequences relevant to a natural language query using deep neural networks. It converts the machine translation problem into the API learning problem. DeepAPI's understanding of natural language queries is demonstrated on two fronts. Firstly, it recognizes semantically related words by embedding them in a vector representation of context instead of matching keywords. Secondly, DeepAPI learns a language query's sequence of words and the associated APIs' sequence separately instead of aligning them to each other. Therefore, DeepAPI can distinguish semantic differences between queries and generate API sequences accordingly. DeepAPI adapts RNN Encoder-Decoder. Using DeepAPI, a language model is trained that encodes each sequence of words (annotation) into a fixed-length context vector, and the context vector is used to decode API sequences. The model is then used to generate API sequences in response to API-related user queries.

2.3 Deep Learning Approaches with Pretrained Knowledge

A more recently established and widely accepted practice in Natural Language Processing (NLP) is using Pre-Trained Language Models (PTM) and then transfer its learned knowledge to various downstream NLP tasks, such as sentiment analysis, question answering, or classification [33]. In NLP, PTMs (such as BERT) are large language models that are trained on large natural language corpora using a deep neural network in an unsupervised manner [52]. These models are then fine tuned for various downstream tasks using limited labeled datasets. For example, BERT [7] is a PTM that is frequently being used for question answering and sentiment classification tasks. As PTMs are trained on large general domain corpora, they learn contextual linguistic information and eliminate the need to train downstream task models from scratch

[45]. PTMs reduce the amount of effort (i.e., new model development time per task) to build models for each task separately, and they reduce the amount of required labeled dataset [22]. Consequently, PTMs are used to transfer the learned knowledge to a new domain or a new task, and in settings where a model has not seen any example during training (known as zero-shot learning) [47]. Although PTMs are used extensively and led to many advances in NLP, their applicability for API Learning and Sequence Generation is barely known [14] but to what extent PTMs can be applied for API Learning problem is largely unknown.

2.3.1 BERT. Devlin et al. [7] designed Bidirectional Encoder Representations from Transformers (BERT) to learn contextual word representations from unlabeled texts. Contextual word embeddings designate a word's representation based on its context by capturing applications of words across different contexts. BERT employed a bidirectional encoder to learn the words' contextual representations by optimizing for Masked Language Model (MLM) and Next Sentence Prediction (NSP) tasks. For MLM, 15% of all the tokens are replaced with a masked token (i.e., [MASK]) beforehand, and the model is trained to predict the masked words, based on the context provided by the non-masked words. For NSP, the model takes sentence-pairs as input for learning to predict whether a pair-match is correct or wrong. During training, 50% of the inputs are true consequent pairs, while the other 50% are randomized non-consequent sentence-pairs. Devlin et al. trained two versions: small-sized $BERT_{BASE}$ and big-sized $BERT_{LARGE}$. $BERT_{BASE}$ is a smaller model with 12 layers and 110 million parameters. $BERT_{LARGE}$ has 24 layers and 340 million parameters. $BERT_{LARGE}$ is more computationally expensive and consumes more memory compared to $BERT_{BASE}$. Please note that based on the results reported in the BERT paper, $BERT_{LARGE}$ always exceeds $BERT_{BASE}$. Although we have not used BERT in our study, this is the building block for the pre-trained models that we are going to discuss in this section.

2.3.2 RoBERTa. Robustly optimized BERT approach (RoBERTa) outperformed all the state-of-the-art benchmarks upon release [22]. Liu et al. modified BERT's pretraining steps that yield substantially better performance on all the classification tasks. RoBERTa increased the amount of mini-batch sizes, data, and training time to train the model. RoBERTa is also trained on dataset that includes longer sequences than before. The masking pattern in RoBERTa was also modified to be generated spontaneously.

Reason behind choosing RoBERTa: RoBERTa outperforms BERT on nine different NLP tasks on the GLUE benchmark. Based on these results, RoBERTa can present a reasonable choice for PTM in our study.

2.3.3 CodeBERT. CodeBERT [9] was developed using a multilayered attention-based Transformer model, BERT [7]. As a result of its effectiveness in learning contextual representation from massive unlabeled text with self-supervised objectives, the BERT model has been adopted widely to develop large pre-trained models. Thanks to the multilayer Transformer [44], CodeBERT developers adopted two different approaches than BERT to learn semantic connections between Natural Language (NL) - Programming Language (PL) more effectively.

Firstly, The CodeBERT developers make use of both bimodal instances of NL-PL pairs (i.e., code snippets and function-level comments or documentations) and a large amount of available unimodal codes [23]. In addition, the developers have pre-trained CodeBERT using a hybrid objective function, which includes masked language modeling [7] and replaced token detection [6]. The incorporation of unimodal codes helps the replaced token detection task, which in turn produces better natural language by detecting plausible alternatives sampled from generators.

Developers trained CodeBERT from Github code repositories in 6 programming languages, where only one pre-trained model is learned for all six programming languages with no explicit indicators used to mark an instance to the one out of six input programming languages. CodeBERT was evaluated on two downstream tasks: natural language code search and code documentation generation. The study found that fine-tuning the parameters of Code-BERT obtained state-of-the-art results on both tasks.

2.3.4   PLBART. PLBART (Program and Language BART) [1] is a sequence-to-sequence model capable of performing a broad spectrum of Code summarization, generation, and translation tasks. We have chosen to use PLBART in our study as it is pre-trained on a vast collection of Java and Python functions and associated NL text.

The motivation behind PLBART was to develop a general-purpose model that can be used in various Program and Language Understanding and Generation (PLUG) applications. A recent development in deep learning coupled with the accessibility of large-scale Programming Language (PL) and associated Natural Language (NL) data led to the automation of PLUG applications. A fundamental feature of PLUG applications is that they require a thorough understanding of program syntax the interdependencies between the semantics of PL and NL. A few research efforts in learning general purposes PL-NL representation encoders, such as CodeBERT, but that too is pretrained on a small-scale bimodal data (code-text pairs). Despite the effectiveness of existing methods, they do not have a pretrained decoder for language generation. Therefore, they still require a large amount of parallel data to train the decoder. The unified PLBART includes PL and NL pre-training on unlabeled data to learn multilingual representations applicable to many PLUG applications.

Reason for including PLBART: PLBART outperforms state-of-the-art models for code summarization, code generation, and code translation in seven programming languages. Furthermore, PLBART's effectiveness in program understanding is demonstrated by experiments on program repair, clone detection, and vulnerable code detection.

3   METHODOLOGY

In this section, we will discuss our dataset, experimental setup, and evaluation metric to measure the performance of the PTMs against the prior approaches.

3.1   Dataset

We leverage a large scale API sequence to annotation corpus provided by Gu et. al. [13], who propos DeepAPI. The description of the dataset and the collection process is described as the following.

Gu et. al. curate a large-scale database that contains pairs of API sequences and natural language annotations. They collect 442,928 Java projects' last snapshots from GitHub. Each of the project has at least one star. After the collection, pairs of <API sequence, annotation> are extracted by parsing source code files into Abstract Syntax Trees (ASTs) using Eclipse's JDT compiler [3]. The dependencies of an entire project repository is analyzed. All field declarations and type bindings for all classes are then documented, and all object types are replaced with the corresponding class types. Subsequently, the API sequence from each method body is extracted using an AST traversal.

The authors appended "APIName.new" to the API sequence after each constructor call, and "APIName.methodName" to the API sequence after each method call of a JDK class instance. When a method is called with a parameter, the parameter methods are appended before the calling method. They derive the API sequence of a sequence of statements by extracting and concatenating the API sequence of each statement consecutively. For conditional and loop statements, the authors sequentially put all the possible branches to extract APIs from the branch statements, respectively. Based on the Javadoc guidance [4], the first sentence of a documentation comment for a method is used as a summary of the corresponding method. The authors traverse the method's AST and used the Eclipse JDT compiler to extract the method summary. The method summary is then used as the natural language description for the corresponding API sequence. Following the process, the Gu et. al. obtain a database consisting of 7,519,907 <API sequence, annotation> pairs. Gu et. al. do not separate the dataset into training, validation, and testing sets specifically. So, we randomly selected 10,000 <API sequence, annotation> out of ~500,000 pairs for the testing set and 7M for training and validation sets.

3.2   Experimental Setup

3.2.1   Baseline Selection: Following the literature, we select three prior approaches as the baseline for evaluating the existing PTMs. These approaches are the Lucene+UPMiner, SWIM, and DeepAPI. We reimplement these three approaches to make sure that all experiments are performed in the same environment and leverage the same training, validation, and testing set. We believe that our implementations are correct as the results are slightly different with the results reported in the prior study [13]. We conjecture that the difference in performance may have resulted from using training and testing data sets that are different from the original implementation.

For Lucene+UPMiner, we perform the following pre-processing steps. First, we treat the Java codes that are retrieved from GitHub as plain texts. Subsequently, we remove the new line (i.e., '\n') and tab (i.e., '\t'). Then, we use Lucene to index the pre-processed code snippets, and apply UPMiner to mine API usage sequences from pre-preocessed code snippets to find the frequent API usage patterns. Following [13], we do not fine-tune the Lucene+UP-Miner approach. For SWIM, we train the word alignment model, and subsequently build the API indexes using the training set. Given a natural language query, SWIM leverage the word-alignment model

---

[3]http://www.eclipse.org/jdt/
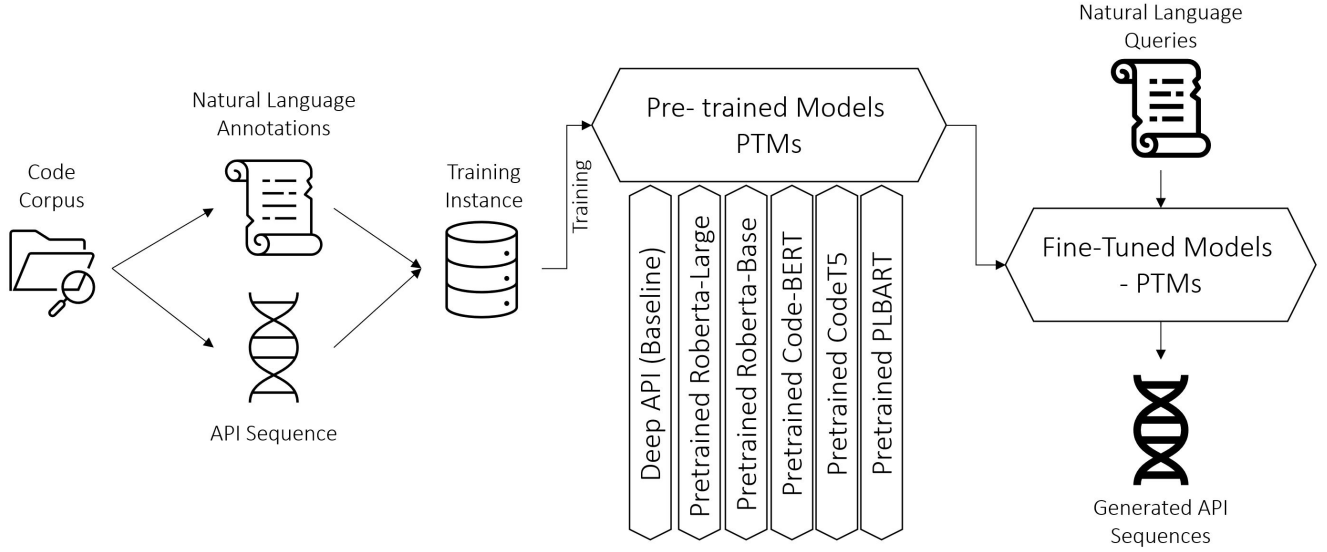[4]http://www.oracle.com/technetwork/articles/java/ index-137868.html

Figure 1: Workflow of our Experiments

to return a list of relevant API sequences, which is used for code synthesis in the later step. However, we only adapt the word alignment model and discard the code synthesis part as we are only interested in the API sequence generation task. For DeepAPI [13], we leverage the original implementation provided by the authors. To train DeepAPI, we set the batch size to 200 and limit the source and target vocabularies to the top 10,000 words that are appear most frequently in the queries and API sequences.

We execute all the experiments on a Linux machine with Intel 2.21 GHz CPU and 16GB memory with two NVIDIA Tesla V100 32GB GPU.

3.2.2 Fine-tuning PTMs: With the PTMs, the API sequence generation task is framed as an NL-to-PL translation problem. The PTM's encoder converts a natural language query into a context vector, and the respective PTM's decoder leverage the context vector to generate the corresponding API sequences.

We fine-tune the off-the-shelf available PTMs by freezing two-thirds of the lower-level layers of the PTMs and unfreezing the rest of the upper-level layers. For example, for RoBERTa-Large, we freeze the 16 lower-level layers out of 24 layers following a similar setting as [2]. All the PTMs are fine-tuned with the same hyperparameter settings, e.g., the batch_size, learning_rate, and num_of_epochs are the same for all the PTMs.

As RoBERTa has been used as the codebase of most of the considered PTMs, we reuse the RoBERTa model's parameters as the starting point. Before feeding the input queries into the model, we also leverage the corresponding PTM's tokenizer to preprocess the input into the format that the model can understand. The PTMs are then fine-tuned using the preprocessed data and tested on the held-out test set. We adopt the following values of hyper-parameter from the literature [52] to fine-tune the PTMs; batch size: 16, and learning rate (AdamW): 2e-5. Further, we train the PTMs and until 50th epochs. Table 1 shows the details and configuration of each PTM.

Table 1: PTM Description

Architecture	Used Model	Parameters	Layers	Hidden	Heads
RoBERTa	roberta-base	125M	12	768	12
RoBERTa	roberta-large	355M	24	1024	16
CodeBERT	roberta-base	125M	12	768	12
PLBART	bart-base	140M	6	768	12
Code-T5	t5-base	220M	12	768	12

3.3 Evaluation Metrics

We use the following metric to evaluate the performance of PTMs against the prior approaches.

3.3.1 BLEU:. We use BLEU score to evaluate how close are the generated API sequences with the ground truth. BLEU score measures the similarity of a candidate sequence to the reference sequence based on the number of n-gram hits. Equation 1 is used to compute the BLEU score.

$$BLEU = BP \cdot exp(\sum_{n=1}^{N} w_n log p_n) \qquad (1)$$

$$p_n = \frac{\text{\# of n-grams appear in the reference + 1}}{\text{\# of n-grams of candidate + 1}} \text{ for n = 1, ... , N} \qquad (2)$$

$$BP = \begin{cases} 1 & \text{if c > r} \\ e^{(1-r/c)} & \text{if c} \leq \text{r} \end{cases} \qquad (3)$$

In Equation 1, p_n refers the n-gram hits between the candidate sequence and the reference, and w_n equals to $1/N$ where N is the number of grams to be considered. We set N equals to 4. Further, BP is the penalty for short candidate that may yield higher n-gram precision. BP is calculated using Equation 3. In the Equation 3, r and c refer to the length of reference and candidate sequences.

In this paper, we express BLEU score in the percentage. A higher BLEU score indicates that the candidate sequence is very similar with the reference sequence. When the candidate sequence matches exactly the reference sequence, the BLEU score becomes 100%.

4 RESULTS

We aim to answer the following research questions:

RQ1: Can PTMs yield better performance than the prior approaches?
RQ2: Can different tokenization techniques boost the performance of the PTMs?

4.1 RQ1: Can PTMs yield better performance than the prior approaches?

We compare PTMs with the three aforementioned approaches as the baseline. The first is Lucene+UP-Miner; a combination of Code Search and Pattern Mining where Lucene is used to perform a code search using a natural language query, and UP-Miner [46] is used to analyze the API usage patterns. The second is SWIM that leverages the statistical word alignment model to learn the API usage patterns and then uses Lucene to search for the relevant API sequences given a natural language query. The last one is DeepAPI [13]; a deep learning-based approach that leverages RNN-based encoder-decoder architecture that models the API sequence generation task as a machine-translation problem, where natural language queries are the input and the API sequences are the output.

Table 2 shows the results of our experiments. In Table 2, Deep-API-Auth is the author's reported BLEU score in [13], while Deep-API-Rep is the score from our experiment that leverages the authors' replication package. Similarly, SWIM and Lucene+UP-Miner are also the scores that are obtained from our experiment. Table 2 indicates that our results are slightly different from the originally reported scores in [13]. Moreover, our results also demonstrate the same trend as the prior study [13]; DeepAPI outperforms Lucene+UP-Miner and SWIM by a significant margin.

Table 2: Results on the Baselines

Name of the Model	BLEU Score
Deep API - Rep	52.25
Deep API - Auth	54.42
SWIM	17.16
Lucene+Up-Miner	21.73

We compare the BLEU score achieved by the PTMs against the BLEU score produced by Deep-API-Rep in Table 3. Overall, the results indicates that the PTMs can outperforms Deep-API-Rep. Specifically, CodeBERT, RoBERTa-LARGE, CodeT5, and PLBART perform better than Deep-API-Rep by 3.72%, 5.26%, 5.82%, and 6.93% in terms of BLEU Score.
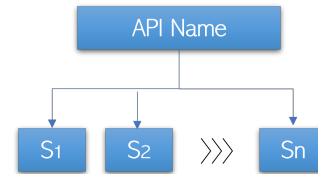
Table 3: PTM's performance

Name of the Model	Checkpoints	BLEU Score	Improvement
RoBERTa-Large	50/50th epoch	57.88	5.63%
RoBERTa-Base	50/50th epoch	51.76	-0.49%
CodeBERT	50/50th epoch	56.34	4.09%
Deep API - Rep	50/50th epoch	52.25	
PLBART	50/50th epoch	**59.55**	**7.3%**
Code-T5	50/50th epoch	58.44	6.19 %

Findings of RQ1

Most off-the-shelf PTMs can outperform Deep-API-Rep by considerable margin, ranging from 3.72% to 6.93% in terms of BLEU score. The PTM that is pre-trained on natural language corpora (i.e., RoBERTa) requires bigger architecture (e.g., more parameters, layers, heads) to outperform Deep-API-Rep.

4.2 RQ2: Can different tokenization techniques boost the performance of the PTMs?

We implement two tokenization techniques to investigate if they can possibly help the PTMs perform better. Given an API name, the default PTMs' tokenizers break down the API names into a sequence of subword tokens as illustrated in Figure 2.

Figure 2: Original Tokenization Approach

In the original tokenization approach as shown in Figure 2, the decoder tries to learn how to rearrange the tokens S_1, S_2, …, S_n to generate valid API names during the fine-tuning. This helps in Natural Language Processing (NLP), to make sense of the grammar and different uses of lemma in different contexts. But API names are constant, and we want to put more focus on the sequence, rather than API-names. Also, it is difficult for the decoder to ignore all the information these sliced token already acquired during the respective PTM's pretraining.

We decided to try a modified tokenization approach, where the API-names will be directly injected to vocabulary by averaging their corresponding token-slices' embeddings. Here, we let the tokenizer break down API-Names into slices; we keep track of the broken down token-slices. Later, we add all the API-Names as separate Unknown embeddings to the Vocabulary, such as $< UNK - 1 >$, $< UNK - 2 >$, . . . , $< UNK - n >$. The Unknown tokens get initialized with the average of the embedding values of tracked token-slices. An illustration of the process is provided in Fig 3.

From Table 4, we observe that the Tokenization Approach 1 (TA1) enhances the performance of the PTMs by a large margin. Leveraging the default tokenization, RoBERTa-BASE could not
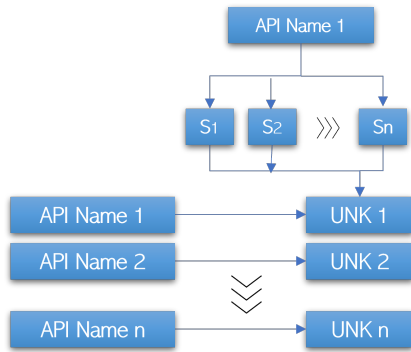
Figure 3: Modified Tokenization Approach 1 (TA1)

outperform Deep-AP-Rep as shown previously in Table 3. After leveraging our Tokenization Approach TA1, RoBERTa-BASE along with all the other PTMs can yield better performance than Deep-API-Rep. Specifically, RoBERTa-BASE, CodeBERT, RoBERTa-LARGE, CodeT5, and PLBART yield 2.51%, 8.02%, 8.58%, 9.23%, and 10.46% better BLEU Score than Deep-API-Rep respectively. Moreover, the performance of RoBERTa-LARGE, RoBERTa-BASE, CodeBERT, PLBART, and CodeT5 when using Tokenization Approach TA1 (see Table 4) are better by 3.32%, 3.37%, 4.3%, 3.53%, and 3.41% in terms of BLEU score when compared against the original tokenization approach (see Table 3).

Table 4: PTM's performance: Tokenization Approach 1 (TA1)

Name of the Model	Checkpoints	BLEU Score
RoBERTa-Large	50/50th epoch	61.2
RoBERTa-Base	50/50th epoch	55.13
CodeBERT	50/50th epoch	60.64
Deep API - Rep	50/30th epoch	52.25
PLBART	50/50th epoch	**63.08**
Code-T5	50/50th epoch	61.85

The Tokenization Approach 1 (TA1) can introduce a problem as the subword tokens retain a lot of natural language information from the pre-training corpora. Such an unrelated information is later contributes to the embedding representation of the whole API names, which we conjecture to have induced some noises towards the representation.

To circumvent the problem, we propose the second Tokenization Approach TA2, where the tokenizer does not break down the API names into subword tokens. First, we add all the API-Names (e.g., $< API-1 >, < API-2 >, \ldots$ shown in Figure 4) as separate tokens to the vocabulary. $< API-n >$. These API names are then converted the corresponding vector representations (<UNK1,2,…,n> in Figure 4). These representations are one-hot encoded vectors. Intuitively, Tokenization Approach 2 (TA2) can help the decoder to put more focus on the API sequences instead of learning to rearrange the subword tokens.
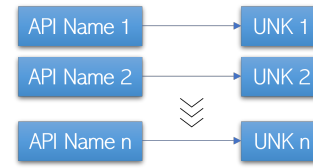

Figure 4: Modified Tokenization Approach 2 (TA2)

Table 5: PTM's performance: Tokenization Approach 2 (TA2)

Name of the Model	Checkpoints	BLEU Score
RoBERTa-Large	50/50th epoch	62.25
RoBERTa-Base	50/50th epoch	55.81
CodeBERT	50/50th epoch	61.51
Deep API - Repl	50/50th epoch	52.25
PLBART	50/50th epoch	63.19
Code-T5	50/50th epoch	62.31

From Table 5, the results indicates that Tokenization Approach 2 (TA2) can yield slightly better performance than when using Tokenization Approach - 1 (TA1). Specifically, RoBERTa-BASE, CodeBERT, RoBERTa-LARGE, CodeT5, and PLBART produced 1.05%, 0.68%, 0.87%, 0.11%, and 0.46% better BLEU Score than their implementation with TA1, respectively.

> **Findings of RQ2**
>
> Our Tokenization Approaches can help PTMs to better learn API-Sequences. Both Tokenization Approach 1 (TA1) and Tokenization Approach 2 (TA2) perform better than the default tokenization, as shown in Table 4 and Table 5. Preventing the tokenizer from decomposing API names into subword tokens can help improve PTMs' performance, if handled properly.

5 DISCUSSION

This section details the lessons we learned from our experiments and discusses threats to their validity.

5.1 Lessons Learned

Reasons behind PTMs generally performing better

The semantic gap between API sequences and natural language descriptions presents a major challenge for API learning. Generally, information retrieval approaches assume that the natural language and code are made up of a bag of words without adequately understanding the high-level semantics of the language. In contrast, the existing deep learning-based approaches focus on two distinct things during pre-training: 1) learning the semantics of natural language and programming language, and 2) understanding how the model can correlate between the natural language queries and the intended API sequences. However, the PTMs have already accumulated the semantic knowledge of natural and programming languages in the pre-training phase. Consequently, when PTMs are finetuned using the ⟨API Sequence, Annotation⟩ pairs, instead of

focusing on the semantics of natural and programming languages and how given queries contribute to the generation of the corresponding API sequences at the same time, they can only focus on the latter task to be more effective.

To achieve better performance, the PTMs rely on three distinct advantages of deep learning-based models as well as the retained NL or NL-PL linguistic knowledge from the pre-training phase. These advantages are: using contextualized word embeddings to make semantic sense of the natural language queries, understanding the sequences order of the queries instead of treating it as a bag-of-words, and generating a translated sequence instead of searching specific samples for the correct arrangement of output APIs [13].

Fine-tuning PTMs pre-trained on NL corpora with large architecture can help to learn API better

From Table 3, we observe that between two PTMs pre-trained on Natural Language (NL) corpora, i.e., RoBERTa-BASE and RoBERTa-LARGE, the latter outperforms the baseline Deep-API-Rep, unlike the BASE model. So, it is recommended that researchers, developers, or practitioners should go for fine-tuning the larger architecture of any given NL pre-trained PTM for achieving better performance. The same performance as the LARGE models can also possibly be achieved by tweaking the parameters during the pre-training of the BASE models. Researches in the neural machine translation field has shown that small mini-batches combined with the appropriate learning rate and step size [27] can improve the performance of a deep learning model. Prior study [22] also confirms that RoBERTa is also amenable to large batch training and proportionate step size and learning rate. If tweaking parameters during pre-training of the smaller models is not an option, we recommend using bigger architectures instead.

Fine-tuning PTMs pre-trained on NL-PL corpora, with those with small architecture, can yield better performance than those that are pre-trained only on NL corpora.

We also found that PTMs pre-trained on NL-PL corpora (e.g., CodeBERT, CodeT5, and PLBART) can outperform the other PTMs that are pre-trained only on NL corpora. We conjecture that this performance boosts stem from to the bimodal (NL-PL) data and code-related objective tasks that are used to pre-train the the models in the pre-training phase.

Different tokenization approaches can possibly boost the performance of the PTMs

As part of the original tokenization approach, the decoder learns to rearrange predefined tokens to create the API name during the fine-tuning phase. This approach helps Natural Language Processing (NLP) tasks by making sense of grammatical structure and the different uses of the lemma in different contexts; however, API names are invariant in our case, and we prefer to focus on the sequence generation rather than API names.

Our Tokenization Approach 1 (TA1) lets the tokenizer break down API names into subword tokens while tracking to which API names each token belongs. Later, we add all the API names as separate Unknown embeddings to the Vocabulary, which get initialized with the average embedding values of the tracked subword tokens. Tokenization Approach 1 (TA1) enhances the performance of the PTMs by a large margin. However, this approach is not completely

without disadvantages. In Tokenization Approach 1 (TA1), those subword tokens retain a lot of textual information from their respective pre-training corpora, which later causes some noise in the API representations. Therefore, we propose a simpler tokenization method that initially hides API-Names from the tokenizer and then adds all API-Names to the Vocabulary as one-hot encoded embeddings. Overall, the PTMs are marginally can perform better when using this approach (TA2) than when they use Tokenization Approach (TA1).

Pre-training and objective tasks of the PTMs play a huge role in downstream tasks

Our study leverage a dataset containing Java API sequences that is curated from GitHub. All PTMs are pre-trained on NL-PL corpora, except for RoBERTa-BASE and LARGE that are pre-trained on only NL corpora. Our results demonstrate that the PTM pre-trained on NL-PL corpora can perform better than the PTMs pre-trained on only NL corpora. Such a finding also aligns with the prior study [1]. Hence, researchers and practitioners should pay special attention before selecting PTMs that have been pre-trained on unrelated domains. The closer the pretraining domain is for the PTMs, the better PTMs can perform for the downstream tasks. Also, not all readily available PTMs make their decoder open-sourced and not all PTMs are pre-trained on a machine translation objective task. As we frame our API Sequence generation task to a machine translation task from natural language to code, the PTMs that have already pre-trained on machine translation task can perform better.

Modified tokenization approach help all the PTMs uniformly

With the proposed tokenization techniques, all PTMs gain almost equal performance gain for the downstream API learning task. We can observe that all PTMs shows a general improvement in their performance and maintained their original ranking, even after implementing the modified tokenization approach. We hypothesize that their underlying common Transformer model, uniform parameter settings, and architecture are responsible for the almost consistent increase in performance.

5.2 Threats to Validity

There are several threats to validity identified in this study. As we have adopted different tools, dataset, approaches, and methods from the literature, we also assume their latent threats to validity:

A wide range of PTMs not evaluated

According to Microsoft's CodexGLUE leader-board [5], three other PTMs can better serve our purpose: Text2Java-BigJavaDataSet, Co-Text, Text2Java-T5. But due to the lack of open-sourced implementation, we could not incorporate Text2Java-BigJavaDataSet and Text2Java-T5. We have contacted the authors of CoText [29], but we are yet to receive a response.

Narrow API domain

Our paper only incorporated one dataset from a previous study on Deep-API Learning, which only examined APIs and related projects from the JDK API. Consequently, the dataset may lack generalizability and fair representation of other libraries and programming languages. But it can very well work as a reference to show

---

[5]https://microsoft.github.io/CodexGLUE/

the potential of a designed approach for guiding the researchers and practitioners.

Annotation quality

Annotations of API sequences are collected from the first sentence of comments in the documentation, which may contain noise. The dataset curators did not consider the fact that some other sentences may also provide useful information. In the future, we plan to curate a more comprehensive dataset that will include more diverse programming languages and more informative annotations.

Size of the PTMs

How large should the PTM be to start seeing improvements over non-PTM approaches is not studied in our work but this is a great research question and we plan to investigate this for our future work. We need to vary the size of the PTM and investigate the performance for each size change. As the experiment will require a substantial amount of time, we plan to integrate this in our future work.

Data Leakage

We have not checked if the *<Natural Language Annotation, API Sequence>* test set were ever a part of the considered PTMs' pre-training. We only ensured that no test data leaked into the training dataset that we used to fine-tune the PTMs by cross-checking the *<Natural Language Annotation, API Sequence>* pairs. The involved PTMs have utilized different code sources collected over different timelines. Although all the papers that proposed the PTMs explicitly mention the sources, these repositories were not made readily available. As we need to track, collect, and process these repositories by going through the research papers and related documentation, this task will take a lot of time to implement and cross-check. Moreover, even if some codes or natural language comments associated with our training dataset were present during the pretraining of these PTMs, these codes and comments were used to achieve a different objective task that focused on learning something different from API learning.

Imbalanced Training and Testing Data

A possible internal validity can be related to the training and testing split in the dataset. We randomly selected 10,000 <API sequence, annotation> out of ~500,000 pairs for testing and 7M for training and validation data. Due to time and resource constraints, we did not use stratified k-fold cross-validation to avoid the bias that might be introduced to the results by the test set. In addition, to mitigate threats to the validity of the results, we kept hyper-parameter values the same for all PTMs in all the steps of experiments. We ran all the experiments on a single machine, and we reported the machine configuration to enforce the reproducibility of the results. Furthermore, we consider the same metrics to compare the PTMs with prior approaches.

Evaluation metric

We relied on a single evaluation metric, BLEU Score. Although this evaluation metric has been heavily used in literature to evaluate machine translation, we could incorporate an evaluation metric based on manual human validation of the translation tasks. In the future, we will report different variations of BLEU scores and human evaluation of API Sequence generation task.

Construct Validity

The selection of the prior approaches and PTMs can pose a validity threat to our study. We identified the three most common approaches as priors by examining the highly practiced methods, tools, and techniques employed by researchers and application developers. We chose these methods as PRIORs after conducting a comprehensive literature review. The PTMs were adopted by following previous studies that performed Code to Text conversion-related tasks with PTMs.

6 RELATED WORK

Our work is closely related with code generation and code retrieval tasks. We describe some related work as follows.

Deep Learning-based Code Generation.

Various deep learning-based approaches have been proposed to address code generation tasks. Ling et. al. [20] proposed Latent Predictor Network (LPN) to translate the natural language to Java and Python languages. LPN represent the output as code tokens at the character granularity and applies copy-mechanism to deal with unknown tokens. A number of studies improve the performance of LPN by incorporating code structural information. Some approaches that belong to this category are Abstract Syntax Network (ASN) [34] and Syntactic Neural Model (SNM) [51]. Both approaches represent the output as ASTs instead of sequences of code tokens. These two approaches can perform better than LPN because it leverages the grammar of the AST to constraint the search space of the output. Patois [37] adopts Syntactic Neural Model [51] and leverages pattern mining technique to improve its performance. The idea is to generate a frequent pattern (a common code fragment) at one time step instead of generating it token by token in several time steps. Moreover, Sun et. al. [41] propose a transformer-based architecture specialized for code generation task called TreeGen.

Deep Learning-based Code Search.

Some approaches have been proposed to address code search task. Gu et. al. [12] proposes DeepCS, a deep-learning approach that maps natural language and code snippets into a shared vector space. The main idea of DeepCS is by training the network to map a pair of relevant natural language descriptions and its code snippet such that both are mapped close to each other in the vector space. There are various similar approaches that are proposed with the same underlying idea, such as UNIF[3], MMAN [11], CDRL [15], CQIL [18], and CARLCS [39]. The main difference between those approaches are the code features (e.g., sequence of code tokens, method names, method signature) used to train the network and the network architecture.

PTM for Sequence Generation.

[36] proposed a novel Chinese Pre-trained Unbalanced Transformer (CPT) that is designed for both natural language understanding (NLU) and natural language generation (NLG) tasks. It consists of three parts: a shared encoder, an understanding decoder, and a generation decoder. Two specific decoders with a shared encoder are pre-trained with masked language modeling (MLM) and denoising auto-encoding (DAE) tasks, respectively. CPT greatly

accelerates the inference of text generation. [4] tried to help app developers generate answers that are related to the users' issues. The app response generation models use deep neural networks and train on large amount of user feedbacks for training. [8] examined different strategies to integrate pre-trained representations into sequence to sequence models and apply it to neural machine translation and abstractive summarization. The study find that pre-trained representations are most effective when added to the encoder network which slows inference by only 14[40] get inspired by the success of BERT and proposed MAsked Sequence to Sequence pre-training (MASS) for the encoder-decoder based language generation tasks. It adopts the encoder-decoder framework to reconstruct a sentence fragment given the remaining part of the sentence: its encoder takes a sentence with randomly masked fragment as input, and its decoder tries to predict this masked fragment so that it can jointly train the encoder and decoder to develop the capability of representation extraction and language modeling. [5] focused on transferring supervision signals of natural language generation (NLG) tasks between multiple languages. The study proposed to pretrain the encoder and the decoder of a sequence-to-sequence model under both monolingual and cross-lingual settings. The pre-training objective encourages the model to represent different languages in the shared space, so that it can conduct zero-shot cross-lingual transfer. [17] is a denoising autoencoder for pretraining sequence-to-sequence models, which is trained by corrupting text with an arbitrary noising function, and learning a model to reconstruct the original text. It uses a standard Tranformer-based neural machine translation architecture. [48] proposed an enhanced multi-flow sequence to sequence pre-training and fine-tuning framework named ERNIE-GEN, which bridges the discrepancy between training and inference with an infilling generation mechanism and a noise-aware generation method. To make generation closer to human writing patterns, this framework introduces a span-by-span generation flow that trains the model to predict semantically-complete spans consecutively rather than predicting word by word. [31] trained GANs for language generation that has proven to be more difficult, because of the non-differentiable nature of generating text with recurrent neural networks. This study showed that recurrent neural networks can be trained to generate text with GANs from scratch using curriculum learning, by slowly teaching the model to generate sequences of increasing and variable length. [25] introduced DeltaLM, a pretrained multilingual encoder-decoder model that regards the decoder as the task layer of off-the-shelf pretrained encoders. Specifically, it augment the pretrained multilingual encoder with a decoder and pre-train it in a self-supervised way. To take advantage of both the large-scale monolingual data and bilingual data, DeltaLM adopts the span corruption and translation span corruption as the pre-training tasks.

PTM for Code Generation and Understanding

[38] categorized approaches to generating source code from natural language descriptions by their input and output forms. The study also suggested the future direction of this research domain to improve automatic code generation using natural language by analyzing the current trend of approaches. [30] proposes Synchromesh, a framework to improve the coding reliability of pre-trained models. Using Target Similarity Tuning, this framework retrieves a few-shot example from a training bank. The second stage involves feeding examples to a pre-trained language model, which samples programs based on Constrained Semantic Decoding. [19] developed AlphaCode, a system for generating code that can create novel solutions to problems that require deeper reasoning. [28] explores the possibility of applying similar techniques to a highly structured environment with strict syntax rules. Based on pre-trained language models, the study proposes an end-to-end machine learning model for code generation in Python. [42] has introduced IntelliCode Compose - a general-purpose multilingual code completion tool that can generate up to entire lines of syntactically correct code from sequences of code tokens of arbitrary types. 1.2 billion lines of Python, C#, JavaScript, and TypeScript source code were used to train a state-of-the-art generative transformer model. [50] studied the effectiveness of incorporating two forms of external knowledge: automatically mined NL-code pairs from the programming QA forums Stack Overflow and API documentation of programming languages. [32] presented ProphetNet, a pre-training-based method for generating English text summaries and questions. ProphetNet was also extended into other domains and languages to create the ProphetNet family pre-training models, named ProphetNet-X, where X can be English, Chinese, Multi-lingual, and so on. Researchers have pretrained ProphetNet-Multi, ProphetNet-Zh, ProphetNet-Dialog-En, and ProphetNet-Dialog-Zh cross-lingual dialog generation models.

7 CONCLUSION AND FUTURE WORK

In this paper, we apply Pre-trained Transformer based Models (PTM) for constructing API usage sequences for an API-related natural language query. According to our empirical study, PTMs perform better, beat the state-of-the-art approaches, and effectively produce API sequences based on natural language queries. Although PTMs have shown promise in Natural Language and Programming Language translation tasks, we are the first to demonstrate their effectiveness in API learning. Additionally, we investigated different tokenization methods that improved the performance of PTMs. Other software engineering problems such as code search and fault localization in API sequence may benefit from our findings in this paper.

For future work, we will empirically study more PTMs and determine their best performance for API learning under different custom hyper-parameters tuning. We will incorporate more libraries from different programming languages to ensure a more fair and generalized dataset. Also, we can adopt a more sophisticated approach to generate annotation than just focusing on the first line of method-level comments. We will also explore the applications of PTMs to code search and fault localization in API sequence problems and investigate the synthesis of sample code from API sequences.

ACKNOWLEDGMENTS

REFERENCES

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. *CoRR* abs/2103.06333 (2021). arXiv:2103.06333 https://arxiv.org/abs/2103.06333

[2] Peter F. Brown, Stephen A. Della Pietra, Vincent J. Della Pietra, and Robert L. Mercer. 1993. The Mathematics of Statistical Machine Translation: Parameter Estimation. *Computational Linguistics* 19, 2 (1993), 263–311. https://aclanthology.org/J93-2003

[3] José Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When deep learning met code search. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 964–974. https://doi.org/10.1145/3338906.3340458

[4] Yue Cao and Fatemeh H. Fard. 2021. Pre-Trained Neural Language Models for Automatic Mobile App User Feedback Answer Generation. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*. 120–125. https://doi.org/10.1109/ASEW52652.2021.00033

[5] Zewen Chi, Li Dong, Furu Wei, Wenhui Wang, Xian-Ling Mao, and Heyan Huang. 2020. Cross-Lingual Natural Language Generation via Pre-Training. 34 (Apr. 2020), 7570–7577. https://doi.org/10.1609/aaai.v34i05.62

[6] Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. 2020. Electra: Pre-training text encoders as discriminators rather than generators. *arXiv preprint arXiv:2003.10555* (2020).

[7] 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).

[8] Sergey Edunov, Alexei Baevski, and Michael Auli. 2019. Pre-trained Language Model Representations for Language Generation. https://doi.org/10.48550/ARXIV.1903.09722

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

[10] Jaroslav Fowkes and Charles Sutton. 2016. Parameter-Free Probabilistic API Mining across GitHub. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 254–265. https://doi.org/10.1145/2950290.2950319

[11] Jian Gu, Zimin Chen, and Martin Monperrus. 2021. Multimodal Representation for Neural Code Search. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2021, Luxembourg, September 27 - October 1, 2021*. IEEE, 483–494. https://doi.org/10.1109/ICSME52107.2021.00049

[12] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 933–944. https://doi.org/10.1145/3180155.3180167

[13] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API Learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 631–642. https://doi.org/10.1145/2950290.2950334

[14] Mohammad Abdul Hadi and Fatemeh Hendijani Fard. 2021. Evaluating Pre-Trained Models for User Feedback Analysis in Software Engineering: A Study on Classification of App-Reviews. *ArXiv* abs/2104.05861 (2021).

[15] Qing Huang, An Qiu, Maosheng Zhong, and Yuan Wang. 2020. A Code-Description Representation Learning Model Based on Attention. In *27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020*, Kostas Kontogiannis, Foutse Khomh, Alexander Chatzigeorgiou, Marios-Eleftherios Fokaefs, and Minghui Zhou (Eds.). IEEE, 447–455. https://doi.org/10.1109/SANER48275.2020.9054830

[16] Jinhan Kim, Sanghoon Lee, Seung-won Hwang, and Sunghun Kim. 2010. Towards an Intelligent Code Search Engine *(AAAI'10)*. AAAI Press, 1358–1363.

[17] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. 2019. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. https://doi.org/10.48550/ARXIV.1910.13461

[18] Wei Li, Haozhe Qin, Shuhan Yan, Beijun Shen, and Yuting Chen. 2020. Learning Code-Query Interaction for Enhancing Code Searches. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2020, Adelaide, Australia, September 28 - October 2, 2020*. IEEE, 115–126. https://doi.org/10.1109/ICSME46990.2020.00021

[19] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-Level Code Generation with AlphaCode. https://doi.org/10.48550/ARXIV.2203.07814

[20] Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomás Kociský, Fumin Wang, and Andrew W. Senior. 2016. Latent Predictor Networks for Code Generation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics. https://doi.org/10.18653/v1/p16-1057

[21] Erik Linstead, Sushil Bajracharya, Trung Ngo, Paul Rigor, Cristina Lopes, and Pierre Baldi. 2009. Sourcerer: Mining and Searching Internet-Scale Software Repositories. *Data Min. Knowl. Discov.* 18, 2 (apr 2009), 300–336. https://doi.org/10.1007/s10618-008-0118-x

[22] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[23] Kien Luong, Mohammad Hadi, Ferdian Thung, Fatemeh Fard, and David Lo. 2021. FACOS: Finding API Relevant Contents on Stack Overflow with Semantic and Syntactic Analysis. *CoRR* abs/2111.07238 (2021). arXiv:2111.07238 https://arxiv.org/abs/2111.07238

[24] Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. 2015. CodeHow: Effective Code Search Based on API Understanding and Extended Boolean Model (E). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 260–270. https://doi.org/10.1109/ASE.2015.42

[25] Shuming Ma, Li Dong, Shaohan Huang, Dongdong Zhang, Alexandre Muzio, Saksham Singhal, Hany Hassan Awadalla, Xia Song, and Furu Wei. 2021. DeltaLM: Encoder-Decoder Pre-training for Language Generation and Translation by Augmenting Pretrained Multilingual Encoders. https://doi.org/10.48550/ARXIV.2106.13736

[26] Collin Mcmillan, Denys Poshyvanyk, Mark Grechanik, Qing Xie, and Chen Fu. 2013. Portfolio: Searching for Relevant Functions and Their Usages in Millions of Lines of Code. 22, 4, Article 37 (oct 2013), 30 pages. https://doi.org/10.1145/2522920.2522930

[27] Myle Ott, Sergey Edunov, David Grangier, and Michael Auli. 2018. Scaling Neural Machine Translation. *CoRR* abs/1806.00187 (2018). arXiv:1806.00187 http://arxiv.org/abs/1806.00187

[28] Luis Perez, Lizi Ottens, and Sudharshan Viswanathan. 2021. Automatic Code Generation using Pre-Trained Language Models. https://doi.org/10.48550/ARXIV.2102.10535

[29] Long N. Phan, Hieu Tran, Daniel Le, Hieu Nguyen, James T. Anibal, Alec Peltekian, and Yanfang Ye. 2021. CoTexT: Multi-task Learning with Code-Text Transformer. *CoRR* abs/2105.08645 (2021). arXiv:2105.08645 https://arxiv.org/abs/2105.08645

[30] Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable code generation from pre-trained language models. https://doi.org/10.48550/ARXIV.2201.11227

[31] Ofir Press, Amir Bar, Ben Bogin, Jonathan Berant, and Lior Wolf. 2017. Language Generation with Recurrent Generative Adversarial Networks without Pre-training. *CoRR* abs/1706.01399 (2017). arXiv:1706.01399 http://arxiv.org/abs/1706.01399

[32] Weizhen Qi, Yeyun Gong, Yu Yan, Can Xu, Bolun Yao, Bartuer Zhou, Biao Cheng, Daxin Jiang, Jiusheng Chen, Ruofei Zhang, Houqiang Li, and Nan Duan. 2021. ProphetNet-X: Large-Scale Pre-training Models for English, Chinese, Multilingual, Dialog, and Code Generation. https://doi.org/10.48550/ARXIV.2104.08006

[33] Xipeng Qiu, Tianxiang Sun, Yige Xu, Yunfan Shao, Ning Dai, and Xuanjing Huang. 2020. Pre-trained Models for Natural Language Processing: A Survey. *CoRR* abs/2003.08271 (2020). arXiv:2003.08271 https://arxiv.org/abs/2003.08271

[34] 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, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, Regina Barzilay and Min-Yen Kan (Eds.). Association for Computational Linguistics, 1139–1149. https://doi.org/10.18653/v1/P17-1105

[35] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. 2016. SWIM: Synthesizing What i Mean: Code Search and Idiomatic Snippet Synthesis *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 357–367. https://doi.org/10.1145/2884781.2884808

[36] Yunfan Shao, Zhichao Geng, Yitao Liu, Junqi Dai, Fei Yang, Li Zhe, Hujun Bao, and Xipeng Qiu. 2021. CPT: A Pre-Trained Unbalanced Transformer for Both Chinese Language Understanding and Generation. https://doi.org/10.48550/ARXIV.2109.05729

[37] Eui Chul Richard Shin, Miltiadis Allamanis, Marc Brockschmidt, and Alex Polozov. 2019. Program Synthesis and Semantic Parsing with Learned Code Idioms. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle,

Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 10824–10834. https://proceedings.neurips.cc/paper/2019/hash/cff34ad343b069ea6920464ad17d4bcf-Abstract.html

[38] Jiho Shin and Jaechang Nam. 2021. A Survey of Automatic Code Generation from Natural Language. *Journal of Information Processing Systems* 17, 3 (2021), 537–555.

[39] Jianhang Shuai, Ling Xu, Chao Liu, Meng Yan, Xin Xia, and Yan Lei. 2020. Improving Code Search with Co-Attentive Representation Learning. In *ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020.* ACM, 196–207. https://doi.org/10.1145/3387904.3389269

[40] Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. 2019. MASS: Masked Sequence to Sequence Pre-training for Language Generation. https://doi.org/10.48550/ARXIV.1905.02450

[41] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2020. TreeGen: A Tree-Based Transformer Architecture for Code Generation. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020.* AAAI Press, 8984–8991. https://aaai.org/ojs/index.php/AAAI/article/view/6430

[42] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. *IntelliCode Compose: Code Generation Using Transformer.* Association for Computing Machinery, New York, NY, USA, 1433–1443. https://doi.org/10.1145/3368089.3417058

[43] Vincent S Tseng, Cheng-Wei Wu, Jun-Han Lin, and Philippe Fournier-Viger. 2015. UP-Miner: A utility pattern mining toolbox. In *2015 IEEE International Conference on Data Mining Workshop (ICDMW).* IEEE, 1656–1659.

[44] 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.

[45] Shoya Wada, Toshihiro Takeda, Shiro Manabe, Shozo Konishi, Jun Kamohara, and Yasushi Matsumura. 2020. A pre-training technique to localize medical BERT and enhance BioBERT. *CoRR* abs/2005.07202 (2020). arXiv:2005.07202 https://arxiv.org/abs/2005.07202

[46] Jue Wang, Yingnong Dang, Hongyu Zhang, Kai Chen, Tao Xie, and Dongmei Zhang. 2013. Mining succinct and high-coverage API usage patterns from source code. In *2013 10th Working Conference on Mining Software Repositories (MSR).* 319–328. https://doi.org/10.1109/MSR.2013.6624045

[47] Jamaal Hay Wenpeng Yin and Dan Roth. 2019. Benchmarking Zero-shot Text Classification: Datasets, Evaluation and Entailment Approach. In *EMNLP.* https://arxiv.org/abs/1909.00161

[48] Dongling Xiao, Han Zhang, Yukun Li, Yu Sun, Hao Tian, Hua Wu, and Haifeng Wang. 2020. ERNIE-GEN: An Enhanced Multi-Flow Pre-training and Fine-tuning Framework for Natural Language Generation. https://doi.org/10.48550/ARXIV.2001.11314

[49] Tao Xie and Jian Pei. 2006. MAPO: Mining API Usages from Open Source Repositories. In *Proceedings of the 2006 International Workshop on Mining Software Repositories* (Shanghai, China) *(MSR '06).* Association for Computing Machinery, New York, NY, USA, 54–57. https://doi.org/10.1145/1137983.1137997

[50] 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. https://doi.org/10.48550/ARXIV.2004.09015

[51] 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, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers,* Regina Barzilay and Min-Yen Kan (Eds.). Association for Computational Linguistics, 440–450. https://doi.org/10.18653/v1/P17-1041

[52] Ting Zhang, Bowen Xu, Ferdian Thung, Stefanus Agus Haryono, David Lo, and Lingxiao Jiang. 2020. Sentiment Analysis for Software Engineering: How Far Can Pre-trained Transformer Models Go?. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME).* 70–80. https://doi.org/10.1109/ICSME46990.2020.00017