Diet Code Is Healthy: Simplifying Programs for Pre-trained Models of Code

Zhaowei Zhang[1], Hongyu Zhang[2], Beijun Shen[1], Xiaodong Gu[1*]
[1]School of Software, Shanghai Jiao Tong University, China
[2]The University of Newcastle, Australia
{andy_zhangzw,bjshen,xiaodong.gu}@sjtu.edu.cn,hongyu.zhang@newcastle.edu.au

ABSTRACT

Pre-trained code representation models such as CodeBERT have demonstrated superior performance in a variety of software engineering tasks, yet they are often heavy in complexity, quadratically with the length of the input sequence. Our empirical analysis of CodeBERT's attention reveals that CodeBERT pays more attention to certain types of tokens and statements such as keywords and data-relevant statements. Based on these findings, we propose DietCode, which aims at lightweight leverage of large pre-trained models for source code. DietCode simplifies the input program of CodeBERT with three strategies, namely, word dropout, frequency filtering, and an attention-based strategy that selects statements and tokens that receive the most attention weights during pre-training. Hence, it gives a substantial reduction in the computational cost without hampering the model performance. Experimental results on two downstream tasks show that DietCode provides comparable results to CodeBERT with 40% less computational cost in fine-tuning and testing.

CCS CONCEPTS

• **Computing methodologies → Natural language processing**.

KEYWORDS

Program simplification, Pre-trained models, Learning program representations, Code intelligence

ACM Reference Format:
Zhaowei Zhang[1], Hongyu Zhang[2], Beijun Shen[1], Xiaodong Gu[1]. 2022. Diet Code Is Healthy: Simplifying Programs for Pre-trained Models of Code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3540250.3549094

1 INTRODUCTION

Pre-trained models of code such as CodeBERT [15] have been the cutting-edge program representation technology, yielding remarkable performance on a variety of software engineering tasks such as code completion [11], code search [15], and clone detection [29]. Having pre-trained on large-scale code corpora, they demonstrate a better understanding of the semantics of source code than previous deep learning models such as code2vec [3] and ASTNN [50].

Despite making a giant leap in accuracy, pre-trained models are often heavy in computation, which significantly hinders their applications in practice. For example, the standard CodeBERT contains 125 million parameters and takes 28 hours to pre-train on 8.5M code. More critically, pre-trained models usually need to be fine-tuned before use, which is cost inefficient due to the large scale of parameters and training data. As such, it is highly desirable to identify the critical feature learned by pre-trained models and reduce the required computational cost by focusing only on the important information from model inputs [34].

To understand critical information learned by pre-trained models, we conduct an empirical analysis of CodeBERT – a pre-trained model for programming and natural languages. Our study aims to find out (i) what kinds of tokens CodeBERT pays the most attention to; and (ii) what kinds of statements are most important to CodeBERT when learning code representations. To answer these two questions, we categorize tokens and statements into a few classes and summarize the attention weights of each class that the pre-trained CodeBERT assigns to. Our results reveal that keywords and data types are the most critical tokens that CodeBERT focuses on. In terms of statements, CodeBERT pays more attention to *method signatures* and *return statements*, which show the overall functionality of a method.

Based on these empirical findings, we propose DietCode, a novel method that aims at lightweight leverage of large pre-trained models for source code. DietCode reduces the computational complexity of pre-trained models by pruning unimportant tokens and statements from the input programs. Three pruning strategies are proposed, including word dropout, frequency filtering, and attention-based pruning. In particular, the attention-based pruning strategy selects tokens and statements that receive the highest attention from pre-trained models. The program simplification algorithm formulates statement selection as a 0-1 knapsack problem where statements are regarded as items, and their attention weights are regarded as values. The algorithm selects statements (items) under the constraint of a given target length (capacity).

We apply DietCode to two downstream tasks, namely, code search and code summarization. We measure the performance with relative length, FLOPs, and time cost, and compare our approach with baseline models, including the original pre-trained code model and SIVAND [34]. Experimental results show that DietCode provides comparable results as RoBERTa (code), with nearly 40% less computational cost in fine-tuning and testing.

*Xiaodong Gu is the corresponding author.

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9413-0/22/11…$15.00
https://doi.org/10.1145/3540250.3549094

Our contributions can be summarized as follows:

- We conduct an in-depth empirical analysis of critical tokens and statements learned by CodeBERT.
- We propose a novel program simplification approach for pre-trained programming language models that can significantly reduce the computation cost while retaining comparable performance.
- We extensively evaluate the proposed approach in two downstream tasks and show the effectiveness of our approach.

2 BACKGROUND

2.1 Pre-Trained Language Models

Pre-trained language models such as BERT [14], GPT-2 [35], and T5 [36] have achieved remarkable success in a variety of NLP tasks [12, 13, 47]. They refer to neural network models trained on large text corpora and can be fine-tuned to low-resource downstream tasks.

State-of-the-art pre-trained models are mainly built upon the Transformer architecture [43]. The Transformer is a sequence-to-sequence learning model using the attention mechanism [4]. It is based on the encoder-decoder architecture, where a source sequence is encoded into hidden states and then taken as input to the decoder for generating a target sequence. Both the encoder and decoder contain multiple identical layers, and each layer consists of a multi-head self-attention network followed by a feed-forward network. Both of the outputs will be normalized before entering the next layer.

The key component of Transformer is the self-attention mechanism that represents a sequence by relating tokens in different positions [43]. The goal of self-attention is to learn the important regions in the input sequence. Unlike traditional recurrent neural networks [10], self-attention networks can learn the dependencies from distant tokens in parallel. For an input sequence $(x_1, \ldots, x_n)$ of length n, the self-attention produces its representation $(z_1, z_2 \ldots z_n)$ as

$$z_i = \sum_{j=1}^{n} \text{Softmax}\left(\frac{(x_i W^Q) \cdot (x_j W^K)^T}{\sqrt{d}}\right) \cdot x_j W^V \tag{1}$$

where W^Q, W^K, and W^V denote the model's parameters. d is the dimension of the input matrix. The model involves several attention heads, and each head's output is concatenated into the final result.

Pre-trained models usually involve large-scale parameters and consume huge computational resources. For example, BERT-base and BERT-large contain 110M and 340M parameters, respectively. As such, the lightweight leverage of pre-trained models is greatly desirable for research and practitioners [39].

2.2 CodeBERT

Recently, researchers have adopted BERT for software engineering tasks and proposed CodeBERT [15]. CodeBERT is a bimodal pre-trained model for natural and programming languages, capturing semantic representations from programming languages [15]. The program representations learned by CodeBERT can be further utilized for downstream tasks such as code search and code summarization.

CodeBERT is built on top of a Transformer encoder [43]. The optimization involves two tasks: masked language modeling (MLM) and replaced token detection (RTD). MLM masks two random tokens from the input pair of code and natural language comments, and aims to predict the original token in an extensive vocabulary. RTD involves two generators and a discriminator. The generators predict the original token for the masked token while the discriminator predicts whether the tokens are original or not. After pre-training, CodeBERT can be adapted to downstream tasks through fine-tuning on the target dataset.

3 EMPIRICAL ANALYSIS

3.1 Study Design

In this section, we describe our study methodology and experimental setup. There are many levels of granularity for code, such as tokens, statements, and functions. As an in-depth study, we begin by investigating the atomic unit of source code, namely, tokens. We next investigate the statement-level knowledge learned by CodeBERT, which contains basic structures and semantic units. We finally explore the function-level knowledge learned by CodeBERT through downstream tasks. In summary, we design our study methodology by addressing the following research questions:

- **RQ1: What critical tokens does CodeBERT learn about?** We study the critical information CodeBERT learned at the token level by analyzing the attention weights assigned to these tokens and visualizing their relative importance.
- **RQ2: What critical statements does CodeBERT learn about?** We further study the statements that CodeBERT assigns the most weights to. We classify code statements into common categories such as *initialization*, *assignment*, and *return*, and present the attention weight of each category assigned by CodeBERT.

To answer these questions, we first need to know how to represent the key information learned by CodeBERT. In other words, how to measure the importance of each token and statement? Since the heart of CodeBERT is the self-attention network, where hidden states of each token are calculated layer-by-layer according to the self-attention weights, we measure the importance of each token using the attention weights in the Transformer layers in CodeBERT after pre-training. Next, we will present the details of how to measure the importance of tokens and statements, respectively.

3.1.1 *Measuring Token Importance Using Attention Weights.*

As introduced in Section 2.2, CodeBERT takes a sequence of source code tokens as input and produces a self-attention weight for each input token. Each weight measures how the corresponding token gains attention from other tokens in the input sequence. The higher the attention weights, the more attention that are paid by other tokens. Therefore, in our study, we measure the importance of each token using the attention weight. CodeBERT has multiple self-attention layers and heads, each producing an attention weight for the same token. We average the attention weights of all layers and heads for each token. In our experiments, we go through the CodeSearchNet corpus [20] and take as input each code snippet to the pre-trained CodeBERT. Then, we calculate the average attention weight that CodeBERT assigns to each token.

Table 1: Statistics of statements. *Arithmetic* means statements with only mathematical operations. *Function Invocation* represents statements that invoke other functions.

Category	Quantity	Category	Quantity
Function Invocation	16,558	Throw	1,460
Method Signature	11,755	Catch	1,309
Variable Declaration	11,701	Arithmetic	628
If Condition	11,646	Case	577
Annotation	8,980	While	459
Return	8,331	Break	341
Getter	3,092	Finally	297
For	2,190	Continue	142
Try	1,797	Switch	210
Logging	1,763	Synchronized	73
Setter	1,721		

3.1.2 Computing Attentions of Statements. Having obtained the importance of each token, we compute the attention weight for each statement. Intuitively, the attention weight for a statement can simply be obtained using the average attention weights of all its tokens. However, different tokens in a statement have different importance. Without considering the global importance of each token, statements that consist of unimportant tokens could obtain even higher attention. Based on this concern, we regularize statement attention by penalizing unimportant tokens that gain fewer attention weights across the whole corpus. More specifically, we calculate the attention weight for a statement S using a weighted average of attention weights of its tokens:

$$a(S) = \sum_{t \in S} w(t) \cdot a(t) \tag{2}$$

where $a(t)$ represents the attention weight of token t in statement S; $w(t)$ denotes the normalized attention weight of token t in the whole corpus described in the previous section. The normalization is implemented using the Softmax function, namely,

$$w(t) = \text{Softmax}_{t \in S}(a(t)) \tag{3}$$

Unlike tokens, statements are often unique in the corpus and cannot be included in a dictionary. To efficiently analyze statements, we classify statements into 21 categories such as *method signature*, *variable declaration*, and *if condition*, and only present the average attention weight for each category. These categories are mainly guided by Java specification [16]. In particular, the *logger* category contains statements with standard logging functionalities such as `log4j`, `Logger`, and `println`. Table 1 shows the categories of statements we have summarized and the corresponding quantities in the CodeSearchNet corpus. We can see that *function invocation* and *method signature* are the most common statements, while *loop* statements such as *while* are relatively rare.

3.2 Results and Analysis

In this section, we present the results of our experiments for each research question.

3.2.1 What does CodeBERT learn about code tokens? (RQ1). Figure 2 highlights the critical tokens in Java that gain the most attention by CodeBERT. The graph is summarized from the CodeSearchNet [20] corpus which contains more than 100,000 Java functions. For tokens in each function, we calculate their attention weights and insert them into a <token, attention> map until the keys of the map are stable. For ease of visualization, we remove tokens that appear less than 50 times in the corpus. Among all the Java tokens, `coverage` gains the lowest attention weight (5.43e-5) while `boolean` has the highest weight (2.94e-2). The standard deviation of these weights is 5.97e-3.

The results show that CodeBERT assigns more attention to Java keywords such as `public` and `boolean`. This is expected since Java keywords frequently appear in Java code and play a dominant role in representing code semantics. Another category of tokens that receives high attention is the data-oriented identifiers such as `Map`, `List`, and `String`, probably because they define key data structures in a function.

Motivated by this finding, we further study the attention of Java keywords. Figure 1 shows the attention weight of each Java keyword. As we can see, the keyword `private` obtains the highest attention weight of 1.15e-2 among all keywords, while `interface` and `transient` gain the lowest attention weights of 7.14e-4 and 9.62e-4, respectively. The standard deviation of these weights is 1.513e-3.

Among all Java keywords, method modifiers such as `public`, `private`, and `static` receive the highest attention weights We conjecture that these modifiers often denote the beginning of a method signature, which is essential to understanding the entire code. Next to these modifiers, `finally` and `return` also receive high attention weights. `Finally` usually represents the end of a function, and the code inside the `finally` block will definitely be executed. `Return` represents the method's output, which may show the method's functionality. Surprisingly, branching-related keywords such as `if` and `switch` receive lower attention, presumably because CodeBERT considers them unimportant for program understanding despite their frequent occurrence in code.

A similar pattern can be observed in Python. For example, CodeBERT prefers keywords such as `return` and `def` that represent the general structure of a function, just like `return` and `public` in Java. In particular, among the branching and loop keywords, `while` ranks higher than `if` and `for`.

Figure 3 shows a heatmap of attentions for a Java function. The code is about reading content from a file. The tokens with higher attention weights are marked in deeper colors. The heatmap shows a similar result as discussed above. `public` and `finally` receive the highest attention weights in this function, while `new` and `if` obtain the lowest attentions. Tokens that are related to the core functionality, such as `File` and `read`, have shown high attention weights. By contrast, tokens that are auxiliary to the main functionality, such as `null` and `Exception` have received much lower attention. Another interesting observation is that grammatical symbols such as } and; are considered necessary by CodeBERT probably because they mark the end of statements and blocks. Overall, the heatmap shows that CodeBERT mainly pays attention to functional tokens that reflect the overall functionality and does not care much about notional tokens about grammar and special branches.

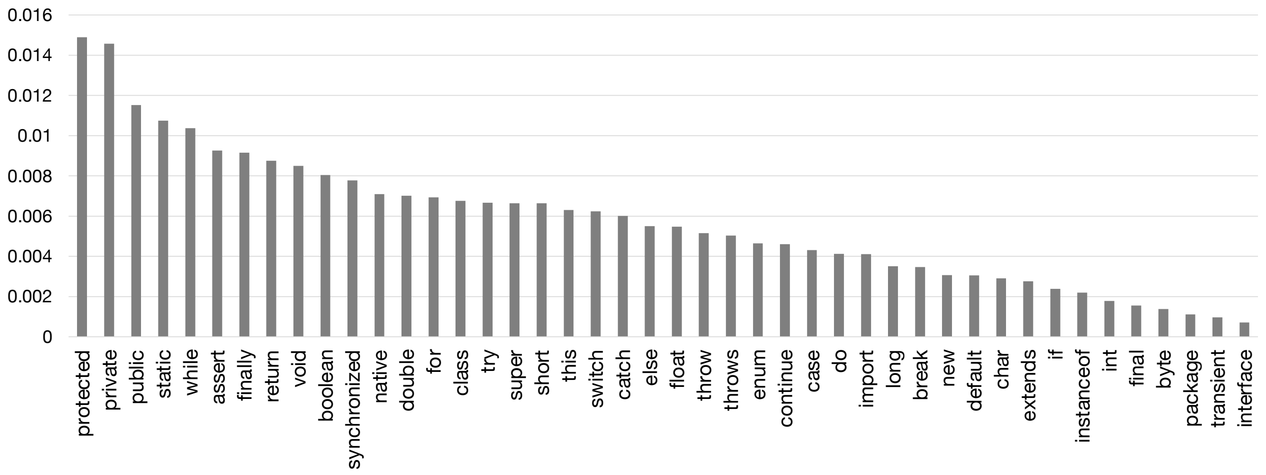


Figure 1: Attention weights of Java keywords learned by CodeBERT

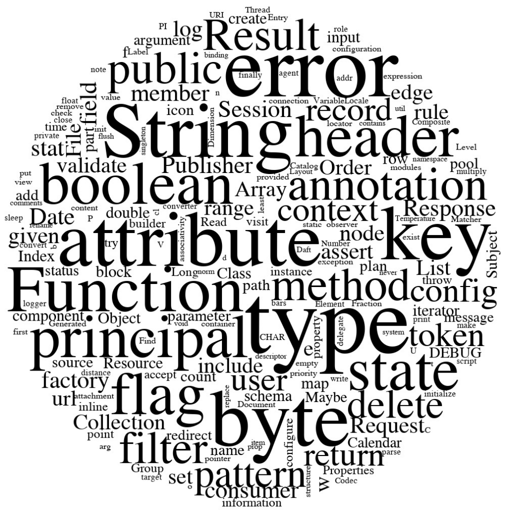


Figure 2: Java tokens highlighted by CodeBERT. The size of each token indicates the attention weight assigned by CodeBERT.

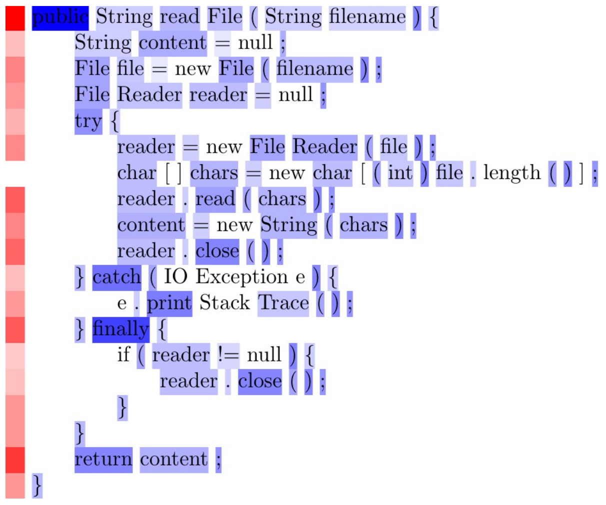

```
public String read File ( String filename ) {
    String content = null ;
    File file = new File ( filename ) ;
    File Reader reader = null ;
    try {
        reader = new File Reader ( file ) ;
        char [ ] chars = new char [ ( int ) file . length ( ) ] ;
        reader . read ( chars ) ;
        content = new String ( chars ) ;
        reader . close ( ) ;
    } catch ( IO Exception e ) {
        e . print Stack Trace ( ) ;
    } finally {
        if ( reader != null ) {
            reader . close ( ) ;
        }
    }
    return content ;
}
```

Figure 3: A heatmap of attention weights for Java statements and tokens. The background color of a token is proportional to the average attention weight assigned to it).

3.2.2 What does CodeBERT learn about code statements? (RQ2). Figure 4 shows the attention weights assigned by CodeBERT for each type of Java statement. As seen, *method signature* has gained the highest attention (4.11e-3), while *case statement* obtains the lowest attention (2.32e-3). The standard deviation of the attention weights over all categories is 4.88e-4. Broadly speaking, CodeBERT focuses more on statements indicating the overall functionality of a method, such as *method signature* and *return*, probably because they contain dense information of the whole function, such as names and targets.

Interestingly, *arithmetic expressions* (expressions with mathematical operations) also receive more attention than other statements. Like natural language, *arithmetic expression* can be viewed as a sequence of operations that tell the functionality of a statement. *Function invocations* has also been shown to be essential for CodeBERT to understand code. Like *arithmetic expressions*, *function invocations* can be regarded as identifiers of other method signatures, which can be literally understood through the function names and parameters.

Surprisingly, CodeBERT does not pay much attention to statements related to control flow structures, such as *while*, *for* and *case*. This indicates that CodeBERT can effectively learn representations of plain texts while being limited in learning structures.

Figure 3 also shows the heatmap of statement attentions for a sample function of Java. Similar to the conclusions above, *method signature* is the most important, which provides a general introduction to the method's functionality. At the same time, statements about *variable declaration and initialization*, such as "`String`

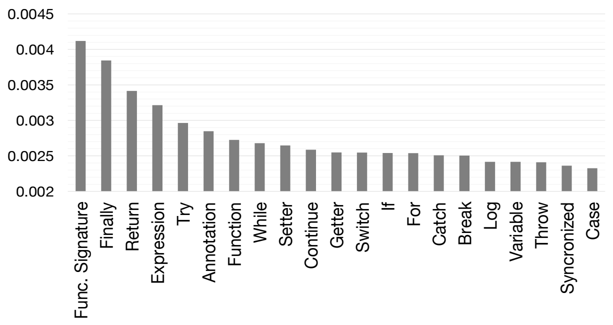


Figure 4: Attention weights of various types of Java statements learned by CodeBERT

`content == null`" receive lower attentions, probably because they only create new variables with initial value `null` without actual operations to them. The attention weight of the *if* statement in the `finally` block is also low (2.92*e*-3). This function only closes the file reader and may have little effect on CodeBERT's understanding of the whole method.

We observe a similar trend for Python statements. *Method signatures* also receive the highest attention weight (6.86*e*-3), while branching statements such as *break* and *continue* have the lowest weights. The standard deviation of attention weights for Python statements is 1.66*e*-3, which is a bit larger than that of Java statements.

Compared to Java, the distributions of Python statements and tokens are a bit more sparse. For example, the *method signature* statement which ranks the first owns a much higher weight than the second category of statements (i.e., *return*). The top keyword `def` also obtains a much higher attention weight than other keywords.

4 DIETCODE: PROGRAM SIMPLIFICATION FOR CODEBERT

As Equation 1 indicates, the computational complexity for computing the attention matrix is $O(d^2n + n^2d)$, which quadratically scales with sequence length. As such, the attention operation becomes a bottleneck when applied to long sequences such as source code [25].

We wonder whether we can remove unimportant tokens or statements from the input programs of CodeBERT. In this way, the time and space costs for the pre-trained model can be substantially reduced. Based on this idea, we propose DietCode, a lightweight pre-trained model for source code. DietCode reduces the computational complexity of CodeBERT by simplifying input programs.

More specifically, our problem formulation is as follows: given a code snippet $C = \{s_1; \ldots; s_{|C|}\}$ which consists of $|C|$ statements, we want to output a pruned code snippet C_p which contains a maximum length of L tokens. The goal of program simplification is to prune as many tokens as possible without a significant impact on the accuracy of downstream tasks.

4.1 Word Dropout

One straightforward strategy for program simplification is the word dropout [21], namely, randomly dropping tokens from the input code to meet a specified sequence length. For each token $w \in S$, we define a binary variable $r_w \in \{0, 1\}$ to indicate whether the token will be kept ($r_w = 1$) or pruned ($r_w = 0$):

$$\begin{aligned} r_w &\sim \text{Bernoulli}(p) \\ C_p &= \{w | w \in C \text{ and } r_w > 0\} \end{aligned} \tag{4}$$

where $p = L/|C|$ denotes the probability of selecting a token. In addition to reducing computational complexity, word dropout has also been shown to improve the robustness of neural networks [21]. Hence, it enhances the performance of many tasks.

4.2 Frequency Filtering

A key concern with the word dropout strategy is that it can also prune important tokens randomly from the input. Hence, we introduce another frequency-based token pruning strategy that removes uncommon tokens while keeping only the most frequently occurred tokens.

Specifically, for each input code snippet $C = \{w_1, ..., w_n\}$, we keep tokens if their frequency is among the top k ($k = |C_p|$) of all tokens in the input, namely,

$$C_p = \{w | w \in C, f(w) \in \text{top}{-}k(f(w_1), ..., f(w_n)), k = |C_p|\} \tag{5}$$

where $f(w)$ denotes the frequency of w in the whole code corpora.

4.3 Attention-based Code Pruning

The frequency filtering strategy can roughly distinguish important tokens, but may only bias common tokens. According to our empirical findings above, CodeBERT pays attention to certain types of tokens and statements. Tokens that receive high attention do not coincide with common tokens. In order to provide a more fine-grained selection of important tokens and statements, we propose an attention-based code pruning strategy that selects tokens and statements based on their attention weights.

The simplification procedure is summarized in Algorithm 1. We begin by summarizing the category of statements for a given programming language according to the methodology in our empirical study. Then, we create attention dictionaries for both statements and tokens. The algorithm runs code pruning in two phases: *selecting statements* and *pruning tokens*.

In the statement selection phase, we select pivot statements that belong to a category that obtain the highest attention weights in our empirical study. Meanwhile, we want the selected statements not to exceed the length constraint (i.e., L tokens). This can be formulated as a 0-1 knapsack problem [33], where statements can be regarded as the items to be collected into a knapsack, with the attention weights being the values, and the statement lengths being the weights. The length constraint can be regarded as the capacity of the knapsack. It is worth noting that we enlarge the capacity by the maximum length of statements for tolerance of selecting one more statement so that the algorithm can further perform token pruning in the next phase.

In the token pruning phase, we greedily remove the tokens that have the lowest attention weights in the lowest weighed statements iteratively until satisfying the maximum number of tokens.

One potential problem is the different scales of statement attentions and token sizes. We find that some shorter sequences are much more likely to be chosen during our experiments even if they

Algorithm 1 The Attention-based Pruning Algorithm

```
Require:
 1: $C=s_1;...;s_{|C|}$: an input code snippet,
 2: $A^T$: attention dictionary of tokens,
 3: $A^S$: attention dictionary of statements,
 4: L: target length
Ensure:
 5: Reduced code snippet $C_p = s'_1;...;s'_{|C_p|}$
 6: for s ∈ C do
 7:     a(s) = $A^S$(s)                          ▷ obtain the statement attention
 8:     for t ∈ s do
 9:         a(t) = $A^T$(t)                      ▷ obtain the token attention
10:     end for
11: end for
12: $C_0$ ← 0-1 Knapsack (items=$\{s\}_{s\in C}$, values=$\{a(s)\}_{s\in C}$,
    weights= $\{|s|\}_{s\in C}$, capacity = $L+\max_{s\in C}(|s|)$    ▷ candidate
    statements
13: for i = 1,..., $\sum_{s\in C_0}(|s|)$ - L do
14:     $s' \leftarrow s' \setminus \{t\}$   where $s' = \mathrm{argmin}_{s\in C_0} a(s)$,
                                 $t = \mathrm{argmin}_{t_0\in s'} a(t_0)$
15: end for
16: $C_p \leftarrow s'_1;...;s'_{|C_0|}$   where $C_0 = \{s'_1, s'_2, ..., s'_{|C_0|}\}$ ▷ concatenate
    statements
```

receive low attention weights. This is probably because the value ratio (the attention weight divided by the sequence length) of a shorter sequence is much larger than the longer one. We amplify the attention weights using min-max normalization and multiply with the number of tokens in the statement, that is,

$$a(s) = \frac{a(s) - \min_{s\in S}(a(s))}{\max_{s\in S}(a(s)) - \min_{s\in S}(a(s))} \times N \quad (6)$$

where N denotes the number of tokens in the statement.

The time complexity of the simplification algorithm is $O(N \cdot (L_T + L_M))$, where L_T denotes the target number of tokens, and L_M denotes the maximum length of statements. The time cost mainly comes from the 0-1 knapsack algorithm. Furthermore, the knapsack algorithm requires a two-dimensional array for dynamic programming, which requires space complexity.

Figure 5 shows an example of program simplification by DietCode. The original code (Figure 5(a)) aims to read content from a file with a given filename. Our goal is to reduce the code to contain only 60 tokens without considerably losing the semantics. In the *statement selection* phase (Figure 5(b)), the simplification algorithm solves a 0-1 knapsack problem by setting the capacity to 78 (including the target length of 60 and the longest statement length 18). Then, DietCode selects trunk statements such as *method signature* and *return*, and removes trivial statements such as *variable assignments* and *catch*. The number of tokens in the code is reduced from 118 to 77. In the *token pruning* phase, DietCode further removes tokens such as variable initialization (Figure 5(c)), and reduces the number of tokens from 77 to 60. Although reduced by around 50%, the main semantic of this function is not destroyed significantly. CodeBERT can still understand the function through key statements such as the method name `readFile`, the key variable `content`, and the invoked function `read`. Although the remaining code is never runnable, the selected critical information can be used for downstream tasks such as code search and summarization.

Table 2: Statistics of Datasets

Corpus	Training	Test
CodeSearchNet (Java)	908,886	1,000,000
CodeSearchNet (Python)	824,343	1,000,000

5 EVALUATION

This section introduces the evaluation of DietCode in two downstream tasks. We aim to answer the following research questions through experiments:

- **RQ3: How effective is DietCode in program simplification?** We evaluate the performance and computation cost of DietCode in two downstream tasks and compare it with baseline models.
- **RQ4: How effective is DietCode under different relative lengths?** We study the effect of different relative lengths on the evaluation scores. Our goal is to find the relative length that leads to the best trade-off between model performance and computational expense.
- **RQ5: What is the effect of different pruning strategies?** The attention-based DietCode performs both statement and token pruning. We conduct an ablation study to identify the effect of pruning each of them.

5.1 Downstream Tasks

We test our algorithm in two downstream tasks, namely, code search and code summarization. They are the most widely used software engineering tasks to demonstrate the capacity of NL-PL understanding [17, 22, 23, 31, 41, 46].

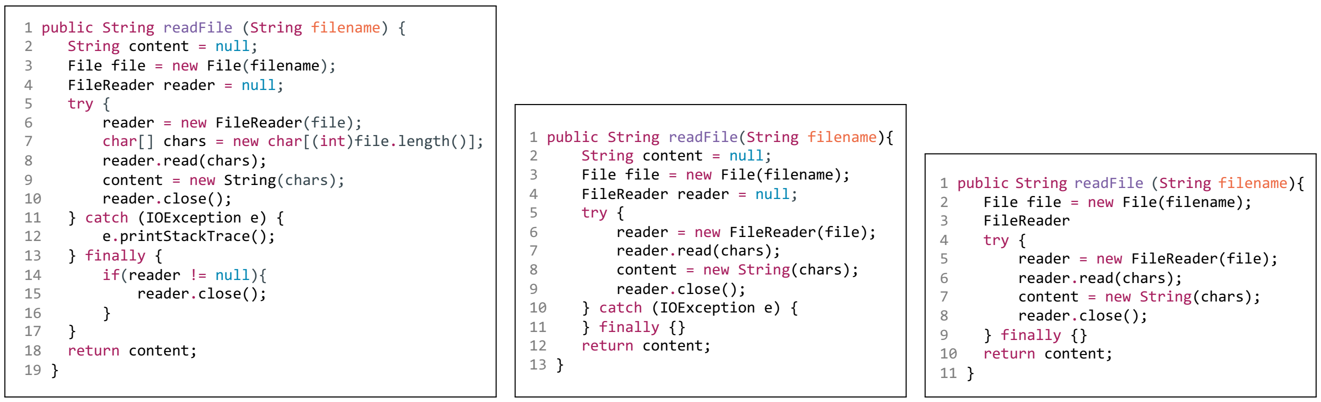
Code Search. Code search is a typical testbed for pre-trained code models such as CodeBERT [15]. The task aims to find code snippets from a codebase that is most relevant to a given query.

Code Summarization. Code summarization aims to generate a natural language summary for an input code snippet. It has also been widely used for evaluating pre-trained models for source code [15, 45]. Through this task, we want to verify whether program simplification by DietCode is effective in generation tasks.

5.2 Datasets

We use the CodeSearchNet to fine-tune and test our model [20], which contains <comment, code> pairs collected from open source projects. *code* means the code snippet of a method, while *comment* is the description of the code that was mainly collected from comments for the whole functions such as Javadocs or docstrings in Python. The statistics of CodeSearchNet is shown in Table 2.

The original data of CodeSearchNet are in the format of token sequences. We preprocess the data by parsing them into statements. We parse statements by splitting brackets and semicolons according to the common guidance of Java [16]. For Python code, there are no delimiters because the indentation is missing in the dataset. Besides, Python does not use brackets and semicolons. Hence, we



```java
public String readFile (String filename) {
   String content = null;
   File file = new File(filename);
   FileReader reader = null;
   try {
       reader = new FileReader(file);
       char[] chars = new char[(int)file.length()];
       reader.read(chars);
       content = new String(chars);
       reader.close();
   } catch (IOException e) {
       e.printStackTrace();
   } finally {
       if(reader != null){
           reader.close();
       }
   }
   return content;
}
```

(a) The original source code

```java
public String readFile(String filename){
    String content = null;
    File file = new File(filename);
    FileReader reader = null;
    try {
        reader = new FileReader(file);
        reader.read(chars);
        content = new String(chars);
        reader.close();
    } catch (IOException e) {
    } finally {}
    return content;
}
```

(b) Code after statement selection

```java
public String readFile (String filename){
   File file = new File(filename);
   FileReader
   try {
       reader = new FileReader(file);
       reader.read(chars);
       content = new String(chars);
       reader.close();
   } finally {}
   return content;
}
```

(c) Code after token pruning

Figure 5: An example of program simplification by DietCode (attention).

split statements using other hint symbols such as "def", "=", ":", and ")".

To reduce the vocabulary size of code tokens, we delexicalize all string constants to a generic token `string` and all numerical constants to the same token `10`. We conserve special numbers such as 0, 1, and -1, which can belong to boolean values.

5.3 Evaluation Metrics

First, we want to measure the maximum extent of code reduction that DietCode can achieve without losing much accuracy. We define **Relative Length (RL)** to measure how much of the code is remained after simplification. It is computed as the percentage of the length of the simplified code snippet $|C_p|$ with respect to the length of the original code $|C|$:

$$RL = \frac{|C_p|}{|C|} \times 100\% \tag{7}$$

where $|\cdot|$ denotes the length of a code snippet (i.e., number of tokens). The smaller the relative length, the greater the simplification to the original code.

We also use **FLOPs** (floating point operations) [19] to measure the effect of model reduction. FLOPs is a widely used metric to measure the complexity of a machine learning model. The higher the FLOPs, the slower the processing of the model.

The third metric is the time cost, including **fine-tuning time (FT Time) and testing time**. We calculate the execution time of DietCode for fine-tuning and testing, measured by the number of hours from the program starting to the termination.

Besides the metrics for complexity, we use two standard metrics to evaluate the effectiveness of DietCode in two downstream tasks respectively. We measure the performance of code search using **MRR** (mean reciprocal rank), which refers to the average of the multiplicative inverse of the position for the first correct answer for the query [20].

We measure the performance of code summarization using the **BLEU-4** (bilingual evaluation understudy) score [27], which calculates the average of n-gram precision on a couple of sequences with a penalty for short sequences.

5.4 Experimental Setup

The demonstrate the strength of our approach, we simplify the input programs for pre-training based models and compare the performance to that of the original pre-trained model. Specifically, we compare our technique to the vanilla versions of three pre-trained models:

- **CodeBERT** [15]: the original CodeBERT without code simplification. We follow the experimental setup in the CodeBERT paper.
- **CodeT5** [45]: a widely used pre-trained programming language model based on the sequence-to-sequence architecture. CodeT5 has demonstrated better performance on generative tasks [45].
- **RoBERTa** [28]: a popular extension of BERT that has demonstrated significant improvement. The original RoBERTa was pre-trained in natural languages. So we also report the results of RoBERTa (code) [15], a variant that was pre-trained on source code.

Besides the pre-trained models, we also compare the accuracy of DietCode with that of traditional deep learning approaches, including **BiRNN** [10], **SelfAttn** [43], **Seq2Seq** [10], and **Transformer** [43]. As they are also baseline models for CodeBERT, we directly take the results from the CodeBERT paper [15].

Finally, we compare DietCode with **SIVAND** [34], which is also a program simplification method proposed by Rabin et al. [34]. SIVAND [1] is a machine-learning-based algorithm that recursively splits source code into fragments and measures its importance through the method name prediction task. The algorithm selects important fragments as the reduced code. In our experiments, we directly use the code provided by the authors.

We implement our model based on the GitHub repository of CodeBERT [2] and CodeT5 [3] using the default hyperparameter settings. All models are optimized with the Adam algorithm [26] with

[1]https://github.com/mdrafiqulrabin/SIVAND
[2]https://github.com/microsoft/CodeBERT
[3]https://github.com/salesforce/CodeT5

Table 3: Performance of various code simplification methods on the code search task (L_{in}=input length; RL=relative length; FT=fine-tuning).

	Java						Python					
Model	L_{in}	RL	FT Time	Test Time	FLOPs	MRR	L_{in}	RL	FT Time	Test Time	FLOPs	MRR
BiRNN [10]	200	100%	9.88h	1.13h	8.34G	0.29	200	100%	7.07h	1.25h	8.34G	0.32
SelfAttn [43]	200	100%	22.83h	4.00h	16.99G	0.59	200	100%	17.78h	2.67h	16.99G	0.69
RoBERTa [28]	200	100%	23.85h	4.78h	16.99G	0.67	200	100%	19.32h	3.53h	16.99G	0.81
RoBERTa (code)	200	100%	21.02h	4.53h	16.99G	0.72	200	100%	20.78h	4.07h	16.99G	0.84
CodeBERT [15]	200	100%	20.82h	3.17h	16.99G	0.74	200	100%	17.92h	2.82h	16.99G	0.84
DietCode												
- Attention	120	60%	11.08h	1.83h	10.19G	0.71	120	60%	9.62h	1.82h	10.19G	0.81
- Dropout	120	60%	10.73h	1.95h	10.19G	0.68	120	60%	9.33h	1.93h	10.19G	0.80
- Frequency	120	60%	10.32h	1.8h	10.19G	0.66	120	60%	8.67h	1.79h	10.19G	0.78
CodeT5 [45]	200	100%	16.83h	2.97h	16.99G	0.72	200	100%	17.61h	2.83h	16.99G	0.837
DietCode												
- Attention	120	60%	9.3h	1.74h	10.19G	0.71	120	60%	8.31h	1.81h	10.19G	0.813
- Dropout	120	60%	9.25h	1.67h	10.19G	0.68	120	60%	9.33h	1.93h	10.19G	0.799
- Frequency	120	60%	8.97h	1.58h	10.19G	0.66	120	60%	8.67h	1.79h	10.19G	0.785

Table 4: Performance of various code simplification methods on the code summarization task.

	Java						Python					
Model	L_{in}	RL	FT Time	Test Time	FLOPs	BLEU-4	L_{in}	RL	FT Time	Test Time	FLOPs	BLEU-4
Seq2Seq [10]	256	100%	6.26h	1.26h	12.750G	15.09	256	100%	4.43h	1.02h	12.750G	15.93
Transformer [43]	256	100%	13.28h	5.43h	24.328G	16.26	256	100%	9.70h	2.43h	24.328G	15.81
RoBERTa [28]	256	100%	15.42h	5.22h	24.328G	16.47	256	100%	10.63h	1.88h	24.328G	18.14
RoBERTa (code)	256	100%	14.5h	6.02h	24.328G	17.50	256	100%	10.45h	2.05h	24.328G	18.58
CodeBERT [15]	256	100%	13.82h	4.8h	24.328G	18.95	256	100%	8.32h	1.87h	24.328G	19.04
DietCode												
- Attention	150	60%	8.18h	1.62h	15.325G	17.29	150	60%	5.35h	1.16h	15.325G	17.08
- Dropout	150	60%	7.95h	1.40h	15.325G	15.63	150	60%	6.81h	1.15h	15.325G	16.04
- Frequency	150	60%	8.62h	1.33h	15.325G	16.13	150	60%	6.12h	1.42h	15.325G	16.55
CodeT5 [45]	256	100%	16.91h	4.88h	54.01G	20.46	256	100%	9.68h	1.48h	54.01G	20.37
DietCode												
- Attention	150	60%	10.25h	1.63h	37.81G	19.25	150	60%	6.57h	0.90h	37.81G	18.53
- Dropout	150	60%	9.95h	1.68h	37.81G	17.65	150	60%	6.67h	0.82h	37.81G	17.07
- Frequency	150	60%	10.23h	1.35h	37.81G	18.74	150	60%	6.33h	0.85h	37.81G	17.77

learning rates of 1e-5 and 5e-5 for the code search and code summarization tasks respectively.

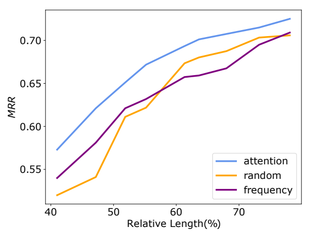
We run all models on a machine with a CPU of Intel(R) Xeon(R) Silver 4214R 2.40GHz and a GPU of Nvidia Tesla P40.

5.5 Experimental Results (RQ3)

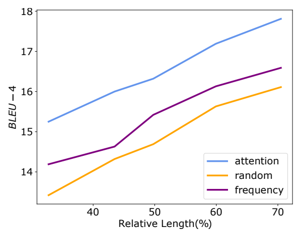
Table 3 and 4 show the performance of DietCode in two downstream tasks. Overall, DietCode reduces the computational cost by around 40% while keeping a comparable accuracy to the original pre-trained models. The original CodeBERT takes nearly one day for fine-tuning and more than 3 hours for code search on the Java test set to achieve an MRR score of 0.74. In contrast, by reducing the length of the input code to 60%, the fine-tuning and testing time can be reduced to 11.08 hours and 1.83 hours, respectively. The FLOPs also drops from 16.99G to 10.19G. At the same time, the accuracy (MRR = 0.71) is comparable to the original CodeBERT (MRR=0.74). It is worth noting that although the performance drops slightly, it still significantly outperforms traditional non-pretraining based methods such as BiRNN (MRR=0.29) and SelfAttn (MRR=0.59).

DietCode on CodeT5 demonstrates a similar efficacy. The vanilla CodeT5 requires more than 16 hours for fine-tuning and around 3 hours for testing on the Java code search task. By dropping out 40% input tokens using DietCode, the fine-tuning and test time is reduced to around 9 and 1.6 hours, respectively. Meanwhile, DietCode (attention) achieves a comparable performance (MRR=0.71) to the original CodeT5 (MRR=0.72) with only a reduction of 1.5%.

DietCode is also effective in the code summarization task. Taking Java as an example, the vanilla CodeBERT achieves a BLEU-4 score of 18.95 at the expense of 13.82 hours of fine-tuning and 4.8 hours of testing. When the target length is set to 150 tokens (i.e., 60% of the original length), the fine-tuning time is reduced to about 60% and the FLOPs drops from 24.328G to only 15.325G. This does not sacrifice accuracy (BLEU-4=17.29).



(a) results on code search



(b) results on code summarization

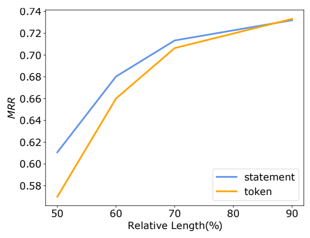
Figure 6: Performance under different relative lengths on Java.

Comparing the three pruning strategies, DietCode with attention shows strength over the two other strategies. For example, DietCode with attention-based pruning achieves an MRR of 0.71 in the Java code search task, which is better than dropout (MRR=0.68) and frequency filtering (MRR=0.66). The results suggest that by finer-grained pruning of tokens based on attention weights, DietCode can achieve more effective program simplification. DietCode on the CodeT5 demonstrates a similar trend in both tasks, where the attention-based pruning achieves better performance (BLEU=19.25) than dropout (BLEU=17.65) and frequency filtering (BLEU=18.74). We observe that this strength becomes less significant in the Python language. We conjecture that the Python language contains fewer auxiliary tokens (such as brackets), resulting in a more uniform distribution of attention weights over all tokens. Therefore, removing tokens with smaller attention is closer to removing random or frequent tokens in the Python language.

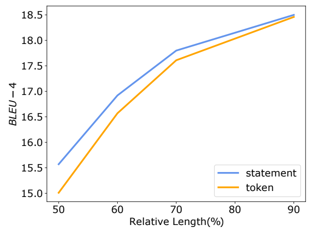
As a noteworthy point, it is inapplicable to test the baseline model SIVAND in our datasets, as it takes a massive amount of time (>30,000 hours according to our estimation) to process more than 1 million functions in the CodeSearchNet benchmarks. This is mainly because the ddmin algorithm used by SIVAND is based on backtracking, where each step the algorithm should invoke the deep learning model for a prediction task. For this reason, we estimate the time cost of SIVAND by sampling 50 functions from CodeSearchNet. Results show that it costs 104 and 77 minutes respectively in the CodeSearchNet (Java) and CodeSearchNet (Python) datasets. Based on this observation, we estimate the testing time in the entire dataset. Since the data we sample is small in size, it is insufficient to fine-tune the model to validate the accuracy.

5.6 Effect of Relative Length (RQ4)

Figure 6 shows the performance of DietCode under different relative lengths and pruning strategies. We can see that the performance of all pruning strategies drops sharply as the relative length decreases. In both two tasks, DietCode with attention performs better than the other two pruning strategies under all relative lengths. When the relative length is above 70%, the performance is comparable to that of the original CodeBERT but the reduction of computational cost can be modest. When the relative length drops below 50%, the performance is deficient since most of the code has been pruned. A relative length of around 60% is probably the best trade-off for these two tasks.



(a) results on code search



(b) results on code summarization

Figure 7: Ablation results of token and statement pruning on Java.

5.7 Effect of Token and Statement Pruning (RQ5)

Figure 7 shows the performance of DietCode with token and statement pruning on the two tasks. In order to verify the effect of statement pruning, we only run the statement selection phase in Algorithm 1. To test the effect of token pruning, we drop the lowest attention tokens directly until meeting the target length. As we can see from the results, a significant drop in performance is observed for both strategies as the relative length decreases. When 90% code remains, the MRR scores of the two pruning strategies are close. As the relative length decreases, the score of the token pruning strategy drops much faster than that of the statement pruning strategy. When the relative length decreases further, their MRR scores drop dramatically. The results suggest that statements are more critical than tokens in learning code representations by CodeBERT. One possible reason could be that token attention usually measures local information, while statement attention considers the relationship between tokens. Therefore, statement attention could keep more semantic of the functions. The code summarization task shares a similar result as the code search task.

6 DISCUSSION

6.1 How Can Pre-Trained Models Understand Simplified Code?

One debatable question is how can pre-trained models understand simplified code which is not compilable and runnable. Source code involves two channels of information: formal & natural [6, 7]. It can either be considered as programs for computers to execute or as a specific natural language for humans to communicate [18]. The former requires the source code to be strictly compilable and runnable, while the latter focuses more on conveying information to humans. We believe that CodeBERT considers more about the natural channel of source code [7, 18]. As such, it pays more attention to keywords without necessarily having to know how the program actually behaves during execution.

Our empirical findings are in agreement with a previous human study on how programmers summarize source code [37], which shows that developers usually start by reading the function name. They may skip some structural information such as `For` and `If` conditions while paying special attention to literal information such as method invocations and variable names, which literally show the intention of the code [37]. The pruning algorithms described in our paper make use of this finding and simplify source code by only selecting critical information (e.g., reflected by the attentions of CodeBERT) in the source code. Therefore, they do not have much impact on the understanding of source code.

6.2 What users may adopt the techniques?

One debatable question is what users may adopt the technique since it has a precision loss. From the results in Table 3 and 4, although the performance drops slightly, it still significantly outperforms traditional non-pretraining based methods such as BiRNN and Self-Attn. Meanwhile, DietCode shows an advantage of computational efficiency (e.g., saving 40% of fine-tuning time). Hence, DietCode provides a practical option for users with limited computational resources but still want quick leverage of large pre-trained models.

6.3 The role of the 0-1 knapsack formulation

One may question whether we really need to transform the problem into the 0-1 knapsack. In our preliminary studies, simply pruning statements often results in sequences that are either too long or too short. Token pruning provides a finer-grained pruning to the input program. This amounts to selecting statements and tokens to obtain the maximum attention while satisfying the length restriction. Hence, we formulate it as a 0-1 knapsack problem and solve it using a greedy strategy. According to the experimental results in Figure 7, a simple statement pruning archives an MRR of around 0.68 on Java code search when the relative length is 60%, which is evidently lower than that of DietCode (MRR=0.71). The same comparison can be observed on the Java code summarization task (i.e., 16.8 vs. 17.3 in terms of BLEU). This suggests the effectiveness of the 0-1 knapsack formulation.

6.4 Implications for Future Research

Through our empirical study, we find that CodeBERT does not recognize grammar and structural information very well. For example, the `if condition`, `for condition`, and `while condition` are three typical statements representing code structures, but they are rarely considered by pre-trained models when learning program representations. This motivates future research on better incorporating code structures into pre-trained code models, for example, converting structural symbols (e.g., the bracket `}`) to a more verbal style (e.g., 'the end of a for statement') before feeding into pre-trained models.

Besides DietCode, there is a broader class of techniques, including model compression [40], model distillation [39], and data reduction [48], which aim to reduce the computational complexity of large pre-trained models at the cost of a slight decrease in performance. DietCode provides an easy, alternative solution by only pruning input programs. We believe this is more practical for developers as they only need to process the data without modifying the model. In the future, we can investigate more techniques such as model compression and distillation.

6.5 Threats to Validity

Internal Threats. When calculating the attention weights for statements, we encounter the out-of-vocabulary (OOV) issue. We resolve this issue by excluding rare categories of statements in our experiments. Nevertheless, some categories of statements could still be necessary though having a low quantity in the corpus. Furthermore, the raw data provided by CodeSearchNet is in the format of plain text. We parse statements using text processing rules designed by ourselves. However, there could be irregular patterns that our parser can not recognize. This could cause noise in our dataset and affect the results of our research.

External Threats. We have just conducted our experiments in Java and Python. Though the conclusions from both languages are similar, other programming languages such as LISP and Erlang could have different attention patterns. In addition, DietCode is evaluated in two downstream tasks: code search and code summarization. However, these tasks rely much on the literal information provided in the source code. Thus, it remains to be verified whether or not the proposed simplification algorithm is applicable to other software engineering tasks.

7 RELATED WORK

7.1 Understanding Pre-trained Models of Code

Besides our work, there have been other studies that also try to explain the mechanisms of pre-trained models for code [1, 30, 32, 38]. Karmakar and Robbes [24] applied four probing tasks on pre-trained code models to investigate whether pre-trained models can learn different aspects of source code such as syntactic, structural, surface-level, and semantic information. Different from our work, they only empirically study the whole pre-trained model, whereas we focus on more specific (critical tokens and statements) knowledge learned by pre-trained models.

There have also been many works that investigate the attention weights of pre-trained models for source code. For example, Wan et al. [44] performed a structural analysis of pre-trained language models for source code. They analyzed the self-attention weights and found that the Transformer attention can capture high-level structural information of source code. AutoFocus [5] aims to reveal the most relevant part of the code to programmers. They measure the relevance of statements using attention weights from a GGNN [2]. Different from our approach, they capture structural knowledge learned by pre-trained models, while we dig more into critical statements and tokens learned by pre-trained models. Furthermore, we propose simplifying programs for pre-trained language models based on the findings.

7.2 Program Simplification

Program simplification has gained increasing attention recently. The state-of-the-art methods such as SIVAND [34] and P2IM [42] a based on the delta debugging prototype [49]. The delta debugging mechanism requires an input code snippet and an auxiliary deep learning model such as the code2vec [3]. The deep learning model takes as input the code snippet and splits it into fragments. Each fragment is then taken as input to the neural network model to perform testing tasks such as method name prediction and misused variables detection. If a fragment obtains a satisfying score, the program will split it further. The process continues until the performance of the subset does not satisfy the target score. Finally, the algorithm produces the smallest code snippet that satisfies the objective of the deep model.

Compared to DietCode, their methods are computationally inefficient because they need to run an auxiliary deep learning model and evaluate the performance at each iteration. For example, it costs hundreds of hours for SIVAND to process 10,000 functions, while DietCode can complete it within two minutes.

8 CONCLUSION

This paper empirically analyzes the critical information learned by CodeBERT, including important tokens and statements in both Java and Python. Our results show that CodeBERT focuses on keywords and data-relevant statements such as method signatures and return statements. Based on our empirical findings, we propose a novel approach named DietCode for lightweight leverage of pre-trained models. DietCode simplifies the input code to a target length by selecting important statements and tokens based on their attention weights using a 0-1 Knapsack algorithm. Experiments on two tasks have shown that DietCode provides comparable results as CodeBERT, with the advantage of computational efficiency.

In the future, we will consider more aspects that CodeBERT learns about code, such as syntax rules and semantic relations, to further reduce the size of source code for pre-trained models. We will also investigate model compression [8, 9] and distillation [39] techniques to further reduce the size of pre-trained programming language models.

Our source code and experimental data are publicly available at https://github.com/zhangzwwww/DietCode

9 ACKNOWLEDGE

This work was sponsored by NSFC No. 62102244, CCF-Tencent Open Research Fund (RAGR20220129), and CCF-Baidu Open Fund (NO. 2021PP15002000).

REFERENCES

[1] 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.

[2] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *International Conference on Learning Representations*.

[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[4] Dzmitry Bahdanau, Kyung Hyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*.

[5] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2019. Autofocus: interpreting attention-based neural networks by code perturbation. In *Proceedings of 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 38–41.

[6] Casey Casalnuovo, E Morgan, and P Devanbu. 2020. Does surprisal predict code comprehension difficulty. In *Proceedings of the 42nd Annual Meeting of the Cognitive Science Society*. Cognitive Science Society Toronto, Canada.

[7] Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar Devanbu, and Baishakhi Ray. 2020. NatGen: Generative pre-training by" Naturalizing" source code. In *Proceedings of the 30th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[8] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. 2017. A survey of model compression and acceleration for deep neural networks. *arXiv preprint arXiv:1710.09282* (2017).

[9] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. 2018. Model compression and acceleration for deep neural networks: The principles, progress, and challenges. *IEEE Signal Processing Magazine* 35, 1 (2018), 126–136.

[10] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

[11] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. 2021. An empirical study on the usage of BERT models for code completion. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 108–119.

[12] Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. 2019. ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators. In *International Conference on Learning Representations*.

[13] Alexis Conneau and Guillaume Lample. 2019. Cross-lingual language model pretraining. *Advances in neural information processing systems* 32 (2019).

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. 4171–4186.

[15] 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. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP): Findings*. 1536–1547.

[16] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. 2000. *The Java language specification*. Addison-Wesley Professional.

[17] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. 2010. Supporting program comprehension with source code summarization. In *Proceedings of ACM/IEEE 32nd international conference on software engineering (ICSE)*, Vol. 2. IEEE, 223–226.

[18] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 837–847.

[19] Raphael Hunger. 2005. *Floating point operations in matrix-vector calculus*. Munich University of Technology, Inst. for Circuit Theory and Signal.

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

[21] Mohit Iyyer, Varun Manjunatha, Jordan Boyd-Graber, and Hal Daumé III. 2015. Deep unordered composition rivals syntactic methods for text classification. In *Proceedings of the 53rd annual meeting of the association for computational linguistics and the 7th international joint conference on natural language processing (volume 1: Long papers)*. 1681–1691.

[22] Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically generating commit messages from diffs using neural machine translation. In *Proceedings of 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 135–146.

[23] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering (TSE)* 28, 7 (2002), 654–670.

[24] Anjan Karmakar and Romain Robbes. 2021. What do pre-trained code models know about code?. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1332–1336.

[25] Sehoon Kim, Sheng Shen, David Thorsley, Amir Gholami, Woosuk Kwon, Joseph Hassoun, and Kurt Keutzer. 2021. Learned token pruning for transformers. *arXiv preprint arXiv:2107.00910* (2021).

[26] Diederik P. Kingma and Jimmy Ba. 2017. Adam: a method for stochastic optimization. arXiv:1412.6980 [cs.LG]

[27] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 7871–7880.

[28] 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).

[29] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

[30] 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 *Proceedings of IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 336–347.

[31] Liming Nie, He Jiang, Zhilei Ren, Zeyi Sun, and Xiaochen Li. 2016. Query expansion based on crowd knowledge for code search. *IEEE Transactions on Services Computing* 9, 5 (2016), 771–783.

[32] Matteo Paltenghi and Michael Pradel. 2021. Thinking Like a Developer? Comparing the Attention of Humans with Neural Models of Code. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 867–879.

[33] David Pisinger. 1995. Algorithms for knapsack problems. (1995).

[34] Md Rafiqul Islam Rabin, Vincent J. Hellendoorn, and Mohammad Amin Alipour. 2021. Understanding neural code intelligence through program simplification. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Association for Computing Machinery, 441–452.

[35] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[36] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research* 21 (2020), 1–67.

[37] Paige Rodeghero, Collin McMillan, Paul W McBurney, Nigel Bosch, and Sidney D'Mello. 2014. Improving automated source code summarization via an eye-tracking study of programmers. In *Proceedings of the 36th international conference on Software engineering*. 390–401.

[38] Anna Rogers, Olga Kovaleva, and Anna Rumshisky. 2020. A primer in bertology: what we know about how BERT works. *Transactions of the Association for Computational Linguistics* 8 (2020), 842–866.

[39] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108* (2019).

[40] Sheng Shen, Zhen Dong, Jiayu Ye, Linjian Ma, Zhewei Yao, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. 2020. Q-BERT: Hessian based ultra low precision quantization of BERT. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 8815–8821.

[41] Kathryn T Stolee, Sebastian Elbaum, and Daniel Dobos. 2014. Solving the search for source code. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23, 3 (2014), 1–45.

[42] Sahil Suneja, Yunhui Zheng, Yufan Zhuang, Jim A Laredo, and Alessandro Morari. 2021. Probing model signal-awareness via prediction-preserving input minimization. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 945–955.

[43] 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 (NeurIPS 2017)*. 5998–6008.

[44] Yao Wan, Wei Zhao, Hongyu Zhang, Yulei Sui, Guandong Xu, and Hai Jin. 2022. What do they capture? – a structural analysis of pre-trained language models for source code. In *In Proceedings of the 44th International Conference on Software Engineering (ICSE)*.

[45] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 8696–8708.

[46] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 87–98.

[47] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. 2019. Xlnet: generalized autoregressive pretraining for language understanding. *Advances in neural information processing systems* 32 (2019).

[48] Deming Ye, Yankai Lin, Yufei Huang, and Maosong Sun. 2021. TR-BERT: Dynamic Token Reduction for Accelerating BERT Inference. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 5798–5809.

[49] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200.

[50] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *Proceedings of IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 783–794.