# Competitive Programming Verification Using Pretrained Transformer Models

**Brendan Kondracki**
bk2793@columbia.edu

**Ke Li**
kl3352@columbia.edu

**Zixiang Yin**
zy2444@columbia.edu

## Abstract

Program synthesis models have made tremendous strides in recent years due to the leveraging of highly parameterized transformer architectures. Solutions to problems such as clone detection and programming language translation have been able to reach human levels of performance under this framework. More recently, transformer models have been shown to perform at a level on par with human coders in competitive programming competitions. Despite this success, questions still exist about the degree to which such models understand the semantics of generated programs, and to what extent they are replicating solutions provided during training due to the breadth of data they are exposed to. In this paper, we demonstrate the understanding capabilities of such models by finetuning a sequence-to-sequence transformer, CodeT5, on the task of competitive programming verification. Furthermore, we utilize a state of the art programming translation metric, CodeBLEU, to synthesize a new label set for the building of a regression model for correctness scoring.

## 1 Introduction

### 1.1 Transformers

Recent transformer models, such as Codex (Chen et al., 2021), have demonstrated high levels of performance on program synthesis tasks in relation to natural language prompts. More recently, AlphaCode (Li et al., 2022) was developed with a fine grained objective of generating solutions to competitive programming problems. This model proved to be highly effective, achieving a ranking in the top 54% of participants when evaluated on competitions hosted on the Codeforces platform. In addition, when fined-tuned on the APPS dataset (Hendrycks et al., 2021), AlphaCode outperformed the much larger parameterized Codex model on competition level problems, demonstrating the effectiveness of this training objective.

In addition to the advances in program synthesis, other models have been developed to ascertain the understanding capabilities of transformers in relation to programming languages. CodeT5 (Wang, 2021) is one such model, which leverages the previously constructed T5 model (Raffel, 2019) to accommodate a number of programming understanding tasks such as code defect and clone detection. A critical element in the pretraining of CodeT5 was the utilization of a masked identifier prediction (MIP) objective, in which code identifiers such as function and variable names were masked with a unique sentinel token and predicted in an auto-regressive manner. The authors of the CodeT5 paper found that the removing of MIP in the model's pretraining negatively impacted its performance on the defect detection task, implying its usefulness in understanding the semantics of code segments.

Preprint. Under review.

### 1.2 CodeBLEU

A traditional evaluation method of language translation is BLEU (Papineni et al., 2002), or the Bilingual Evaluation Understudy, which counts the number of n-grams in a candidate translation which match those found in a set of reference translations. The BLEU score is quick to calculate, language independent, and easy to understand. Despite this, however, the BLEU metric on its own has proved to be unreliable when applied to the context of programming languages. For instance, if we have two candidates $a - b$ and $c + d$, and the reference prompt is $a + b$, $a - b$ will receive a higher BLEU score as it shares more tokens with the reference but it is logically wrong. In addition, BLEU scoring fails to take into account the functional use of identifiers in a program, as two code segments can perform identical logical steps and operations with completely different variable and function names. Therefore, applying BLEU in the context of code translation is suboptimal in nature. New evaluation methods such as CodeBLEU (Ren et al., 2020) have tried to alleviate such concerns by attempting to capture both syntactic and semantic equivalence between program segments in its scoring. The metric integrates a weighted version of the BLEU score, an accuracy measure which compares the sub-trees of candidate and reference syntax, and another measure that utilizes data flow information:

$$CodeBLEU = \alpha \cdot BLEU + \beta \cdot BLEU_{weight} + \gamma \cdot Match_{ast} + \theta \cdot Match_{df}$$

First, the weighted BLEU is introduced to assign higher weights for different language specific keywords. For example, words such as "if", "else", "return", "import", etc., would receive higher weights when compared to non-keyword based text in the code segment. The formula for the weighted BLEU is:

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

where $BP$ is the brevity penalty and $p_n$ represents the weighted n-gram matching precision. Secondly, the syntatic AST Match calculates the accuracy through comparing all the candidates and the reference subtrees using the tree-sitter [1] parsing result. Then, the score is calculated by the following equation:

$$Match_{ast} = Count_{clip}(T_{cand})/Count(T_{ref})$$

where $Count_{clip}(T_{cand})$ is the number of matched candidate subtrees and $Count(T_{ref})$ is the number of reference subtrees. Thirdly, semantic data-flow match is used to detect the semantic mistake in the source code. Data-flow (Guo et al., 2020) is used to convert a source code as a graph where nodes are variables and edges are the value that each variable comes from. After we obtain the data-flow graph, each item is normalized by renaming them by their order amongst all data-flow items. Then, the semantic data-flow match score is calculated as:

$$Match_{df} = Count_{clip}(DF_{cand})/Count(DF_{ref})$$

where $Count_{clip}(DF_{cand})$ is the number of matched candidate data-flows and $Count(DF_{ref})$ is the number of reference data-flows.

Currently there are only a handful of ways to evaluate code generation models, and it is not clear any one of them outperforms the others. In order to understand the degree to which such models understand the semantics of generated programs, we propose to construct a model aimed at scoring code segments by utilizing the metric of CodeBLEU and MIP in CodeT5, which we believe can be leveraged to great effect in the verification of programming solutions, as the semantic understanding of code segments is critical in the successful evaluation of solutions, particularly for incorrect ones.

[1] https://github.com/tree-sitter/tree-sitter

## 2 Methods

### 2.1 Binary Classification

To gauge the initial effectiveness of CodeT5 in understanding the semantic relationship between problem descriptions and corresponding solutions, we defined a new binary classification task with which to fine-tune CodeT5 on. Namely, given an input of $(problem, solution)$, the objective of the model is to return a unigram sequence of 1 or 0, denoting whether the given solution correctly satisfies the problem requirements. As is the case with other CodeT5 tasks, we created a unique "$Verify$ :" prefix which was prepended to each data sample before being passed into our model in order to identify the targeted objective. In addition, a $[SEP]$ token was placed in between the problem description and programming solution for each example in order for the model to differentiate between the two excerpts.

We utilized the pretrained CodeT5 model provided through Huggingface as the target model to finetune for our task. Due to hardware restrictions, we utilized the 60 million parameter small version of the model as opposed to the 220 million parameter base version. Looking at the performance of both models on previously constructed CodeT5 tasks, we believe the difference in performance between the two models to be minimal.

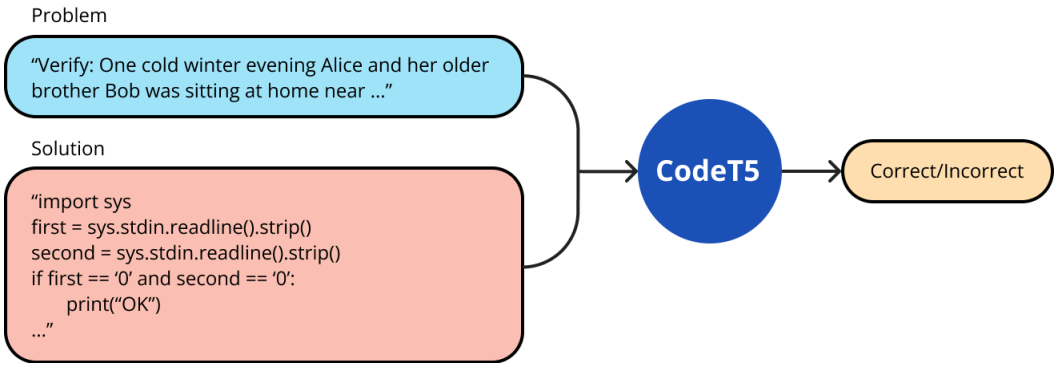


Figure 1: Illustration of our Binary Classification task

### 2.2 Regression

Next, in order to further gauge the effectiveness of CodeT5 to understand the semantics of programming language, we defined a new regression task for scoring the correctness of code segments on a scale of 0-1 relative to their natural language prompts. Intuitively, we gave a label of 1 to all correct solutions of a given problem, and generated labels for all incorrect solutions through the use of CodeBLEU as follows: For each incorrect solution, we conducted CodeBLEU scoring using a maximum of 15 correct reference solutions for each of the four scoring components. We selected the values for $alpha$, $beta$, $gamma$, $theta$ to be 0.1, 0.1, 0.4, 0.4, respectively, as recommended in the CodeBLEU paper.

After labeling our data in this fashion, we then formatted our target values by rounding them to two decimal places. This allowed us to reduce the range of expected output values in order to effectively leverage the sequence-to-sequence structure, a technique used for a similar regression problem in the T5 paper. After finishing doing this, we then went ahead and utilized a problem structure similar to the one specified for the binary classification task. Namely, we prepended each concatenation of $(problem, solution)$ pair with a unique "$Score$ :" prefix, and seperated the problem and solution excerpts with a unique $[SEP]$ token. In addition, we utilized the 60 million parameter small version of CodeT5 model, for the same computational reasons as specified in the previous subsection.

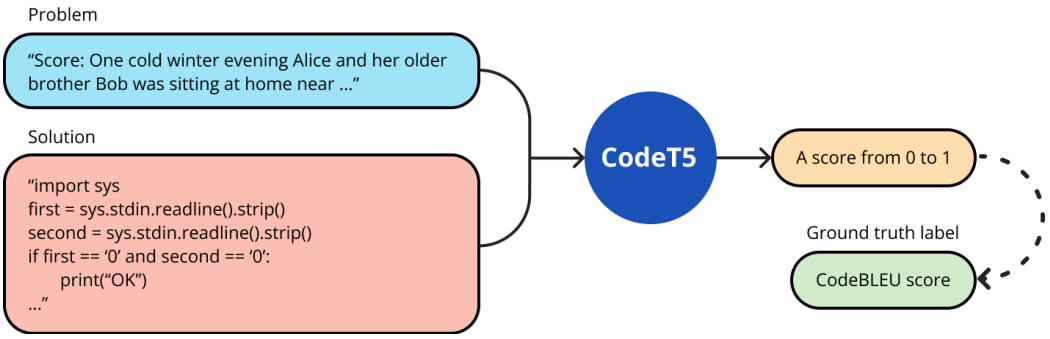


Figure 2: Illustration of our Regression task

### 2.3 Data

In order to evaluate competitive programming questions and answers, we utilized the CodeContests [2] dataset that Alphacode was trained on. This dataset consists of 13,000 competitive programming problems, with corresponding correct/incorrect solutions, across Aizu, AtCoder, CodeChef, Codeforces, and HackerEarth platforms. In order to accommodate the 512 token maximum input length of CodeT5, we pruned the original CodeContests dataset on all $(problem, solution)$ concatenations which did not meet these requirements. After conducting this pruning, we were left with roughly 5 million examples consisting of solutions written in Python, Java, and C++. Due to hardware and time restrictions, we decided to conduct our training solely on solutions written in Python, because this was the language with the highest prevalence in CodeT5's initial training. This left us with 1,050,304 Python exclusive problem/solution pairs to utilize for our binary classification training/validation process. Among these exclusive problem/solution pairs, 438,996 of them corresponded to correct solutions, while the remaining 611,308 corresponded to incorrect solutions.

For the purpose of regression training, we had to further prune this set of solutions due to the existence of certain problem sets containing no correct reference solutions. The lack of such solutions prevented the effective labeling of any incorrect solutions for these problem sets, therefore requiring us to ignore such sets entirely. After this pruning was completed, we were left with a total of 1,045,739 Python exclusive problem/solution pairs to use for our regression training and validation.

## 3 Results

### 3.1 Binary Classification

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.82 | 0.85 | 0.84 | 157291 |
| 1 | 0.78 | 0.73 | 0.76 | 111835 |
| accuracy | | | 0.80 | 269126 |
| macro avg | 0.80 | 0.79 | 0.80 | 269126 |
| weighted avg | 0.80 | 0.80 | 0.80 | 269126 |

Figure 3: Results on validation data

[2] https://github.com/deepmind/code_contests

We conducted our initial training over a single epoch on 75% of the 1,050,304 examples, and utilized the remaining 25% for validation. We utilized AdamW optimization with an initial learning rate of $1 \times 10^{-4}$. In addition, we utilized a batch size of 16, with 8 gradient accumulation steps, yielding an effective global batch size of 128.

As seen in Figure 1, our fine-tuned model achieved an overall accuracy of 80% on our validation data, consisting of 269,126 examples. In addition, we found that on a per-class basis, our model achieved higher precision and recall scores on negative data samples compared to positive ones. We believe this could be due to the slight imbalance between these two classes, as roughly 58% of our data belongs to the negative class.

### 3.2 Regression

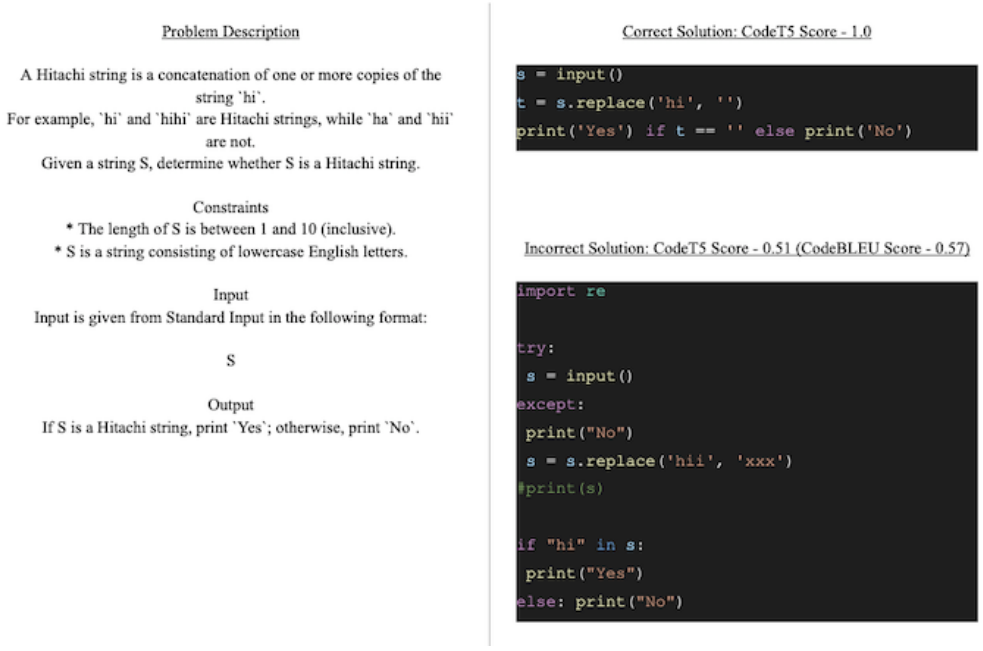


Figure 4: Example input/output on regression model

Similarly to our binary classifier, we conducted our training over a single epoch on 75% of the 1,045,739 examples, and utilized the remaining 25% for validation. In addition, we used the same hyperparameters for our optimizer, learning rate, batch size, and gradient accumulation steps. After training our regression model, we achieved a Mean Absolute Error of $0.16$ over a hold-out validation dataset.

Figure 4 shows an example problem from our validation dataset, as well as a corresponding correct and incorrect solution. Analyzing the two submissions, we can see that the top code segment correctly solves the problem of detecting Hitachi strings by replacing occurrences of the substring "hi" with the empty string, and then checking whether the modified string is empty or not. On the other hand, the bottom code segment follows a similar syntactic structure as the correct solution, taking in user input and performing a replace operation on it, however fails to correctly replace the proper substrings before performing the final evaluation. Our fine-tuned CodeT5 model accurately predicts the top code segment as being correct relative to the problem description. In addition, our model also predicts a value of $0.51$ for the bottom code segment, coming very close to the CodeBLEU evaluated label of $0.57$.

Code for our data and evaluation can be found at `https://github.com/brendankon/COMS4995-Team10`.

### 3.3 Limitations

Since we have no ground truth (e.g. human evaluation scores) about incorrect solutions, we take the recommended values in the CodeBLEU paper for four hyperparameters $(\alpha, \beta, \gamma, \theta)$ while building our regression model. For this reason, the used hyperparameters might not be optimal if the underlying distribution of CodeContests differs from that of the dataset used in the CodeBLEU paper. In addition, hardware constraints played a role in both our data and model utilizations. Due to time and resource limitations, we were not able to take advantage of our entire dataset, and merely built our models for evaluating Python code. Furthermore, our methods utilized a smaller 60 million parameter CodeT5 model. We believe that although this did not greatly hinder the overall accuracy of our fine-tuned models, performance improvements could be made by repeating these experiments using the 220 million parameter CodeT5 model as our base.

## 4 Conclusion

In this paper, we propose a novel perspective of evaluating the quality of competitive programming excerpts by using highly parameterized transformer models. In addition to syntactic correctness, we further demonstrate the degree to which such models understand the semantics of generated codes by finetuning a sequence-to-sequence transformer, CodeT5, on a programming competition dataset with CodeBLEU scores as ground truth. After training, our verification and scoring model achieved a satisfactory accuracy of 80% and an MAE of 0.16 respectively on a hold-out validation set.

In the future, we plan to fully utilize the CodeContests dataset and expand the functionality of our model so that it could evaluate synthesized code not only in Python but also in other languages such as Java, Go, C$^{\#}$, etc. as well.

## References


Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harri Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, et al. (2021) Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. (2021) Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938, 2021*.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi (2021) CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. *arXiv preprint arXiv:2109.00859*

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, Peter J. Liu (2019) Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *arXiv:1910.10683*

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, et al. (2022) Competition-Level Code Generation with AlphaCode *arXiv:2203.07814*

Kishore Papineni, Salim Roukos, Todd Ward, and Wei jing Zhu. (2002) Bleu: a method for automatic evaluation of machine translation. In ACL.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Ming Zhou, Ambrosio Blanco, and Shuai Ma. (2020) Codebleu: a method for automatic evaluation of code synthesis. arXiv preprint arXiv:2009.10297.

Guo, D.; Ren, S.; Lu, S.; Feng, Z.; Tang, D.; Liu, S.; Zhou, L.; Duan, N.; Yin, J.; Jiang, D.; et al. 2020. GraphCodeBERT: Pre-training Code Representations with Data Flow. arXiv preprint arXiv:2009.08366.