

DeepClone: Modeling Clones to Generate Code Predictions^{*}

Muhammad Hammad^{1**}, Önder Babur¹, Hamid Abdul Basit², and Mark van den Brand¹

¹ Eindhoven University of Technology, Netherlands

{m.hammad,o.babur,m.g.j.v.d.brand}@tue.nl

² Prince Sultan University, Saudi Arabia

hbasit@psu.edu.sa

Abstract. Programmers often reuse code from source code repositories to reduce the development effort. Code clones are candidates for reuse in exploratory or rapid development, as they represent often repeated functionality in software systems. To facilitate code clone reuse, we propose *DeepClone*, a novel approach utilizing a deep learning algorithm for modeling code clones to predict the next set of tokens (possibly a complete clone method body) based on the code written so far. The predicted tokens require minimal customization to fit the context. DeepClone applies natural language processing techniques to learn from a large code corpus, and generates code tokens using the model learned. We have quantitatively evaluated our solution to assess (1) our model’s quality and its accuracy in token prediction, and (2) its performance and effectiveness in clone method prediction. We also discuss various application scenarios for our approach.

Keywords: language modeling · deep learning · code clone · code prediction

1 Introduction

Writing new code is an expensive activity, consuming considerable time and effort. Developers frequently perform adhoc code reuse, searching for code snippets over the web or in some codebase, followed by judicious copying and pasting [7]. Features like code snippet search, code prediction, code auto-completion and code generation can help developers to write code quickly and easily. Lately, language modeling have been effectively employed for these tasks [15, 17, 27].

A Language Model (LM) estimates the likelihood of sequences of tokens based on a training dataset, by assigning probabilities to tokens (words, subwords, or punctuation marks) or character sequences (sentences or words occurring after a given sequence [15]). Shannon first used language modeling [19] to predict the

^{*} The final authenticated version is available online at https://doi.org/10.1007/978-3-030-64694-3_9

^{**} Corresponding Author

next element following some given English text. Since then, several language models have been developed to perform different tasks. Various statistical and Deep Neural Networks (DNN) based techniques for language modeling have been applied to natural languages. These techniques are also applicable to programming languages [2, 4, 10, 24].

DNN techniques are powerful machine learning models that perform well in language modeling for source code, outperforming statistical language modeling techniques [15]. Performance of DNN techniques improves automatically through experience by learning data patterns; their power arising from the ability to perform parallel computations for a large number of training steps. LMs constructed using DNNs are called Neural Language Models (NLM), which have been used for various software development tasks like code completion [15, 27] and code clone detection [24] etc.

One common application of language modeling is code prediction [2, 4] - a technique for predicting next likely token(s) on the basis of user input (i.e., code written so far), by learning the source code features. These code predictions have a fixed threshold for the length of generated token sequence. In this work, we show how clone methods of arbitrary length, extracted from a large code repository, can enhance regular code generation and prediction applications.

Code clones are repeated patterns in code, usually created with the copy-paste ad hoc reuse practice. Around 5 to 50% of the code in software applications can be contained in clones [18]. Clones are generally considered harmful, and several techniques exist for avoiding and eliminating them [8]. However, clones can be useful in certain software development scenarios [14], one of them being exploratory development where the rapid development of a feature is required, and the remedial unification of newly generated clone is not clearly justified. Also, a piece of cloned code is expected to be more stable and poses less risk than new development.

We believe that clone methods, together with the non-cloned code, can be a useful component of a LM, as they represent commonly used functionality, and can be used for code prediction and completion. In this work, we exploit the re-usability aspect of code clones to build a NLM for predicting code tokens up-to the level of method granularity. We believe that our approach can help in improving the quality of code prediction by also predicting complete method bodies of arbitrary length based on clone methods. In this work, we have made the following contributions:

1. We present a novel approach for code prediction by explicitly modeling code clones along with non-cloned code.
2. Our approach can generate code predictions of complete method body of arbitrary length on the basis of user input.
3. We have quantitatively evaluated our approach using the BigCloneBench dataset, in terms of model quality and performance in various tasks including token prediction, and clone method prediction.

2 Related Work

To the best of our knowledge, no previous techniques has modeled code clones together with non-cloned code for predicting code tokens up-to the complete method granularity. However, many techniques have explored language modeling for token prediction, code suggestions or code completion. White et al. [25] applied Recurrent Neural Network (RNN) to model Java source code for code prediction. Bold [4] modeled Java language method statements and English language datasets by using Long Short Term Memory (LSTM). He compared the performance of next token prediction task with each other, arguing that method statements highly resemble English language sentences and are comparable to each other. Hellendoorn and Devanbu [10] noticed that source code NLMs underperform due to the unlimited vocabulary size as new identifiers keep coming with higher rates, and limiting vocabulary is not a good solution for NLMs. They proposed a nested scope, dynamically updatable, unlimited vocabulary count-based n-gram model, which outperforms the LSTM model on the task of token prediction. In contrast, Karampatsis et al. [15] solved the issue of unlimited vocabulary size by applying byte-pair encoding (BPE) technique in modeling the code. They compared the performance of n-gram and Gated Recurrent Unit (GRU) language models trained on source code datasets, and demonstrated that NLM trained on GRU can outperform n-gram statistical models on code completion and bug detection tasks if BPE technique is applied. Zhong et al. [27] applied the LSTM model with sparse point network to build a language model for JavaScript code suggestion. Deep TabNine ³ is a recently developed software programming productivity tool, successfully fine-tuned by using GPT-2 on approximately two million GitHub files capturing numerous programming languages, to predict the next chunk of code.

3 BigCloneBench for Code Clones

BigCloneBench [20,21] is the largest clone benchmark dataset, consisting of over 8 million manually validated clone method pairs in IJaDataset 2.0 ⁴- a large Java repository of 2.3 million source files (365 MLOC) from 25,000 open-source projects. BigCloneBench contains references to clones with both syntactic and semantic similarities. It contains the references of starting and ending lines of method clones existing in the code repository. In forming this benchmark, methods that potentially implement a given common functionality were identified using pattern based heuristics. These methods were manually tagged as true or false positives of the target functionality by judges. All true positives of a functionality were grouped as a clone class, where a clone class of size n contains $\frac{n(n-1)}{2}$ clone pairs. The clone types and similarity of these clone pairs were later identified in a post-processing step. Currently, BigCloneBench contains clones

³ <https://tabnine.com/blog/deep>

⁴ <https://sites.google.com/site/asegsecold/projects/seclone>

corresponding to 43 distinct functionalities. Further details can be found in the relevant publications [20, 21].

3.1 Dataset Preparation

We are using a reduced version of IJaDataset containing only the source files whose clone method references exist in BigCloneBench [20, 21]. The dataset is distributed into a number of smaller subsets, on the basis of 43 distinct functionalities.

We have performed several pre-processing steps to build our mutually exclusive training, testing, and validation datasets. First, we filtered IJaDataset files, keeping those which have references of true positive clone methods and discarding false positive clone references in BigCloneBench dataset (*Filtering*). Next, we distributed the set of files into training, validation, and testing datasets (*Distribution*). We adopted stratified sampling [22] to ensure that all types of clone methods appear in training, validation, and testing datasets. We distributed the set of files in each functionality folder into portions as per the following ratio: 80% training, 10% validation, and 10% testing. Then, we copied those files from original distribution to three separate folders such as training, validation, and testing. If any of the file already exist in one of those folders, we discarded it to avoid exact duplication [1]. Tables A5⁵ and 3 show the statistics of our datasets. Next, we marked the clone methods in the IJaDataset files by placing the meta-token $\langle \text{soc} \rangle$ at the start, and $\langle \text{eoc} \rangle$ at the end (*Marking*). We normalized our code by removing whitespaces, extra lines, comments (*Normalization*), and tokenized it by adapting Java 8 parser from Javalang⁶ Python library. We also replaced integer, float, binary, and hexadecimal constant values with the $\langle \text{num_val} \rangle$ meta-token (*Replacement*). Similarly, string and character values were replaced with $\langle \text{str_val} \rangle$. This reduced our vocabulary size, leading to faster training of the model [15, 25]. Finally, we merged all the tokenized data from the training, validation and testing files into three text files, i.e. train.txt, valid.txt, and test.txt (*Merging*). Table A4 demonstrates the pre-processing steps on an example of binary search clone method, while Table 3 gives an overview of our experimental dataset.

4 Neural Language Models for Code Clones

A number of techniques were available for developing an LM for BigCloneBench dataset such as n-gram [10], LSTM [12], GRU [5], GPT-2 [17]; as well as parameter settings for training those models. We could not evaluate all the possible combinations (hundreds) and especially very large scale models/training due to the resource limitations. We selected GRU [5] and GPT-2 [17] as they have been reported to outperform other comparable models with recommended configurations. In the following sections we describe the two models.

⁵ Tables A1, A2, A3, A4 and A5 can be accessible through link <https://www.win.tue.nl/~mhammad/deepclone.html>

⁶ <https://github.com/c2nes/javalang>

4.1 Gated Recurrent Units (GRU)

Gated recurrent units (GRUs) are a gating mechanism in RNNs [5], which is similar to LSTM [12] but has a forget gate and fewer parameters as it lacks an output gate. However, it is known to perform better than LSTM on certain tasks. To prepare our dataset (Section 3.1), we applied the recently proposed configuration settings for GRU deep learning model by Karampatsis et al. [15], which outperforms n-gram models on code completion and bug detection tasks. They used byte-pair encoding (BPE) technique to solve the unlimited vocabulary problem [2]. This problem makes it infeasible to train LMs on large corpora. BPE is an algorithm originally designed for data compression, in which bytes that are not used in the data replace the most frequently occurring byte pairs or sequences [6]. BPE starts by splitting all the words in characters. The initial vocabulary contains all the characters in the data set and a special end-of word symbol @@, and the corpus is split into characters plus @@. Then, it finds the most common pair of successive items in the corpus (initially characters, then tokens). This pair is merged in a new token which is added to the vocabulary; all occurrences of the pair are replaced with the new token. The process is repeated n times, which is called a merge operation (MO). We applied static settings with a large training set (50 epochs, 64 mini-batch size) and chose 10000 BPE MOs as it performs better than other BPE MOs such as 2000 and 5000. Static settings have been used to train a model on a fixed training corpus, and later evaluated on a separate test dataset. To train the LM, we first learned encoding by using the training set with the help of subword library ⁷. Then, we segmented the training, validation, and test sets using the learned encoding, and applied the MOs from BPE to merge the characters into subword units in the vocabulary.

4.2 Generative Pretrained Transformer 2 (GPT-2)

OpenAI developed a large-scale unsupervised LM called GPT-2 (Generative Pre-trained Transformer 2) [17] to generate several sound sentences of realistic text by extending any given seed. GPT-2 is a large transformer-based LM with 1.5 billion parameters, trained on a dataset of 8 million web pages. GPT-2 is trained with a simple objective: predict the next word, given all of the previous words within some text. We focus on fine-tuning a GPT-2 transformer [17] pre-trained model for generating code clones, even though it has been trained on English language. We applied fine-tuning of a pre-trained model on IJaDataset - a Java language dataset - as there exists a large amount of overlapping vocabulary with English language. GPT-2 transformer has demonstrated impressive effectiveness of pre-trained LMs on various tasks including high quality text generation, question answering, reading comprehension, summarization, and translation [17].

GPT-2 also has built in BPE tokenizer. We selected a small version of GPT2 (GPT2-117) as our base model, as it does not take too much time and resources to fine-tune, and is enough to evaluate our approach. The GPT2-117 [17] pre-trained model has vocabulary size of 50257, 117M parameters, 12-hidden layers,

⁷ <https://github.com/rsennrich/subword-nmt>

768-hidden states, and 12-attention heads. We have fine-tuned our GPT-2 based model on the partition of a GPU-1080Ti cluster (276 CPU cores, 329728 CUDA cores, 5.9 TB memory)⁸ for approximately 9 hours by using HuggingFace Transformer Library. In our experiment, we have performed training and evaluation with batch size per GPU of 1 for 5 epochs. We have used a learning rate of 5e-5 and the gradient accumulation steps (number of update steps to accumulate before performing a backward/update pass) as 5. Default values have been used for other hyper-parameters, as mentioned in the language modeling code⁹.

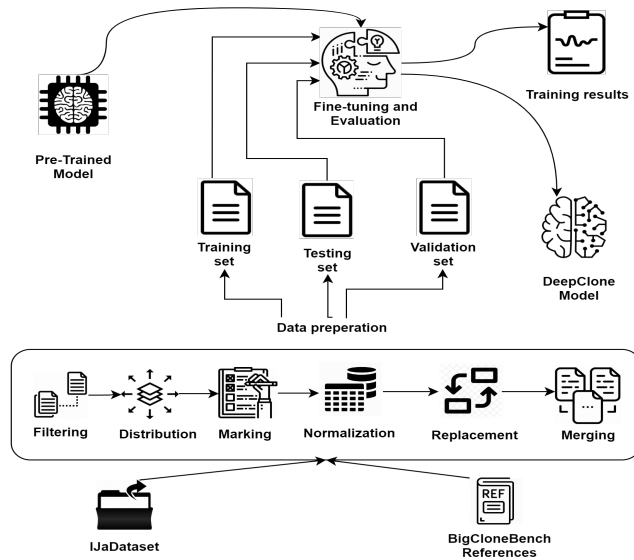


Fig. 1: DeepClone training process

5 Comparative Evaluation: GRU vs GPT-2 Based Models

We perform both intrinsic and extrinsic evaluations of GRU and GPT-2 based models to compare their performance. We calculate the perplexity scores (as done in related work [25, 26]) to measure the quality of models (i.e. intrinsic evaluation), which is an inverse of cross-entropy (as used in [10, 15]). Perplexity is a measurement of how well a given LM predicts sample data. It estimates the

⁸ <https://userinfo.surfsara.nl/>

⁹ <https://github.com/huggingface/transformers/blob/master/examples/language-modeling>

average number of code tokens to select from at each point in a sequence [2]. It is a natural evaluation metric for LMs, which represent a probability distribution over a subsequence or an entire dataset (Equation 1):

$$P(L) = \exp\left(-\frac{1}{M} \sum_i^M \log P(t_i|t_0 : t_{i-1})\right) \quad (1)$$

$P(t_i|t_0 : t_{i-1})$ is the conditional probability assigned by the model to the token t at index i . By applying \log of conditional probability, cross-entropy loss is calculated. M refers to the length of tokens. Hence, perplexity is an exponentiation of the average cross entropy loss from each token $[0, M]$. We calculate the perplexity on the validation set (**P1**) and the testing set (**P2**) for GRU and GPT-2 based models, which clearly displays that the GPT-2 based model outperforms the other by a large margin (Table 1).

We further measure the performance of both models on specific tasks such as token prediction (i.e. extrinsic evaluation). Given a number of code sequences as input, we collect the top 10 predictions from GRU and GPT-2 based models, and compute the top-k accuracy (the fraction of times the correct prediction appears in the top k predictions) for $k \in [1, 10]$. Moreover, we measure the Mean Reciprocal Rank (MRR) scores of both language models (LM), which has been used by many researchers for evaluating code prediction such as [10,15]. For each prediction done by the LM, we collect a ranked list of 10 predictions. For each of those lists, the reciprocal rank corresponds to the multiplicative inverse of the rank of the first correct answer. MRR in turn is the average of reciprocal ranks for all the input sequences used in the evaluation.

Table 1 shows the top-k accuracies as well as the MRR scores. Clearly, the results suggest that the GPT-2 based model performs more accurately compared to the GRU based model on pre-processed Java source code containing clone methods. The table also indicates that there is almost 77.808% chance to get a correct token in the first option, and 94.999% chance to have a correct output in the top-10 predicted outcomes for GPT-2 based model. To further quantify the accuracy of our models for token prediction task, we report an MRR score of 83%, which indicates an excellent performance in evaluating a ranked list of predictions for GPT-2 based model. As GPT-2 based model gives us highest performance in terms of perplexity on validation set (**P1**) and test set (**P2**), MRR, and top-k accuracy, we continue with that model for the rest of paper, named it as DeepClone model for further evaluation.

Table 1: Comparative evaluation results for GPT-2 and GRU models

Model	Perplexities		Accuracies				
	Validation (P1)	Test (P2)	MRR	Top 1	Top 3	Top 5	Top 10
GPT-2	2.145	2.146	84.329%	77.808%	90.040%	92.766%	94.999%
GRU	13.92	13.86	73.507%	66.948%	79.0715%	82.02%	84.787%

6 Further Evaluation of DeepClone Model

In this section we describe further evaluations of DeepClone on additional aspects of the model.

Training Evaluation At each checkpoint (the 500th logging step) of the training steps, we evaluate DeepClone model performance by calculating the perplexity on the validation set. Figure 2a describes the variations in perplexity on the validation set after each checkpoint. We observe that we achieve lowest perplexity **P1** (2.145) at step 24500. Figure 2c displays the convergence of the learning rate after each checkpoint. Learning rate helps in determining how quickly a neural network model learns a problem by adjusting the weights of a network with respect to the value of loss function. Another measure called loss function calculates a model error, which identifies how well a model predicts the expected outcome for any data point in the training set. GPT-2 uses cross-entropy loss function, which is to measure the performance of a LM whose output is a probability value between 0 and 1. Figure 2b displays a convergence of training losses after each checkpoint, which indicates how well the model behaves after each checkpoint of optimization. The loss value is finally minimized to 0.75 at step 24500, which is a sign of a well optimized deep learning model. Figure 1 describes the training process of the GPT-2 based model, which mentions the steps described in Section 3 that are used to perform the fine-tuning of our model. All these numbers imply a successful training and an accurate model. We have published our training results online¹⁰.



Fig. 2: Training graphs

The Effect of Using Clone Markers Besides the overall perplexity on the testing dataset (**P2**), we re-calculate the perplexity using the testing dataset but without the clone method markers (i.e. $\langle \text{soc} \rangle$ and $\langle \text{eoc} \rangle$). The motivation for this

¹⁰ <https://tensorboard.dev/experiment/tk1XqDi8RMqtrMjmVyQ9Sg>

additional measurement is as follows. Hindle et al. [11] observed that due to the repetitive nature of the code, there exist predictable statistical properties, which n-gram language models can capture and leverage for software engineering tasks. The sign of a good model is that it can capture the patterns in the dataset very well, which is particularly important for the task of clone method prediction. In Table 1, we can see an increase (3.6%) when comparing the original perplexity score of 2.146 (**P2**) and the perplexity on the test dataset without clone markers of 2.182 (**P3** see Table 4), showing that DeepClone performs better when the code has marked clone methods.

Evaluation per Clone Method In order to determine which clone method snippets are more predictable compared to the others, we calculate average perplexity score (\overline{PPL}) for each functionality type (see Table A5). We first extract the code snippet for each type of clone method for our testing dataset, and calculate the perplexity score. The scores, as depicted in Table A5, indicate how likely these can be predicted by DeepClone model. We also analyze several factors which can affect the perplexity of clone methods. BigCloneBench contains syntactic similarity scores for each clone method pair on the basis of tokens, calculated by using a line-based metric after normalization. We calculated the mean (μ) and variance (σ^2) values to determine the overall syntactic similarity of all the clone methods per each type of functionality, as listed in Table A5.

We observe that the perplexity scores vary according to the syntactic similarity between clone methods, as well as the number of clone method snippets in the training set. From the results, we can see that the "Test palindrome" type of clone method (number 44), which is used to test if a string is a palindrome, has the lowest perplexity score. It's thus well predicted by DeepClone. We attribute this to the high mean syntactic similarity (0.903 ± 0.040) among those types of clone methods, and the relatively small number of snippets (133) used in training. Too few number of snippets in the training may lead to (a) high perplexities and low predictability e.g. for "GCD" (number 26) to find the greatest common denominator and (b) no evaluation performed for "Decompress zip archive" clone method (number 5). Note that factors beside syntactical similarity and number of clone methods in the training set can also affect the perplexity score. In BigCloneBench, there are many false positive clone methods and other non-clone code snippets, which may be syntactically similar to true positive clone methods. Other factors such as clone types and hyper-parameters for GPT-2 are left to be explored in future work.

Non-Clone Methods vs Clone Methods Allamanis [1] noticed that a language model achieves low perplexity scores for code duplication, and high perplexity score for less duplicated code. In order to observe that difference, we calculated the perplexity scores for all the clone method snippets and non-clone method snippets in the testing dataset. We extracted clone method snippets by tracing the tokens, which come inclusively between $\langle \text{soc} \rangle$ and $\langle \text{eoc} \rangle$ tokens. All other snippets were considered to be a part of non-cloned code. We then calculate

the perplexity for each snippet. Finally, we take an average of the perplexities for both type of code snippets. Table 4, **P4** represents the average perplexity score for the clone method snippets, and **P5** represents the average perplexity of the non-cloned method snippets. We performed one-tailed Wilcoxon rank sum test to statistically compare **P4** with **P5**, which indicates that P4 is indeed less than P5 ($p < 0.001$). This shows that DeepClone correctly predicts clone method snippets much better than non-cloned snippets in general.

Performance on Other Datasets To evaluate the performance of DeepClone on another Java dataset, we use Allamanis et al.’s corpus [2] that contains over 14 thousand popular Java projects from Github. For base-lining, we focus only on 38 test projects that have been used in previous studies [10, 15]. We follow the same steps for dataset preparation as mentioned in Section 3.1, i.e., normalization, replacement, and merging. The dataset does not contain clone markers as no corresponding clones reference benchmark is available. As the main purpose of clone markers is to help in predicting clone methods, so it will not severely affect the results of predicting next tokens, as also noticed in Section 6. On this dataset, we achieve a perplexity of 2.996 (**P6**) equivalent to 1.097 as cross-entropy. We further calculate other accuracy measures like MRR (81.523%), top-1 (74.416%), top-3 (87.613%), top-5 (90.704%), and top-10 (93.152%). These results (see Table 2) outperform the static settings of previous studies [10, 15]. This indicates that DeepClone model is perfectly fine-tuned with GPT-2 over Java corpus in general, and it contains an excessive amount of overlapping vocabulary with Allamanis et al.’s [2] selected corpus.

Table 2: Performance of Allamanis et al.’s [2] dataset on different models

Model	Settings	Cross-Entropy	MRR
LSTM/300 [10]	Static	3.22	66.1%
LSTM/650 [10]	Static	3.03	67.9%
BPE NLM (512) [15]	BPE 10000, Static, Small train	4.77	63.75%
DeepClone	Section 4.2	1.097	81.523%

7 Clone Method Prediction

In this section, we demonstrate how clone methods can be predicted from DeepClone model on the basis of user context. Furthermore, we measure its various aspects like rapid development and quality.

7.1 Experimental Design

For predicting a clone method based on the user context, there exist several text generation methods such as beam search [23] and nucleus sampling [13].

All these methods have a specific decoding strategy to shape the probability distribution of LM with higher probabilities assigned to higher quality texts. We selected nucleus sampling as it is claimed to be best the strategy for generating large amount of high quality text, comparable to human written text [13]. By using a fine-tuned model and nucleus sampling, we can expect a coherent set of code tokens for clone method prediction. Holtzman et al. [13] have also achieved coherent text generation results with similar settings.

We performed a small scale (100 context queries) experiment to predict next token subsequences by choosing different subsequence sizes of 10, 20, 30, 50, and 100 tokens. Among these, subsequences with size 20 gave us the best results in terms of top-k accuracy and MRR. We extracted subsequences of 20 tokens from the testing dataset, and moved the sliding window one step ahead to obtain further subsequences. From these we randomly selected 735 subsequences containing a total of 14,700 tokens, in which $\langle \text{soc} \rangle$ token is a part of each subsequence, indicating a start of clone method. We passed these one by one to DeepClone model, and kept on predicting new tokens with nucleus sampling (threshold value 0.95) until the meta-token $\langle \text{eoc} \rangle$ (i.e. end of clone) appeared. We used the text generation script¹¹ of HuggingFace Transformer Library in this case. Note that certain parameters, e.g. the number of subsequences and size of tokens per subsequence are chosen to perform a preliminary evaluation, to be fine-tuned and optimized in a follow-up study. The focus of this paper is to demonstrate the feasibility of our methodology for predicting clone methods. We have mentioned examples from our results in Tables (A1, A2, A3). To make the outputs readable in the tables, we have formatted the code by using the online tool¹² along with little manual editing, which we plan to automate in future.

7.2 Evaluation

Rapid Development In this experiment, we successfully generated 92,926 tokens associated with clone methods. Given the 735 cases, this amounts to an average of ~ 126 tokens per case. As a comparison, other approaches [13, 23] traditionally employ a threshold-based strategy of generating a certain number of code tokens up to a maximum threshold value of t . Note that t is typically a low value, e.g. 1 for simple token prediction, and 5-20 for the popular Deep Tab-Nine auto-completer. Nevertheless, we can see that DeepClone, with the clone marking strategy, is able to outperform threshold-based strategies even with an extremely generous configuration of $t = 50$ or even 100. Furthermore, threshold-based strategies may not generate a set of code tokens for a complete method in a single pass, as the length of complete method varies. Marking the clone method regions in the dataset helps DeepClone to generate complete methods in a single pass. We conclude DeepClone model not only helps developers to code rapidly, but also provides a coherent set of code tokens for a complete method.

¹¹ https://github.com/huggingface/transformers/blob/master/examples/text-generation/run_generation.py

¹² https://www.tutorialspoint.com/online_java_formatter.htm

Quality We measured the quality of DeepClone output by using ROUGE (Recall-Oriented Understudy for Gisting Evaluation) [16]. It is designed to compare an automatically generated summary or translation against a set of reference summaries (typically human-generated). In our context, it helps us to automatically determine the quality of original DeepClone output by comparing it with the ground truth. ROUGE doesn't try to assess how fluent the clone method is. It only tries to assess the adequacy by simply counting how many n-grams in the DeepClone output match the ones in the ground truth. As ROUGE is based only on token overlap, it can determine if the same general concepts are discussed between an automatic and a reference summary, but it cannot determine if the result is coherent or the clone method is semantically correct. High-order n-gram ROUGE measures try to judge fluency to some degree.

We calculated precision (P), recall (R), and F-measure (F) of ROUGE-1, ROUGE-2, and ROUGE-L between DeepClone output and ground truth. ROUGE-1 refers to the overlap of unigrams between some reference output and the output to be evaluated. ROUGE-2, in turn, checks for bigrams instead of unigrams. The reason one would use ROUGE-1 over or in conjunction with ROUGE-2 (or other finer granularity ROUGE measures), is to also indicate fluency as part of the evaluation. The intuition is that the prediction is more fluent if it more closely follows the word orderings of the reference snippet. Finally, ROUGE-L measures longest matching sequence of tokens between machine generated text/code and human produced one by using longest common subsequence (LCS). LCS has a distinguishing advantage in evaluation: it captures in-sequence (i.e. sentence level flow and word order) matches rather than strict consecutive matches. DeepClone predicted clone method can be extremely long, capturing all tokens in the retrieved clone methods, but many of these tokens may be useless, making it unnecessarily verbose. This is where precision comes into play. It measures what portion of the DeepClone output is in fact relevant and desirable to be kept with respect to the reference output.

$$\text{Precision} = \frac{\# \text{ of overlapping tokens}}{\text{total tokens in the predicted output}} \quad (2)$$

Recall in the context of ROUGE measures what portion of the reference output was successfully captured by the DeepClone output.

$$\text{Recall} = \frac{\# \text{ of overlapping tokens}}{\text{total } \# \text{ tokens in reference snippet}} \quad (3)$$

We also report the F-measure which provides a single score that balances both the concerns of precision and recall.

$$\text{F-Measure} = 2 * \frac{\text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}} \quad (4)$$

We have measured different ROUGE scores, i.e. ROUGE-1, ROUGE-2, and ROUGE-L, to evaluate the similarity (and the quality to a certain extent) of the DeepClone output to ground truth. In this step, we extract only the tokens between $\langle \text{soc} \rangle$ and $\langle \text{eoc} \rangle$ (inclusive) from the DeepClone output and ground truth

(see Tables A1, A2, and A3). We observe quite reasonable scores for ROUGE-1 and ROUGE-L against ROUGE-2 (Table 5). This depicts that the DeepClone output contains reasonably well overlap of uni-grams and longest sequence matches of tokens with the ground truth compared to bi-grams.

In our qualitative investigation, we experienced two different scenarios based on the input context. The first one is when the context contains the method name. It is straightforward for the neural language technique to generate the predicted clone method following the given method name and current context. Table A1 gives an example of this scenario, where "transpose" method name is mentioned in the context and our approach predicts the clone method, whose functionality type matches the ground truth. The second scenario is based on the context that does not contain a method name. This can have two different output sub-scenarios. The first one is when the functionality type of the ground truth do not match. As we see in Table A3, the context does not have the full signature of the clone method. This makes the generated output by DeepClone using nucleus sampling deviate from the functionality type of the ground truth. Ground truth belongs to "copy file" functionality, while DeepClone output belongs to "delete directory", which eventually leads to low and largely deviating ROUGE scores between the DeepClone output and the ground truth (see Table 5 and the example in Table A3). These clone methods may or may not fulfil the desired goal of the user. So, it might be useful to guide the users to include the clone method name in the context for better results. The other output sub-scenario is when we manage to successfully generate DeepClone output whose functionality type matches with the ground truth. In Table A2, "copy file" method name is not mentioned in the context, but the functionality type of the DeepClone output matches with the ground truth. We notice that the total number of "copy file" clone methods used in DeepClone training are 2,454, which allows nucleus sampling to generate DeepClone output closer to ground truth in example A2. Overall, we believe our approach yields good results and can assist the developers by correctly predicting clone methods in different scenarios.

8 Discussion

Our approach leads to promising results. The performance metrics in the training (learning rate approaching 0, minimized loss) and validation (perplexity of 2.145) phases all indicate a fine-tuned model. The series of calculated perplexity scores allow us to conclude that DeepClone model can predict regularities successfully in terms of clone markers, including the code in general and the individual clone snippets in particular. The extrinsic evaluation reveals that we achieve high accuracy, notably 95% in the top 10 suggestions, as well as larger number of tokens than a threshold-based strategy even with a generous threshold of 100. With a high quality and accurate model as the foundation, we next discuss the potential use cases to exploit our model, as well as the limitations to our work.

Table 3: Final Distribution of BigCloneBench Dataset

	Files	Clone Methods	Tokens
Training	9,606	11,991	16,933,894
Validation	1,208	1,499	2,130,360
Testing	1,234	1,502	2,235,982
Total	12,048	14,992	21,300,236

Table 4: Perplexities

P3	P4	P5	P6
2.182	2.410	2.767	2.996

Table 5: Empirical evaluation results between DeepClone output and ground truth

ROUGE-1	
Precision	0.667 ± 0.192
Recall	0.559 ± 0.226
F-measure	0.56 ± 0.185
ROUGE-2	
Precision	0.479 ± 0.217
Recall	0.398 ± 0.218
F-measure	0.4 ± 0.202
ROUGE-L	
Precision	0.652 ± 0.165
Recall	0.586 ± 0.183
F-measure	0.599 ± 0.153

Use Cases for DeepClone DeepClone model can be utilized to assist developers in various scenarios. Some of these have already been mentioned above: predicting the next token (as typically done by many LMs) or the complete clone method body. The latter, while seemingly straightforward, can be enhanced with a more elaborate ranking and retrieval mechanism rather than simply generating the most likely sequence of tokens one after another. For that purpose, the additional information in BigCloneBench, including the exact clone method clusters (methods representing the same functionality), clone types, and so on can be exploited. Another use case might involve clone refactoring (and avoidance), by recommending extract method refactoring instead of predicting a complete clone method snippet. In combination with some additional code transformations, the clone methods can be converted to reusable assets (e.g. in the form of libraries). The model can also be used to perform code search for common functionalities.

Limitations and Threats to Validity Our approach is the first step for code prediction that raises the granularity level to complete methods. However, we cannot expect exactly the same clone method being predicted or completed as the one used in training by DeepClone model. In prediction tasks, generating well-formed outputs is challenging, which is well-known in natural language generation [9]. The desired output might be a variation of another, previously observed sample [9], due to the probabilistic nature of the LM; the space of possible clone methods that could be generated grows exponentially with the length of the clone methods. An extension of the current work would involve displaying the most similar cloned methods (as is) from the dataset to the user.

Some limitations originate from the selected dataset. BigCloneBench only contains clone references of methods for the 43 common functionalities, i.e. not all the clones are marked in the dataset. Although it is enough to validate our methodology, modeling all the clones might result in more interesting findings.

Although BigCloneBench is a well-known dataset, it does not necessarily represent Java language source code entirely (a threat to external validity).

In our study, we relied on the HuggingFace transformer implementation of GPT-2 to train and evaluate DeepClone model. While GPT-2 is a reliable architecture used in many NLP experiments [17], HuggingFace transformer implementation is still an emerging project. However, our results and trends are aligned with those obtained in the field of NLP. Hence, we are positive that the results are reliable. As for the clone method prediction, we have only used nucleus sampling. Other techniques such as beam search can also be explored.

9 Conclusion and Future Work

In this work, we proposed DeepClone, a deep learning based cloned code language model. We have performed intrinsic and extrinsic evaluations to determine its performance in predicting clone methods. The extensive evaluation suggests that our approach significantly improves code prediction by exploiting deep learning and code clones. In future work, we plan to implement the potential use cases of this model (Section 8). The proposed LM can be improved by hyper-parameter optimizations, as well as by better training (e.g. on a larger dataset or larger pre-trained GPT-2 models). We also plan to investigate how to tackle different types and granularity levels of code clones such as simple clones, structural clones, file clones, and clones of other artifact types such as models [3, 8].

Acknowledgment

Dr. Sohaib Khan (CEO at Hazen.ai) provided valuable feedback on the experimentation. SURFsara provided credits for experiments. The project is partly funded by Prince Sultan University Faculty Research Fund.

References

1. Allamanis, M.: The adverse effects of code duplication in machine learning models of code. In: Proc. of the 2019 ACM SIGPLAN Int. Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. pp. 143–153 (2019)
2. Allamanis, M., Sutton, C.: Mining source code repositories at massive scale using language modeling. In: Proceedings of the 10th Working Conference on Mining Software Repositories. pp. 207–216. IEEE Press (2013)
3. Babur, Ö., Cleophas, L., van den Brand, M.: Metamodel clone detection with SAMOS. *Journal of Computer Languages* **51**, 57 – 74 (2019)
4. Boldt, B.: Using lstms to model the java programming language. In: International Conference on Artificial Neural Networks. pp. 268–275. Springer (2017)
5. Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y.: Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078* (2014)
6. Gage, P.: A new algorithm for data compression. *C Users Journal* **12**(2), 23–38 (1994)

7. Gharehyazie, M., Ray, B., Filkov, V.: Some from here, some from there: Cross-project code reuse in github. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR). pp. 291–301. IEEE (2017)
8. Hammad, M., Basit, H.A., Jarzabek, S., Koschke, R.: A systematic mapping study of clone visualization. *Computer Science Review* **37**, 100266 (2020)
9. Hashimoto, T.B., Guu, K., Oren, Y., Liang, P.S.: A retrieve-and-edit framework for predicting structured outputs. In: *Advances in Neural Information Processing Systems*. pp. 10052–10062 (2018)
10. Hellendoorn, V.J., Devanbu, P.: Are deep neural networks the best choice for modeling source code? In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. pp. 763–773. ACM (2017)
11. Hindle, A., Barr, E.T., Gabel, M., Su, Z., Devanbu, P.: On the naturalness of software. *Communications of the ACM* **59**(5), 122–131 (2016)
12. Hochreiter, S., Schmidhuber, J.: Long short-term memory. *Neural computation* **9**(8), 1735–1780 (1997)
13. Holtzman, A., Buys, J., Forbes, M., Choi, Y.: The curious case of neural text degeneration. *arXiv preprint arXiv:1904.09751* (2019)
14. Kapser, C.J., Godfrey, M.W.: “cloning considered harmful” considered harmful: patterns of cloning in software. *Empirical Software Engineering* **13**(6), 645 (2008)
15. Karampatsis, R.M., Babii, H., Robbes, R., Sutton, C., Janes, A.: Big code!= big vocabulary: Open-vocabulary models for source code (2020)
16. Lin, C.Y.: Rouge: A package for automatic evaluation of summaries. In: *Text summarization branches out*. pp. 74–81 (2004)
17. Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I.: Language models are unsupervised multitask learners. *OpenAI Blog* **1**(8), 9 (2019)
18. Roy, C.K., Cordy, J.R.: A survey on software clone detection research. *Queen’s School of Computing TR* **541**(115), 64–68 (2007)
19. Shannon, C.E.: Prediction and entropy of printed english. *Bell system technical journal* **30**(1), 50–64 (1951)
20. Svajlenko, J., Islam, J.F., Keivanloo, I., Roy, C.K., Mia, M.M.: Towards a big data curated benchmark of inter-project code clones. In: 2014 IEEE International Conference on Software Maintenance and Evolution. pp. 476–480. IEEE (2014)
21. Svajlenko, J., Roy, C.K.: Bigcloneeval: A clone detection tool evaluation framework with bigclonebench. In: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 596–600. IEEE (2016)
22. Trost, J.E.: Statistically nonrepresentative stratified sampling: A sampling technique for qualitative studies. *Qualitative sociology* **9**(1), 54–57 (1986)
23. Vijayakumar, A.K., Cogswell, M., Selvaraju, R.R., Sun, Q., Lee, S., Crandall, D., Batra, D.: Diverse beam search for improved description of complex scenes. In: *Thirty-Second AAAI Conference on Artificial Intelligence* (2018)
24. White, M., Tufano, M., Vendome, C., Poshyvanyk, D.: Deep learning code fragments for code clone detection. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. pp. 87–98. ACM (2016)
25. White, M., Vendome, C., Linares-Vásquez, M., Poshyvanyk, D.: Toward deep learning software repositories. In: *Proceedings of the 12th Working Conference on Mining Software Repositories*. pp. 334–345. IEEE Press (2015)
26. Zaremba, W., Sutskever, I., Vinyals, O.: Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329* (2014)
27. Zhong, C., Yang, M., Sun, J.: Javascript code suggestion based on deep learning. In: *Proceedings of the 2019 3rd International Conference on Innovation in Artificial Intelligence*. pp. 145–149 (2019)