# QTC4SO: Automatic Question Title Completion for Stack Overflow

Yanlin Zhou[†], Shaoyu Yang[†], Xiang Chen[†*], Zichen Zhang[†], Jiahua Pei[†]
[†]*School of Information Science and Technology*, *Nantong University*, China
Email: yanlin.z@stmail.ntu.edu.cn, shaoyuyoung@gmail.com, xchencs@ntu.edu.cn, 384758226@qq.com, 761365414@qq.com

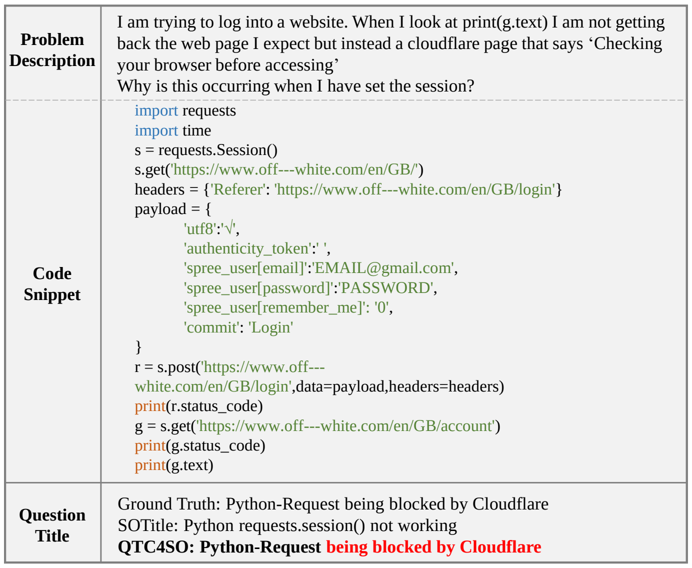
***Abstract*—Question posts with low-quality titles often discourage potential answerers in Stack Overflow. In previous studies, researchers mainly focused on directly generating question titles by analyzing the contents of the posts. However, the quality of the generated titles is still limited by the information available in the post contents. A more effective way is to provide accurate completion suggestions when developers compose titles. Inspired by this idea, we are the first to study the problem of automatic question title completion for Stack Overflow and then propose a novel approach QTC4SO. Specifically, we first preprocess the gathered post titles to form incomplete titles (i.e., tip information provided by developers) for simulating the scene of this task. Then we construct the multi-modal input by concatenating the incomplete title with the post's contents (i.e., the problem description and the code snippet). Later, we adopt multi-task learning to the question title completion task for multiple programming languages. Finally, we adopt a pre-trained model T5 to learn the title completion patterns automatically. To evaluate the effectiveness of QTC4SO, we gathered 164,748 high-quality posts from Stack Overflow by covering eight popular programming languages. Our empirical results show that compared with the approaches of directly generating question titles, our proposed approach QTC4SO is more practical in automatic and human evaluation. Therefore, our study provides a new direction for automatic question title generation and we hope more researchers can pay attention to this problem in the future.**

***Index Terms*—Stack Overflow, Question Title Completion, Multi-task Learning, Pre-trained Model, Human Study**

## I. Introduction

Stack Overflow has been widely used by developers as one of the most common ways to seek answers related to technical questions [1]–[3]. A high-quality question post is likely to attract more attention from potential answerers and the importance of post quality was verified in previous studies [4]–[6], [6]–[8]. To help developers effectively write question posts, Stack Overflow has provided a list of quality assurance guidelines for community members[1]. Moreover, researchers also aimed to improve the post quality from different perspectives (such as duplicated question detection [9], [10], question title generation [11]–[13]). In this study, we mainly focus on the problem of question title generation for Stack Overflow. To our best knowledge, Gao et al. [11], [14] were the first to study this problem and proposed the approach CODE2Que by analyzing the code snippets. Then Liu et al. [12] proposed the approach SOTitle and Zhang et al. [13] proposed the approach CCBERT, which aimed to generate question titles by analyzing the bi-modal inputs (i.e., problem descriptions and the code snippets).

However, the quality of question titles generated by previous approaches [11]–[13] is still limited, which is restricted by the information available in the post contents. During the preparation process of the post, a more promising way is to provide accurate completion suggestions when developers compose titles. We use a real post from Stack Overflow[2] shown in Fig. 1 to illustrate our research motivation. For this post, if we analyze the problem description and the code snippet, SOTitle [12] can generate the title "Python requests.session() not working", which cannot effectively reflect the purpose of this post. However, if the developer provides some tip information (i.e., "Python-Request"), our proposed approach QTC4SO can provide an accurate completion suggestion (i.e., "being blocked by Cloudflare").


**Problem Description**
I am trying to log into a website. When I look at print(g.text) I am not getting back the web page I expect but instead a cloudflare page that says 'Checking your browser before accessing'
Why is this occurring when I have set the session?

**Code Snippet**
```
import requests
import time
s = requests.Session()
s.get('https://www.off---white.com/en/GB/')
headers = {'Referer': 'https://www.off---white.com/en/GB/login'}
payload = {
        'utf8':'√',
        'authenticity_token':' ',
        'spree_user[email]':'EMAIL@gmail.com',
        'spree_user[password]':'PASSWORD',
        'spree_user[remember_me]': '0',
        'commit': 'Login'
}
r = s.post('https://www.off---
white.com/en/GB/login',data=payload,headers=headers)
print(r.status_code)
g = s.get('https://www.off---white.com/en/GB/account')
print(g.status_code)
print(g.text)
```

**Question Title**
Ground Truth: Python-Request being blocked by Cloudflare
SOTitle: Python requests.session() not working
**QTC4SO: Python-Request being blocked by Cloudflare**

Fig. 1. A post from Stack Overflow to show the motivation of our study

In this study, we aim to study the problem of question title completion for Stack Overflow. To solve this problem, we propose a novel approach QTC4SO. In particular, given a post's contents (i.e., the problem description and the code snippet) and an incomplete title, QTC4SO can automatically

* Xiang Chen is the corresponding author.
[1]https://stackoverow.com/help/how-to-ask

[2]https://stackoverflow.com/questions/49087990

provide the completion suggestion. To train the title completion model, we first gather high-quality question posts from Stack Overflow by considering eight popular programming languages. Each selected post contains a complete question title, the problem description, and the code snippet. Then, we preprocess the gathered original question titles to adapt to our question title completion task. Specifically, we generate three different incomplete titles for each complete title. To generate the incomplete title, we randomly mask a certain number of words at the end of this title. Then, we formalize the question title completion for each programming language as separate but related tasks and resort to multi-task learning [15], which can help to better exploit complementary and shared knowledge between different tasks. Finally, we adopt a pre-trained model T5 [16] to learn the potential title completion patterns from our gathered corpus.

In our empirical design, we gathered 164,748 high-quality posts from Stack Overflow by covering eight programming languages as our experimental subject. To verify the effectiveness of QTC4SO, we consider five performance measures (i,e., ROUGE [17], GLEU [18], BLEU [19], Perfect predictions, Levenshtein distance [20]), which have been widely used in previous text summarization [21]–[23] and text completion [24]–[28] studies. In our study, we want to answer the following four research questions (RQs).

**RQ1: Can QTC4SO generate higher-quality question titles than state-of-the-art baselines in an automatic evaluation way?**

**Results.** We consider a traditional language model $N$-gram [29] and a recently proposed question title generation approach SOTitle [12] as baselines. Comparison results show that QTC4SO can achieve better performance than these two baselines for different programming languages.

**RQ2: Can QTC4SO achieve the best performance when considering both the problem description and the code snippets in the post?**

**Results.** We design two control approaches (i.e., $w/o\ desc$ and $w/o\ code$) to verify the contribution of code snippets and problem descriptions for QTC4SO. According to the results of our ablation experiment, we find using both problem descriptions and code snippets can help to achieve the best performance.

**RQ3: Can QTC4SO achieve the best performance when considering the pre-trained model T5?**

**Results.** We investigate the performance impact of different pre-trained models (such as T5 [16], BERT [30], CodeBERT [31], and BART [32]) on QTC4SO. The final results demonstrate that QTC4SO with T5 can achieve the best performance.

**RQ4: Can QTC4SO generate higher-quality title completion suggestions than state-of-the-art baselines by human study?**

**Results.** Since the automated evaluation results may not correlate well with the quality of the generated titles [33], we further conduct a human study to investigate the quality of titles generated by QTC4SO in terms of the similarity, naturalness, and informativeness criteria. Our human evaluation results also demonstrate the competitiveness of QTC4SO.

Compared to previous studies on question title generation [11]–[13], our study investigates a more practical problem. By providing some tip information when developers compose question titles, our proposed approach QTC4SO can help to provide accurate title completion suggestions, which can eventually generate higher-quality question titles.

The main contributions are summarized as follows.

- **Direction.** We are the first to study the task of question title completion for Stack Overflow, which opens a new direction for automatic question title generation.
- **Approach.** For this task, we propose a novel approach QTC4SO based on the multi-modal input (i.e., the incomplete title with the contents of the post). QTC4SO adopts multi-task learning to this task for multiple programming languages and a pre-trained model T5 to learn the title completion patterns automatically.
- **Corpus.** We build a large-scale corpus by mining Stack Overflow for this task, which contains 164,748 high-quality posts covering eight programming languages.
- **Study.** We conduct both automatic evaluation and human evaluation to verify the competitiveness of QTC4SO.
- **Tool.** We developed a browser plugin tool based on QTC4SO to assist developers in composing question titles. The video demonstration of using our tool is available at https://www.youtube.com/watch?v=4njf4zgDdRs.

## II. Our Proposed Approach QTC4SO

We show the framework of QTC4SO in Fig. 2. Specifically, QTC4SO contains three phases: the corpus construction phase, the model construction phase, and the model inference phase. In the rest of this section, we show the details of each phase.

### *A. The Corpus Construction Phase*

In this phase, we first use programming language tags to collect relevant question posts from the data dump $Posts$, which includes all question posts generated in Stack Overflow from July 2008 to March 2022. Here, we mainly concern with eight popular programming languages (i.e., Python, Java, C#, JavaScript, PHP, Ruby, Go, and HTML), which are chosen based on the popularity statistics of tags in Stack Overflow[3]. Then we filter low-quality posts by heuristic rules. In our study, we design four rules according to suggestions from previous studies [34], [35] and the characteristics of this task. These four heuristic rules are shown as follows.

- **Rule 1:** The score of the question posts should not be smaller than 10.
- **Rule 2:** The question posts should contain the code snippets.
- **Rule 3:** The question posts should have the accepted answers.
- **Rule 4:** The title length of the question posts should not be smaller than 4 words.

[3]https://stackoverflow.com/tags?tab=popular

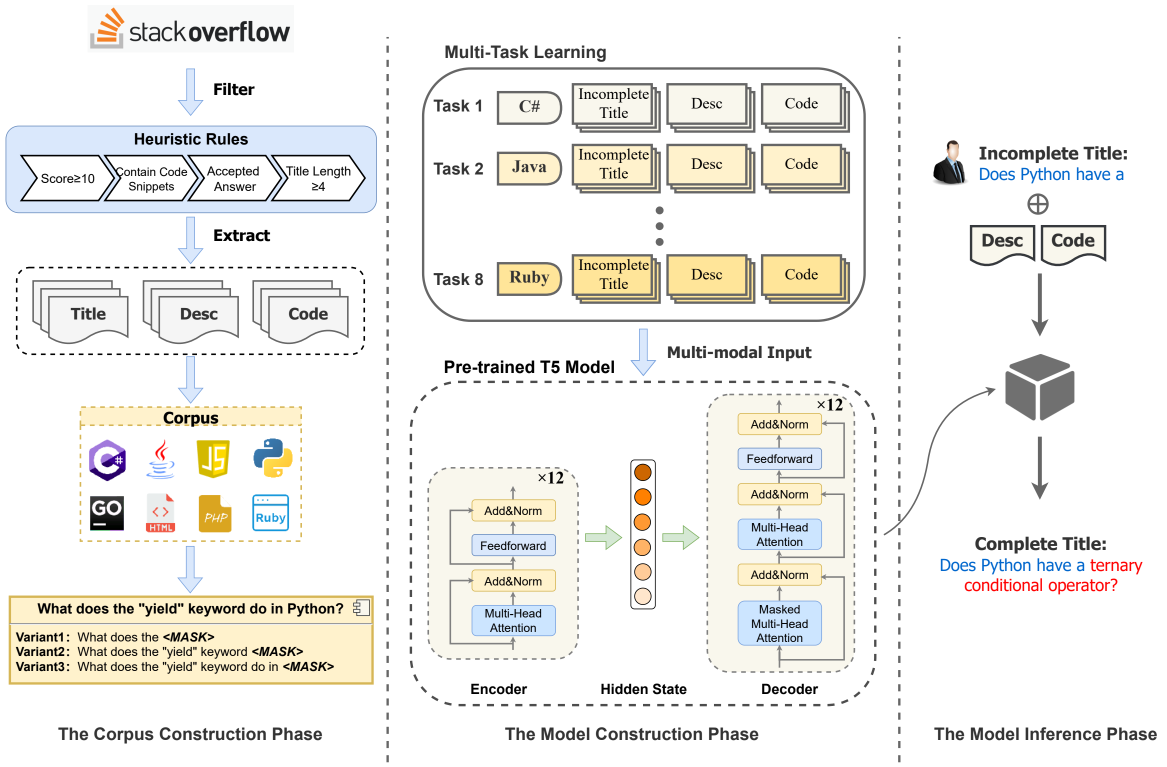

stackoverflow
Filter
Heuristic Rules
Score≥10
Contain Code Snippets
Accepted Answer
Title Length ≥4
Extract
Title
Desc
Code
Corpus
What does the "yield" keyword do in Python?
Variant1: What does the <MASK>
Variant2: What does the "yield" keyword <MASK>
Variant3: What does the "yield" keyword do in <MASK>
The Corpus Construction Phase
Multi-Task Learning
Task 1 C# Incomplete Title Desc Code
Task 2 Java Incomplete Title Desc Code
Task 8 Ruby Incomplete Title Desc Code
Multi-modal Input
Pre-trained T5 Model
Add&Norm
Feedforward
Multi-Head Attention
Masked Multi-Head Attention
×12
Encoder
Hidden State
Decoder
The Model Construction Phase
Incomplete Title:
Does Python have a
Desc
Code
Complete Title:
Does Python have a ternary conditional operator?
The Model Inference Phase

Fig. 2. Framework of our proposed approach QTC4SO

Next, we build the initial corpus by gathering the question title, the problem description, and the corresponding code snippet from each collected post. In this corpus, we treat each post as a triplet <Title, Desc, Code>. Previous studies [11]–[13] mainly generated question titles from scratch. However, our study mainly concerns the question title completion task. Therefore, we need to prepare incomplete titles before training the question title completion model. Assuming that the question title $t_i$ contains $n$ words, we can define the masking operator as follows. Specifically, this operator first generates a random number $n_{rand}$ ($1 < n_{rand} < n$). Then it generates an incomplete title (i.e., variant) with the last $n_{rand}$ words masked. Examples of generating incomplete titles for a specific post are shown in the lower left part of Fig. 2. We illustrate the rationality of our designed masking operator as follows. First, since developers always compose titles in a linear way (i.e., from the first word to the last word), the masking operator selects the last consecutive words to mask. Second, since our study mainly concerns the question title completion task, the incomplete titles should at least contain one word. Therefore, the masking operator can at most mask $n-1$ words. In our study, we generate three different incomplete titles for each title by applying the masking operator with three different values of $n_{rand}$, and the reasons are illustrated as follows. First, using Rule 4 can ensure that each title can generate at least three different incomplete titles. Second, generating more incomplete titles may help to improve model performance, but also significantly increase the model construction cost.

## *B. The Model Construction Phase*

In this phase, we formalize the question title completion tasks for different programming languages as separate but related tasks and then use multi-task learning [15] by exploiting complementary and shared knowledge. To synthesize the multi-modal input, we concatenate the incomplete question title and the post contents (i.e., the problem description and the code snippet). To alleviate the OOV (Out-Of-Vocabulary) problem, we apply the SentencePiece method [36] to split the multi-modal input. Finally, we train our model by adopting a pre-trained model T5 [16] to automatically learn title completion patterns.

*1) Multi-task Learning:* In our study, we represent the multi-modal input by concatenating the incomplete title $X_{incomp}$ with the content of the question post, which contains the problem description $X_{desc}$ and the code snippet $X_{code}$. In particular, we first use a special identifier (<body>) to distinguish $X_{incomp}$ and $X_{desc}$. We second use a special identifier (<code>) to distinguish $X_{desc}$ and $X_{code}$. Finally, we add a $[mask]$ tag after the incomplete title to indicate the masked part of the title, which will be predicted by our trained model. Since we formalize question title completion tasks for different programming languages as separate but related tasks, we resort to multi-task learning [15] in our study, in which a shared model is trained on multiple tasks at the same time. As a method of transferring knowledge across related tasks, multi-task learning [15] improves model generalization by exploiting the domain-specific information contained in the related tasks and capturing the common features through sharing hidden layers among all tasks. Thus, the over-fitting issue can be alleviated and the model can be more adaptable to new tasks in the future. As shown in Fig. 2, we assign the task-specific prefix to the input $X$ for different programming languages (e.g., the prefix "JS:" represents the programming language JavaScript), which enables the model to distinguish different tasks. The final input format is shown as follows.

$$\begin{aligned} X = prefix \oplus X_{incomp} \oplus [MASK] \\ \oplus < body > \oplus X_{desc} \oplus < code > \oplus X_{code} \end{aligned} \quad (1)$$

**SentencePiece.** To alleviate the OOV problem, we use the SentencePiece method [36] to split the input $X$. As a simple, efficient, and language-independent pre- and post-processor, SentencePiece can be easily integrated into neural network-based text generation systems, which are increasingly moving towards language-independent architectures. SentencePiece implements two subword segmentation algorithms: byte-pair-encoding (BPE) and the unigram language model. Therefore, by using SentencePiece, we can build a full end-to-end system without relying on any language-specific processing.

*2) Pre-trained Encoder-Decoder Transformer Model:* The T5 model proposed by Raffel et al. [16] is a unified pre-trained encoder-decoder Transformer model [37], which can convert all text-based language problems into a text-to-text format and allow for transfer learning and multi-task learning. The primary building block of T5 is self-attention [38] and T5 can be trained with two stages: the pre-training stage and the fine-tuning stage. Specifically, the former stage allows for the construction of a shared knowledge base that is useful for a large number of sequence-to-sequence tasks. While the latter stage fine-tunes the pre-trained model to a specific downstream task.

In our study, we fine-tune a pre-trained T5 model and its encoder-decoder Transformer implementation roughly follows the original proposal of Raffel et al. [16]. First, an input token sequence is mapped to an embedding sequence and then is passed into the encoder. Each block of the encoder is made up of two subcomponents: a self-attention layer and a small feed-forward network. Self-attention [38] is calculated by using queries ($Q$) and keys ($K$) with the dimension $d_k$, and values ($V$) with the dimension $d_v$. To be more precise, the dot product of the queries and keys is first computed. Then the weight of each value is then calculated using the softmax function after each has been divided by $\sqrt{d_k}$. The output matrix is calculated as follows:

$$Attention(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \quad (2)$$

The feed-forward network (FFN) is composed of two linear transformations separated by a ReLU activation.

Layer normalization [39] is applied to the input of each subcomponent. Following the layer normalization, a residual skip connection [40] adds each subcomponent's input to its output.

The decoder is similar to the encoder in structure except that it includes a masked self-attention layer, the purpose of which is to prevent the model from noticing unknown information during model training. Suppose in the ($k$+1)-$th$ decoding step, we have generated the first $k$ tokens of the predicted title and then input the embedding vector of the generated sequence and the hidden vectors of the encoder output to the decoder, the model will generate the ($k$+1)-$th$ token.

Since the output of our question title completion task is a long sequence, we find utilizing beam search [41] can improve the performance. Beam search can return a list of the most likely output sequences, which can provide the developer with a few of the most likely title completion suggestions. Specifically, it selects the $m$ tokens with the lowest heuristic cost at each time step, where $m$ denotes the beam width, by scanning through each step's title tokens one by one. It continues to select probable tokens for the following tokens after trimming any remaining branches until it comes across the end-of-sequence sign. Finally, for each question post, our model may return $m$ candidate question titles. According to their average probabilities during the beam search operation, we rank the generated candidate question titles.

*3) Fine-tuning the T5 Model:* After modeling the multi-modal inputs by concatenating the incomplete titles with the contents of the question posts and adding prefixes to them, we fine-tune the T5 model in a multi-task setting, in which each task is automatic question title completion for a specific programming language.

In our study, we train the parameters $\theta$ of our model by minimizing the loss function for eight programming languages as follows.

$$\theta^* = arg \min_{\theta} \frac{1}{8} \sum_{i=1}^{8} L(y_i, f(x_i; \theta)) \quad (3)$$

### *C. The Model Inference Phase*

In this phase, by analyzing a question post's content and tip information (i.e., the incomplete title) provided by the developer, our trained model can provide completion suggestions for composing the question title.

## III. Experimental Setup

### *A. Research Questions*

In our empirical study, we design the following four research questions (RQs).

**RQ1: Can QTC4SO generate higher-quality question titles than state-of-the-art baselines in an automatic evaluation way?**

**Motivation.** To demonstrate the effectiveness of QTC4SO, we choose five automatic measures to evaluate the quality of question titles generated by QTC4SO and our considered baselines in this RQ.

**RQ2: Can QTC4SO achieve the best performance when considering both the problem description and the code snippets in the post?**

**Motivation.** In this RQ, we want to conduct ablation experiments by investigating the performance impact of different input modals (i.e., the problem description and the code snippet) on QTC4SO.

**RQ3: Can QTC4SO achieve the best performance when considering the pre-trained model T5?**

**Motivation.** In this RQ, we want to investigate the performance impact of different pre-trained models on QTC4SO. Therefore, we consider the other three popular pre-trained models from the field of natural language processing.

**RQ4: Can QTC4SO generate higher-quality title completion suggestions than state-of-the-art baselines by human study?**

**Motivation.** To effectively evaluate the semantic information in the generated question titles and avoid the weakness of the automatic evaluation measures (e.g., some measures are designed by only considering the lexical overlap between the ground-truth titles and the generated titles), we want to conduct a human study to evaluate the effectiveness of QTC4SO in this RQ.

## *B. Experimental Subject*

As described in Section II-A, we finally gathered a total of 164,748 question posts from Stack Overflow based on four heuristic rules. Then we split the gathered initial corpus according to 80%: 10%: 10% as the training set, the validation set, and the test set, respectively. Notice in our corpus split, we consider the temporal relationship of posts and guarantee that the test set contains the latest posts. The reason is that partitioning posts in chronological order is more applicable to the real-world application, and can relieve the data leakage problem caused by the homogeneous questions between the training set and the test set. The statistical information of the corpus for different programming languages is shown in Table I.

TABLE I
THE STATISTICAL INFORMATION OF OUR CONSTRUCTED CORPUS

| Language | Training Set | Validation Set | Test Set |
|---|---|---|---|
| Python | 30,404 | 3,801 | 3,801 |
| Java | 25,136 | 3,142 | 3,143 |
| C# | 25,332 | 3,167 | 3,167 |
| JavaScript | 27,219 | 3,402 | 3,403 |
| PHP | 10,573 | 1,322 | 1,322 |
| Ruby | 4,521 | 565 | 566 |
| Go | 1,742 | 218 | 218 |
| HTML | 6,867 | 858 | 859 |
| **Total** | **131,794** | **16,475** | **16,479** |

Finally, to adapt the initial corpus to the title completion task, we generate three different incomplete titles for each question title based on the masking operator introduced in Section II-A.

## *C. Performance Measures*

We consider five performance measures, which have been widely used in previous similar tasks (such as text summarization [21]–[23], source code understanding and generation [42]–[48] and text completion [24]–[28]). Notice, when computing the values of these measures for the question title completion approaches (such as QTC4SO), we only focus on the masked tokens in the ground-truth title (i.e., the ground-truth string) and the completed tokens in the generated title (i.e., the generated string).

Since the title completion task can be treated as a special text summarization task, we first consider three text similarity-based measures.

**ROUGE**. ROUGE [17] is based on recall. In our study, we use ROUGE-L, which is based on the longest common subsequence, to measure the lexical overlap between the generated string and the ground-truth string.

**BLEU**. BLEU [19] is used to measure the degree of similarity between the generated string and the ground-truth string. Specifically, BLEU_n determines how many $n$-grams appear in the generated string by calculating the $n$-gram accuracy between the generated string and the ground truth.

**GLEU**. GLEU [18] is a customized metric from BLEU. GLEU is widely used to evaluate GEC (Grammatical Error Correction) systems and is shown to highly correlate with human judgments on GEC tasks [49].

Since question title completion is also similar to the code completion task, we further consider two related measures.

**Perfect predictions (PP).** This measure returns the percentage of instances where the model predicts the same sequence as the ground-truth sequence. In our question title completion task, we calculate the percentage of perfect predictions (PP_k) when the model successfully guesses the first $k$ masked tokens.

**Levenshtein distance (LD)**. This measure [20] is based on the word level and calculates the minimum number of edit operations required to convert from one string to the other string. In our study, we compute the Levenshtein distance (LD_k) by counting the smallest number of word edits required to convert the predicted string into the ground truth title by considering the first $k$ masked tokens. Different from the previous four measures, the smaller the value of LD, the more similar the two strings (i.e., the better the performance of the model).

To guarantee the correct implementation of these performance measures, we aim to use mature libraries. For example, we compute the values of BLEU, GLEU, and ROUGE using the NLTK package[4] and the ROUGE package[5], respectively.

## *D. Experimental Settings*

We implement QTC4SO and baselines by using the Pytorch framework[6] and the pre-trained parameters of T5-base[7]. The hyperparameter setting of our experiments is shown in Table II. To alleviate the overfitting problem, we use an early stop strategy [50] (i.e., the model stops training when the loss on the validation set does not decrease in 10 consecutive iterations, and gather the parameter values from the model with the best performance). Since multi-task learning is used in

[4]http://nltk.org/
[5]https://pypi.org/project/rouge
[6]https://pytorch.org/
[7]https://huggingface.co/t5-base

QTC4SO, we select the training step where the model achieves the best performance on different tasks.

TABLE II
THE HYPERPARAMETER SETTING OF OUR PROPOSED APPROACH QTC4SO

| Hyper-parameter Name | Hyper-parameter Value |
|---|---|
| encoder layers | 12 |
| decoder layers | 12 |
| hidden size | 768 |
| early_stopping_patience | 10 |
| num_beams | 10 |
| maximum input length | 512 |
| maximum output length | 48 |
| length_penalty | 1.2 |

Our experiments were conducted on a computer with an Intel(R) Xeon(R) 4210 CPU and a GeForce RTX3090 GPU with 24 GB memory.

## IV. RESULT ANALYSIS

### A. Result Analysis for RQ1

**Approach.** Since we are the first to study automatic question title completion for Stack Overflow, there are no corresponding baselines for this problem. Therefore, we consider a traditional language model $N$-gram and the recently proposed question title generation approach SOTitle [12] as our baselines.

**SOTitle.** SOTitle [12] is a Transformer-based approach by leveraging the code snippets and the problem descriptions. It uses a multi-head attention mechanism to efficiently capture long-term dependencies.

**$N$-gram.** $N$-gram [29] is a traditional language model. The next token can be predicted using only the preceding $N$-1 tokens. In our study, by iterating over the language model's whole vocabulary, the model can determine the most likely continuation of an input title sequence. We investigated different $N$ ($N \leq 4$) values by running the models on the training set and determined the best value (i.e., 3-gram).

We replicated these baselines by using scripts shared by the original studies [12], [29]. Then we used the same data split strategy and optimized the hyper-parameters of these baselines to ensure a fair comparison.

**Results.** Table III shows the comparison results of QTC4SO and baselines in terms of all performance measures for eight different programming languages respectively. Then we emphasize the best performance for each performance measure in bold. Notice there are some differences between the title generation approach and the title completion approach. Therefore, for SOTitle, we compute the values of the first three measures by analyzing the generated title and the ground-truth title. Then we compute the values of the last two measures by analyzing the first $k$ tokens of the generated title and the ground-truth title. In this table, we can find that our proposed approach QTC4SO can achieve the best performance. Taking the C# programming language as an example, compared to SOTitle, QTC4SO can improve the performance by 16.39%, 42.06%, 25.88%, 131.69%, and 18.80% for ROUGE-L, GLEU, BLEU_4, PP_5, and LD_5 respectively. Specifically, the traditional text completion model $N$-gram performs poorly on our title completion task. We conjecture that a possible reason is that $N$-gram can not effectively solve the long-term dependency problem since this approach learns only a limited number of $N$ consecutive tokens. Moreover, we find that QTC4SO significantly outperforms the title generation method SOTitle since QTC4SO considers the developer's intention in question title generation. Obviously, if we compute the performance value by considering all the tokens in the generated and ground-truth titles, QTC4SO can achieve better performance in the first three measures.

Fig. 3 shows two examples[8] of generating titles for question posts. The top part of the figure shows the question body. While the bottom part shows the ground truth and the titles generated by different approaches. Notice we emphasize the completion suggestion in red for $N$-gram and QTC4SO. In both examples, we find that the titles generated by SOTitle and $N$-gram cannot effectively reflect the purpose of the posts. However, the titles generated by QTC4SO can provide the key contents of the question posts in a brief and accurate way, thus demonstrating the effectiveness of QTC4SO and the importance of developer participation when composing titles.

**Summary for RQ1:** QTC4SO can generate higher-quality question titles than state-of-the-art baselines in an automatic evaluation way.

### B. Result Analysis for RQ2

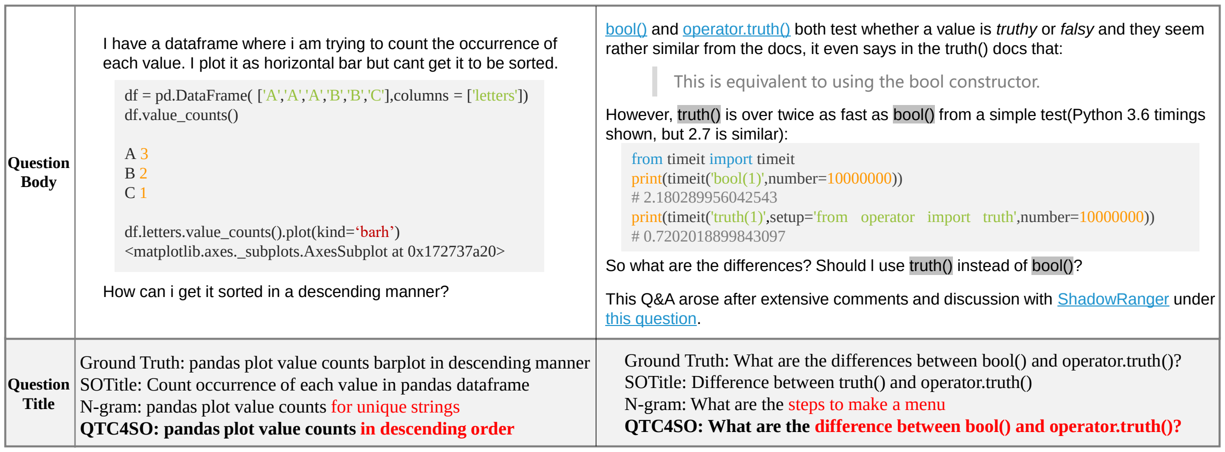
**Approach.** In this RQ, we design two control approaches to verify the contributions of the problem description and code snippets for QTC4SO. Specifically, we use $w/o\ desc$ to denote the control approach, which does not consider the problem description for QTC4SO. In a similar way, we use $w/o\ code$ to denote the control approach, which does not consider the code snippet for QTC4SO. To guarantee a fair comparison, QTC4SO and two control approaches follow the same experimental setup.

**Results.** Due to the paper length limitation, we only show the performance of QTC4SO with different input modalities for four programming languages in Table IV, and all results can be found on our project homepage. When only considering code snippets ($w/o\ desc$), the performance of QTC4SO is significantly decreased. Specifically, in terms of ROUGE-L, the performance of QTC4SO drops by 138.60% on average. When only considering the problem descriptions ($w/o\ code$), the performance of QTC4SO is slightly decreased. These results demonstrate that the information from code snippets and problem descriptions is in a certain complementary. Moreover, after we manually analyze some sampled posts, we find considering both the code snippets and problem descriptions can help to improve the quality of the generated titles, although the problem descriptions can make more contributions for QTC4SO, since we find the problem descriptions can provide more semantic information for the title completion task.

[8]The URL of the first post is https://stackoverflow.com/questions/49059956 and the URL of the second post is https://stackoverflow.com/questions/49009870.

TABLE III
COMPARISON RESULTS BETWEEN OUR PROPOSED APPROACH QTC4SO AND TWO BASELINES

| Language | Approach | ROUGE-L (%) | GLEU (%) | BLEU_1 / 2 / 3 / 4 (%) | PP_1 / 3 / 5 (%) | LD_1 / 3 / 5 |
|---|---|---|---|---|---|---|
| Python | N-gram | 4.17 | 1.95 | 2.87 / 0.85 / 0.30 / 0.10 | 7.75 / 0.13 / 0.02 | 0.92 / 2.87 / 4.83 |
| | SOTitle | 27.19 | 13.21 | 29.03 / 20.29 / 15.15 / 11.84 | 21.55 / 7.97 / 2.96 | 0.79 / 2.46 / 4.16 |
| | **QTC4SO** | **30.95** | **17.96** | **32.09 / 23.20 / 17.85 / 14.37** | **39.52 / 13.34 / 7.03** | **0.61 / 2.07 / 3.64** |
| Java | N-gram | 2.91 | 1.30 | 2.16 / 0.44 / 0.00 / 0.00 | 6.02 / 0.00 / 0.00 | 0.94 / 2.91 / 4.88 |
| | SOTitle | 25.61 | 12.63 | 27.22 / 19.13 / 14.32 / 11.28 | 21.16 / 7.45 / 2.97 | 0.79 / 2.48 / 4.21 |
| | **QTC4SO** | **29.99** | **18.08** | **29.94 / 21.75 / 16.95 / 13.84** | **39.11 / 13.10 / 6.95** | **0.61 / 2.11 / 3.70** |
| C# | N-gram | 3.08 | 1.41 | 2.16 / 0.52 / 0.16 / 0.06 | 6.03 / 0.06 / 0.00 | 0.94 / 2.90 / 4.86 |
| | SOTitle | 26.96 | 13.22 | 27.86 / 20.00 / 15.26 / 12.25 | 21.66 / 8.08 / 3.66 | 0.78 / 2.46 / 4.17 |
| | **QTC4SO** | **31.38** | **18.78** | **31.27 / 23.35 / 18.53 / 15.42** | **40.90 / 14.30 / 8.48** | **0.59 / 2.04 / 3.51** |
| JavaScript | N-gram | 3.12 | 1.36 | 2.39 / 0.54 / 0.16 / 0.07 | 6.33 / 0.01 / 0.00 | 0.94 / 2.90 / 4.85 |
| | SOTitle | 29.44 | 14.96 | 30.26 / 21.90 / 16.94 / 13.70 | 22.89 / 9.02 / 4.06 | 0.77 / 2.43 / 4.08 |
| | **QTC4SO** | **33.19** | **20.48** | **33.34 / 24.91 / 19.87 / 16.57** | **41.53 / 15.30 / 8.45** | **0.59 / 1.99 / 3.51** |
| PHP | N-gram | 3.18 | 1.55 | 2.21 / 0.57 / 0.27 / 0.19 | 6.28 / 0.11 / 0.12 | 0.94 / 2.90 / 4.87 |
| | SOTitle | 29.38 | 15.02 | 30.00 / 21.66 / 16.90 / 13.85 | 21.86 / 8.17 / 4.31 | 0.78 / 2.46 / 4.11 |
| | **QTC4SO** | **32.84** | **19.97** | **32.55 / 24.51 / 19.60 / 16.38** | **40.52 / 14.76 / 9.10** | **0.60 / 2.02 / 3.44** |
| Ruby | N-gram | 3.77 | 1.78 | 2.52 / 0.72 / 0.00 / 0.00 | 6.42 / 0.08 / 0.00 | 0.94 / 2.90 / 4.86 |
| | SOTitle | 27.23 | 13.55 | 27.78 / 19.71 / 15.11 / 12.13 | 21.91 / 8.66 / 3.51 | 0.78 / 2.44 / 4.09 |
| | **QTC4SO** | **32.97** | **20.02** | **32.59 / 24.30 / 19.10 / 15.50** | **40.69 / 15.33 / 8.02** | **0.59 / 2.01 / 3.52** |
| Go | N-gram | 3.97 | 1.79 | 2.67 / 0.77 / 0.38 / 0.00 | 7.03 / 0.23 / 0.00 | 0.93 / 2.87 / 4.82 |
| | SOTitle | 30.30 | 14.83 | 30.82 / 22.18 / 16.58 / 12.73 | 27.52 / 11.93 / 4.39 | 0.73 / 2.23 / 3.84 |
| | **QTC4SO** | **32.05** | **18.89** | **31.48 / 22.51 / 17.23 / 13.74** | **43.12 / 14.67 / 8.71** | **0.57 / 1.97 / 3.47** |
| HTML | N-gram | 3.38 | 1.54 | 2.43 / 0.61 / 0.18 / 0.07 | 6.56 / 0.06 / 0.01 | 0.93 / 2.89 / 4.85 |
| | SOTitle | 26.53 | 11.74 | 26.27 / 16.94 / 11.51 / 8.04 | 21.54 / 7.57 / 2.07 | 0.79 / 2.49 / 4.24 |
| | **QTC4SO** | **32.06** | **18.85** | **32.13 / 21.60 / 15.36 / 11.39** | **38.96 / 9.04 / 2.87** | **0.61 / 2.14 / 3.78** |



**Question Body**

I have a dataframe where i am trying to count the occurrence of each value. I plot it as horizontal bar but cant get it to be sorted.

```
df = pd.DataFrame( ['A','A','A','B','B','C'],columns = ['letters'])
df.value_counts()

A 3
B 2
C 1

df.letters.value_counts().plot(kind='barh')
<matplotlib.axes._subplots.AxesSubplot at 0x172737a20>
```

How can i get it sorted in a descending manner?

bool() and operator.truth() both test whether a value is *truthy* or *falsy* and they seem rather similar from the docs, it even says in the truth() docs that:

> This is equivalent to using the bool constructor.

However, truth() is over twice as fast as bool() from a simple test(Python 3.6 timings shown, but 2.7 is similar):

```
from timeit import timeit
print(timeit('bool(1)',number=10000000))
# 2.180289956042543
print(timeit('truth(1)',setup='from operator import truth',number=10000000))
# 0.7202018899843097
```

So what are the differences? Should I use truth() instead of bool()?

This Q&A arose after extensive comments and discussion with ShadowRanger under this question.

**Question Title**

Ground Truth: pandas plot value counts barplot in descending manner
SOTitle: Count occurrence of each value in pandas dataframe
N-gram: pandas plot value counts for unique strings
**QTC4SO: pandas plot value counts in descending order**

Ground Truth: What are the differences between bool() and operator.truth()?
SOTitle: Difference between truth() and operator.truth()
N-gram: What are the steps to make a menu
**QTC4SO: What are the difference between bool() and operator.truth()?**

Fig. 3. Examples of generated question titles by different approaches

**Summary for RQ2:** For the question title completion task, problem descriptions can make more contributions than code snippets. Moreover, considering bi-modal information can achieve the best performance for QTC4SO.

### *C. Result Analysis for RQ3*

**Approach.** To investigate the performance impact of different pre-trained models on QTC4SO. we further consider other three state-of-the-art pre-trained models, which have been widely used in previous text summarization studies [21]–[23].

**BERT**. BERT [30] is a pre-trained model based on the Transformer architecture, which contains a bi-directional attention mechanism. By removing the decoder layer of the transformer, BERT has achieved impressive performance in many downstream tasks (e.g., code completion [51], text classification [52], sentiment analysis [53]).

**CodeBERT.** CodeBERT [31] considers the original BERT-based architecture, but with some modifications in key parameters. CodeBERT pre-trained both natural language (NL) and programming language (PL) data from the CodeSearchNet database in a pre-training task.

**BART.** BART [32] is implemented as a sequence-to-sequence model with a bidirectional encoder for corrupted text and a left-to-right autoregressive decoder. BART is particularly efficient when fine-tuned for neural machine translation.

TABLE IV
THE PERFORMANCE OF QTC4SO WITH DIFFERENT INPUT MODALITIES

| Language | Approach | ROUGE-L (%) | GLEU (%) | BLEU_1 / 2 / 3 / 4 (%) | PP_1 / 3 / 5 (%) | LD_1 / 3 / 5 |
|---|---|---|---|---|---|---|
| Python | *w/o desc* | 15.24 | 7.53 | 16.38 / 9.64 / 6.71 / 5.26 | 22.59 / 3.84 / 2.05 | 0.77 / 2.53 / 4.36 |
| | *w/o code* | 28.79 | 16.10 | 29.20 / 20.24 / 14.92 / 11.50 | 36.48 / 11.16 / 5.45 | 0.64 / 2.15 / 3.78 |
| | **QTC4SO** | **30.95** | **17.96** | **32.09 / 23.20 / 17.85 / 14.37** | **39.52 / 13.34 / 7.03** | **0.61 / 2.07 / 3.64** |
| Java | *w/o desc* | 12.77 | 6.46 | 13.56 / 8.15 / 5.92 / 4.80 | 19.24 / 3.47 / 2.13 | 0.81 / 2.61 / 4.43 |
| | *w/o code* | 27.09 | 15.79 | 27.06 / 18.66 / 13.79 / 10.66 | 35.56 / 10.69 / 4.77 | 0.64 / 2.20 / 3.83 |
| | **QTC4SO** | **29.99** | **18.08** | **29.94 / 21.75 / 16.95 / 13.84** | **39.11 / 13.10 / 6.95** | **0.61 / 2.11 / 3.70** |
| Ruby | *w/o desc* | 17.44 | 9.26 | 16.75 / 11.28 / 8.54 / 6.97 | 23.03 / 6.30 / 3.61 | 0.77 / 2.46 / 4.19 |
| | *w/o code* | 28.11 | 16.02 | 27.89 / 19.56 / 14.36 / 10.90 | 34.69 / 11.62 / 5.35 | 0.65 / 2.18 / 3.81 |
| | **QTC4SO** | **32.97** | **20.02** | **32.59 / 24.30 / 19.10 / 15.50** | **40.69 / 15.33 / 8.02** | **0.59 / 2.01 / 3.52** |
| Go | *w/o desc* | 17.81 | 9.81 | 17.44 / 11.14 / 7.66 / 5.84 | 22.78 / 4.29 / 1.52 | 0.77 / 2.54 / 4.37 |
| | *w/o code* | 29.44 | 17.01 | 30.45 / 21.65 / 16.63 / 13.54 | 39.14 / 13.09 / 9.47 | 0.61 / 2.06 / 3.52 |
| | **QTC4SO** | **32.05** | **18.89** | **31.48 / 22.51 / 17.23 / 13.74** | **43.12 / 14.67 / 8.71** | **0.57 / 1.97 / 3.47** |

To guarantee a fair comparison, we also keep the experimental setting the same in the model fine-tuning phase.

**Results.** Here, we also show the comparison results of using different pre-trained models for four programming languages in Table V, and all results can be found on our project homepage. In this table, we can find that QTC4SO with T5 can significantly outperform the other three models in terms of all performance measures. For example, in terms of PP_1, QTC4SO with T5 can improve the performance on average for four programming languages by 55.73%, 19.05%, and 91.12%, respectively when compared to BERT, CodeBERT, and BART.

**Summary for RQ3:** Compared to the other three pre-trained models, QTC4SO with T5 can achieve the best performance.

### D. Result Analysis for RQ4

**Approach.** In this RQ, we perform a human study to evaluate the quality of the titles generated by SOTitle and our proposed approach QTC4SO. We follow the human evaluation methodology used in previous studies [21] and consider three evaluation criteria (i.e., similarity, naturalness, and informativeness).

- **Similarity**. Measuring the similarity between the generated title and the ground truth.
- **Naturalness**. Measuring the grammaticality and fluency of the generated title.
- **Informativeness**. Measuring the amount of content expressed by the generated title, which ignores its fluency.

The detailed score scheme for these criteria can be found on our project homepage. Notice the score values range from 1 to 4. The higher the score, the better the quality of the generated titles.

Then we randomly select 20 posts for each programming language in the testing set, which contains a total of 160 posts. For each post, we gather the ground-truth title and two titles generated by QTC4SO and SOTitle respectively. We invited eight graduate students, who are not co-authors and are familiar with Stack Overflow to participate in our study. Later, 40 posts of each two programming languages were manually evaluated by two graduate students according to the above-mentioned three criteria and the participants do not know which title is generated by QTC4SO. Finally, we averaged the scores of two students for each post. Moreover, the recruited students were free to use the Internet to look for related concepts that they are unfamiliar with. And we require students to evaluate only 20 posts in a half-day to ensure the quality of our human evaluation.

**Results.** The evaluation results are shown in Fig. 4. In terms of the similarity criterion, thanks to the title hints by the developers, QTC4SO can achieve higher similarity scores than SOTitle with an average score of 2.98, which means that the titles generated by QTC4SO are considered to be of good quality and can be used as titles without major modifications. In terms of the naturalness criterion, these two approaches achieve similar performance as expected, which proves that most of the titles generated by the deep learning models are considered easy to read and understand. In terms of the informativeness criterion, QTC4SO can generate more comprehensive titles than SOTitle. This shows by considering the developer's intent and using a pre-trained T5 model, QTC4SO is more capable of handling long-term dependencies in multi-modal inputs and is more advantageous in terms of semantic understanding. To assess the differences in student scoring results, we used Fleiss Kappa [54] to measure the consistency of scoring results. The overall Kappa value was 0.752, which indicates a general agreement among these students.

**Summary for RQ4:** Our human study shows that QTC4SO outperforms the baseline SOTitle by considering similarity, naturalness, and informativeness.

## V. DISCUSSION

### A. Advantages of Multi-task Learning in QTC4SO

To analyze the advantage of multi-task learning in QTC4SO, we use $QTC4SO_{single}$ to represent the control approach, which trains the models separately for each programming language. Due to the paper length limitation, Table VI shows a portion of the comparison results, and the complete results can be found on our project homepage. In this table, we can find that for each programming language, the performance of QTC4SO based on multi-task learning outperforms that

TABLE V
COMPARISON RESULTS OF USING DIFFERENT PRE-TRAINED MODELS FOR QTC4SO

| Language | Approach | ROUGE-L (%) | GLEU (%) | BLEU_1 / 2 / 3 / 4 (%) | PP_1 / 3 / 5 (%) | LD_1 / 3 / 5 |
|---|---|---|---|---|---|---|
| Python | BERT | 19.33 | 10.08 | 18.28 / 11.00 / 7.06 / 4.66 | 27.26 / 4.77 / 1.07 | 0.73 / 2.46 / 4.26 |
| | CodeBERT | 25.81 | 15.73 | 20.72 / 14.14 / 10.19 / 7.63 | 34.33 / 7.02 / 2.31 | 0.66 / 2.30 / 4.00 |
| | BART | 25.17 | 14.79 | 26.16 / 19.26 / 15.01 / 12.36 | 20.59 / 7.08 / 3.63 | 0.79 / 2.42 / 4.04 |
| | **QTC4SO (T5)** | **30.95** | **17.96** | **32.09 / 23.20 / 17.85 / 14.37** | **39.52 / 13.34 / 7.03** | **0.61 / 2.07 / 3.64** |
| Java | BERT | 15.98 | 8.31 | 15.05 / 8.64 / 5.39 / 3.48 | 25.34 / 4.00 / 0.85 | 0.75 / 2.51 / 4.33 |
| | CodeBERT | 22.85 | 13.66 | 18.05 / 11.91 / 8.29 / 6.01 | 32.61 / 5.60 / 1.50 | 0.67 / 2.36 / 4.12 |
| | BART | 23.11 | 13.71 | 23.98 / 17.71 / 14.00 / 11.55 | 20.50 / 6.94 / 3.66 | 0.80 / 2.45 / 4.06 |
| | **QTC4SO (T5)** | **29.99** | **18.08** | **29.94 / 21.75 / 16.95 / 13.84** | **39.11 / 13.10 / 6.95** | **0.61 / 2.11 / 3.70** |
| Ruby | BERT | 18.17 | 9.81 | 17.55 / 10.98 / 7.54 / 5.30 | 25.80 / 4.84 / 1.60 | 0.74 / 2.50 / 4.29 |
| | CodeBERT | 25.36 | 15.75 | 20.03 / 13.96 / 9.94 / 6.99 | 33.04 / 7.02 / 1.47 | 0.67 / 2.33 / 4.07 |
| | BART | 25.88 | 15.63 | 27.65 / 21.09 / 17.06 / 14.38 | 22.25 / 8.31 / 4.95 | 0.78 / 2.39 / 3.97 |
| | **QTC4SO (T5)** | **32.97** | **20.02** | **32.59 / 24.30 / 19.10 / 15.50** | **40.69 / 15.33 / 8.02** | **0.59 / 2.01 / 3.52** |
| Go | BERT | 19.55 | 11.08 | 17.72 / 10.69 / 7.36 / 5.34 | 25.99 / 4.51 / 1.89 | 0.74 / 2.45 / 4.26 |
| | CodeBERT | 27.28 | 17.29 | 21.98 / 14.45 / 10.06 / 7.68 | 36.54 / 5.64 / 3.03 | 0.64 / 2.29 / 4.02 |
| | BART | 25.18 | 14.88 | 25.98 / 19.27 / 14.90 / 12.25 | 21.68 / 7.67 / 4.17 | 0.78 / 2.38 / 3.94 |
| | **QTC4SO (T5)** | **32.05** | **18.89** | **31.48 / 22.51 / 17.23 / 13.74** | **43.12 / 14.67 / 8.71** | **0.57 / 1.97 / 3.47** |

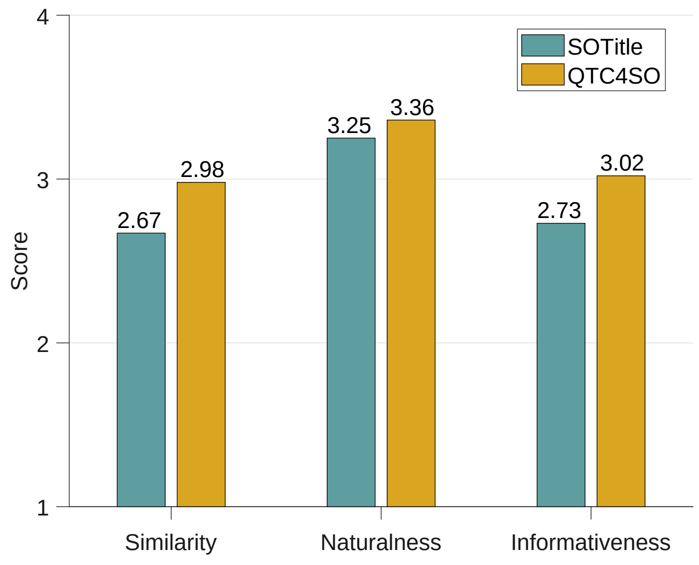


Fig. 4. The average score value of our human study by considering similarity, naturalness, and informativeness

of the models trained on the single programming language separately. For example, in terms of ROUGE-L, compared to $QTC4SO_{single}$, QTC4SO can improve the performance by at most 28.00% for eight programming languages. Moreover, we also find that using multi-task learning can significantly improve the performance of low-source programming languages (i.e., Ruby and Go). The possible reason is that $QTC4SO_{single}$ cannot effectively learn useful information due to insufficient training data of low-source programming languages. While using multi-task learning, QTC4SO can learn shared and complementary information from other programming languages with sufficient training data, thus helping to improve each other's performance.

## *B. Threats to Validity*

**Internal validity.** The first internal threat is the potential faults in the implementation of QTC4SO and baselines. To alleviate this threat, we used mature frameworks (such as PyTorch and transformers) to ensure the code implementation correctness. For the baselines, we used the scripts shared by previous work [12]. The second internal threat is related to various hyperparameter settings. There exists an excessive time consumption problem in finding an optimal hyperparameter setting for QTC4SO. However, based on our current hyperparameter setting, QTC4SO can still achieve promising performance and the performance can be further improved by hyperparameter optimization.

**External validity.** The main external threat is related to the quality of our constructed corpus. To alleviate this threat, we first select the most popular programming languages discussed in Stack Overflow to guarantee the generalization of our empirical studies. We second design four heuristic rules to filter low-quality question posts. We third split the corpus by considering the temporal relationship of posts. Finally, we design the masking operator by considering the characteristics of this task. In the future, we want to design more rules to further improve the quality of the corpus.

**Construct validity.** The main construct threat is related to our used performance measures. To evaluate the effectiveness of QTC4SO, we consider classical measures from similar tasks (such as text summarization, and code completion). Moreover, we also conducted a human study to evaluate the quality of the generated titles.

**Conclusion validity.** The main conclusion threat is related to the bias of our human study. To alleviate this threat, We first invited graduate students, which are familiar with Stack Overflow to participate in our study. We second provided a tutorial before our human study, which can ensure that all of the graduate students can understand our protocol. Finally, we use Fleiss Kappa [54] to measure the consistency among all graduate students.

TABLE VI
COMPARISON RESULTS BETWEEN $QTC4SO_{single}$ AND QTC4SO FOR DIFFERENT PROGRAMMING LANGUAGES

| Language | Approach | ROUGE-L (%) | GLEU (%) | BLEU_1 / 2 / 3 / 4 (%) | PP_1 / 3 / 5 (%) | LD_1 / 3 / 5 |
|---|---|---|---|---|---|---|
| Python | $QTC4SO_{single}$ | 30.50 | 17.68 | 31.80 / 22.76 / 17.39 / 13.98 | 38.77 / 12.87 / 6.95 | 0.61 / 2.09 / 3.66 |
| | **QTC4SO** | **30.95** | **17.96** | **32.09 / 23.20 / 17.85 / 14.37** | **39.52 / 13.34 / 7.03** | **0.61 / 2.07 / 3.64** |
| Java | $QTC4SO_{single}$ | 28.01 | 16.72 | 26.71 / 19.31 / 14.90 / 12.01 | 36.89 / 12.51 / 6.90 | 0.63 / 2.15 / 3.74 |
| | **QTC4SO** | **29.99** | **18.08** | **29.94 / 21.75 / 16.95 / 13.84** | **39.11 / 13.10 / 6.95** | **0.61 / 2.11 / 3.70** |
| Ruby | $QTC4SO_{single}$ | 27.25 | 15.08 | 26.58 / 18.46 / 13.40 / 10.02 | 35.28 / 11.14 / 4.14 | 0.65 / 2.18 / 3.82 |
| | **QTC4SO** | **32.97** | **20.02** | **32.59 / 24.30 / 19.10 / 15.50** | **40.69 / 15.33 / 8.02** | **0.59 / 2.01 / 3.52** |
| Go | $QTC4SO_{single}$ | 25.04 | 14.28 | 26.33 / 17.64 / 12.87 / 9.95 | 34.40 / 10.16 / 4.55 | 0.66 / 2.18 / 3.79 |
| | **QTC4SO** | **32.05** | **18.89** | **31.48 / 22.51 / 17.23 / 13.74** | **43.12 / 14.67 / 8.71** | **0.57 / 1.97 / 3.47** |

## VI. RELATED WORK

### A. Question Title Generation for Stack Overflow

To maintain the usefulness of Q&A websites (such as Stack Overflow), it is vital to ensure the quality of question posts. [5], [55]–[57]. In recent years, some researchers focused on the problem of question title generation for Stack Overflow, since high-quality titles are also a key factor in measuring the quality of question posts. Gao et al. [11] were the first to propose a sequence-to-sequence model, which can automatically generate question titles by analyzing the code snippets. Later, Liu et al. [12] further considered the information in the problem description and proposed a Transformer-based approach SOTitle. Zhang et al. [13] also proposed a deep learning-based question title generation model by leveraging the bi-modal information of the question body.

However, the quality of titles generated by existing approaches is still affected by the information available in the post content. A more reasonable approach is that developers can provide some tips when composing titles. Therefore, we are the first to study the problem of question title completion for Stack Overflow. To solve this problem, we build a large-scale corpus by mining Stack Overflow and then propose a novel approach QTC4SO based on the multi-modal input, multi-task learning, and the pre-trained model T5. Both automatic evaluation and human study verify the effectiveness of our study.

### B. Code Completion

In current studies of artificial intelligence for software engineering, the task most similar to our investigated problem is code completion, which suggests the next code token from a given code context. In the early studies, researchers [58], [59] utilized statistical language models to automatically learn the naturalness of source code based on a probabilistic of the occurrence of source code. However, this kind of approach cannot effectively solve the long-term dependency problem when the source code is long. Recently, researchers resorted to deep learning and code representation techniques to solve this problem. For example, Li et al. [60] designed a pointer mixture network to predict the AST (Abstract Syntax Tree) node. Kim et al. [61] proposed TravTrans, which is a Transformer-based language model that incorporates different coding styles. Liu et al. [25], [62] proposed a self-attentional neural architecture for code completion with multi-task learning. Izadi et al. [26] proposed CodeFill, which combines learned structure and naming information. This approach is based on a parallel Transformer architecture and multi-task learning.

Motivated by the code completion task, we introduce the idea of completion into the problem of question title generation for Stack Overflow and open a new direction for this research topic. Moreover, we also refer to the state-of-the-art approaches in code completion when designing QTC4SO, such as the usage of the multi-modal input, the paradigm of pre-training and fine-tuning, and multi-task learning.

## VII. CONCLUSION

This study provides a new perspective for studying automatic question title generation for Stack Overflow. Specifically, we aim to generate completion suggestions by considering tip information provided by developers when composing titles, and this task is named as question title completion. Then we propose a novel data-driven approach QTC4SO. QTC4SO adopts multi-task learning to this task for multiple programming languages and a pre-trained model T5 to automatically learn the title completion patterns based on the multi-modal inputs. To train the title completion model, we build a large-scale corpus by mining Stack Overflow. The automatic evaluation and human evaluation results show that QTC4SO can generate higher-quality titles than a state-of-the-art question title generation approach SOTitle [12]. Finally, we developed a browser plugin tool to make our proposed approach QTC4SO more practical.

**Open Science.** To support the open science community, we share our gathered corpus, scripts, trained model, detailed experimental results, and the developed plugin tool in the GitHub repository (https://github.com/QTC4SO/QTC4SO).

## ACKNOWLEDGEMENT

Yanlin Zhou and Shaoyu Yang have contributed equally to this work and they are co-first authors. The authors would like to thank the anonymous reviewers for their insightful comments and suggestions, which can substantially improve the quality of this work. This research was partially supported by the Postgraduate Research & Practice Innovation Program of Jiangsu Province (SJCX22_1616).

## References

[1] M. Choetkiertikul, D. Avery, H. K. Dam, T. Tran, and A. Ghose, "Who will answer my question on stack overflow?" in *2015 24th Australasian Software Engineering Conference*. IEEE, 2015, pp. 155–164.

[2] W. Zhu, H. Zhang, A. E. Hassan, and M. W. Godfrey, "An empirical study of question discussions on stack overflow," *Empirical Software Engineering*, vol. 27, no. 6, pp. 1–25, 2022.

[3] K. Cao, C. Chen, S. Baltes, C. Treude, and X. Chen, "Automated query reformulation for efficient search based on query logs from stack overflow," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1273–1285.

[4] S. Mondal, C. K. Saifullah, A. Bhattacharjee, M. M. Rahman, and C. K. Roy, "Early detection and guidelines to improve unanswered questions on stack overflow," in *14th Innovations in software engineering conference (formerly known as India software engineering conference)*, 2021, pp. 1–11.

[5] L. Ponzanelli, A. Mocci, A. Bacchelli, M. Lanza, and D. Fullerton, "Improving low quality stack overflow post detection," in *2014 IEEE international conference on software maintenance and evolution*. IEEE, 2014, pp. 541–544.

[6] F. Calefato, F. Lanubile, and N. Novielli, "How to ask for technical help? evidence-based guidelines for writing questions on stack overflow," *Information and Software Technology*, vol. 94, pp. 186–207, 2018.

[7] D. Correa and A. Sureka, "Fit or unfit: analysis and prediction of'closed questions' on stack overflow," in *Proceedings of the first ACM conference on Online social networks*, 2013, pp. 201–212.

[8] Y. Yao, H. Tong, T. Xie, L. Akoglu, F. Xu, and J. Lu, "Want a good answer? ask a good question first!" *arXiv preprint arXiv:1311.6876*, 2013.

[9] Y. Zhang, D. Lo, X. Xia, and J.-L. Sun, "Multi-factor duplicate question detection in stack overflow," *Journal of Computer Science and Technology*, vol. 30, no. 5, pp. 981–997, 2015.

[10] M. Ahasanuzzaman, M. Asaduzzaman, C. K. Roy, and K. A. Schneider, "Mining duplicate questions of stack overflow," in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2016, pp. 402–412.

[11] Z. Gao, X. Xia, J. Grundy, D. Lo, and Y.-F. Li, "Generating question titles for stack overflow from mined code snippets," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 4, pp. 1–37, 2020.

[12] K. Liu, G. Yang, X. Chen, and C. Yu, "Sotitle: A transformer-based post title generation approach for stack overflow," in *2022 IEEE 29th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 577–588.

[13] F. Zhang, X. Yu, J. Keung, F. Li, Z. Xie, Z. Yang, C. Ma, and Z. Zhang, "Improving stack overflow question title generation with copying enhanced codebert model and bi-modal information," *Information and Software Technology*, vol. 148, p. 106922, 2022.

[14] Z. Gao, X. Xia, D. Lo, J. Grundy, and Y.-F. Li, "Code2que: a tool for improving question titles from mined code snippets in stack overflow," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1525–1529.

[15] X. Liu, P. He, W. Chen, and J. Gao, "Multi-task deep neural networks for natural language understanding," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019, pp. 4487–4496.

[16] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, P. J. Liu *et al.*, "Exploring the limits of transfer learning with a unified text-to-text transformer." *J. Mach. Learn. Res.*, vol. 21, no. 140, pp. 1–67, 2020.

[17] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," in *Text summarization branches out*, 2004, pp. 74–81.

[18] A. Mutton, M. Dras, S. Wan, and R. Dale, "Gleu: Automatic evaluation of sentence-level fluency," in *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, 2007, pp. 344–351.

[19] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[20] V. I. Levenshtein *et al.*, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, no. 8. Soviet Union, 1966, pp. 707–710.

[21] B. Wei, Y. Li, G. Li, X. Xia, and Z. Jin, "Retrieve and refine: exemplar-based neural comment generation," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 349–360.

[22] J. Zhang, X. Wang, H. Zhang, H. Sun, and X. Liu, "Retrieval-based neural source code summarization," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 1385–1397.

[23] G. Yang, Y. Zhou, X. Chen, and C. Yu, "Fine-grained pseudo-code generation method via code feature extraction and transformer," in *2021 28th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2021, pp. 213–222.

[24] J. Li, R. Huang, W. Li, K. Yao, and W. Tan, "Toward less hidden cost of code completion with acceptance and ranking models," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 195–205.

[25] F. Liu, G. Li, B. Wei, X. Xia, Z. Fu, and Z. Jin, "A unified multi-task learning model for ast-level and token-level code completion," *Empirical Software Engineering*, vol. 27, no. 4, pp. 1–38, 2022.

[26] M. Izadi, R. Gismondi, and G. Gousios, "Codefill: Multi-token code completion by jointly learning from structure and naming sequences," *arXiv preprint arXiv:2202.06689*, 2022.

[27] A. Ciurumelea, S. Proksch, and H. C. Gall, "Suggesting comment completions for python using neural language models," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 456–467.

[28] A. Mastropaolo, E. Aghajani, L. Pascarella, and G. Bavota, "An empirical study on code comment completion," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 159–170.

[29] G. Kondrak, "N-gram similarity and distance," in *International symposium on string processing and information retrieval*. Springer, 2005, pp. 115–126.

[30] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[31] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020, pp. 1536–1547.

[32] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "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*, 2020, pp. 7871–7880.

[33] X. Hu, Q. Chen, H. Wang, X. Xia, D. Lo, and T. Zimmermann, "Correlating automated and human evaluation of code documentation generation quality," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 4, pp. 1–28, 2022.

[34] K. Bajaj, K. Pattabiraman, and A. Mesbah, "Mining questions asked by web developers," in *Proceedings of the 11th Working conference on mining software repositories*, 2014, pp. 112–121.

[35] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A comprehensive study on deep learning bug characteristics," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 510–520.

[36] T. Kudo and J. Richardson, "Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, 2018, pp. 66–71.

[37] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[38] J. Cheng, L. Dong, and M. Lapata, "Long short-term memory-networks for machine reading," in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, 2016, pp. 551–561.

[39] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," *arXiv preprint arXiv:1607.06450*, 2016.

[40] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[41] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *Advances in neural information processing systems*, vol. 27, 2014.

[42] Z. Li, Y. Wu, B. Peng, X. Chen, Z. Sun, Y. Liu, and D. Paul, "Setransformer: A transformer-based code semantic parser for code comment generation," *IEEE Transactions on Reliability*, 2022.

[43] G. Yang, Y. Zhou, X. Chen, X. Zhang, T. Han, and T. Chen, "Exploitgen: Template-augmented exploit code generation based on codebert," *Journal of Systems and Software*, vol. 197, p. 111577, 2023.

[44] G. Yang, K. Liu, X. Chen, Y. Zhou, C. Yu, and H. Lin, "Ccgir: Information retrieval-based code comment generation method for smart contracts," *Knowledge-Based Systems*, vol. 237, p. 107858, 2022.

[45] Z. Li, Y. Wu, B. Peng, X. Chen, Z. Sun, Y. Liu, and D. Yu, "Secnn: A semantic cnn parser for code comment generation," *Journal of Systems and Software*, vol. 181, p. 111036, 2021.

[46] C. Yu, G. Yang, X. Chen, K. Liu, and Y. Zhou, "Bashexplainer: Retrieval-augmented bash code comment generation based on fine-tuned codebert," in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2022, pp. 82–93.

[47] G. Yang, X. Chen, Y. Zhou, and C. Yu, "Dualsc: Automatic generation and summarization of shellcode via transformer and dual learning," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 361–372.

[48] K. Liu, G. Yang, X. Chen, and Y. Zhou, "El-codebert: Better exploiting codebert to support source code-related classification tasks," in *Proceedings of the 13th Asia-Pacific Symposium on Internetware*, 2022, pp. 147–155.

[49] C. Napoles, K. Sakaguchi, M. Post, and J. Tetreault, "Ground truth for grammatical error correction metrics," in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, 2015, pp. 588–593.

[50] L. Prechelt, "Early stopping-but when?" in *Neural Networks: Tricks of the trade*. Springer, 1998, pp. 55–69.

[51] M. Ciniselli, N. Cooper, L. Pascarella, D. Poshyvanyk, M. Di Penta, and G. Bavota, "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, 2021, pp. 108–119.

[52] X. Chen, P. Cong, and S. Lv, "A long-text classification method of chinese news based on bert and cnn," *IEEE Access*, vol. 10, pp. 34 046–34 057, 2022.

[53] R. Chandra and V. Kulkarni, "Semantic and sentiment analysis of selected bhagavad gita translations using bert-based language framework," *IEEE Access*, vol. 10, pp. 21 291–21 315, 2022.

[54] J. L. Fleiss, "Measuring nominal scale agreement among many raters." *Psychological bulletin*, vol. 76, no. 5, p. 378, 1971.

[55] J. Yang, C. Hauff, A. Bozzon, and G.-J. Houben, "Asking the right question in collaborative q&a systems," in *Proceedings of the 25th ACM conference on Hypertext and social media*, 2014, pp. 179–189.

[56] M. Duijn, A. Kucera, and A. Bacchelli, "Quality questions need quality code: Classifying code fragments on stack overflow," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 410–413.

[57] P. Arora, D. Ganguly, and G. J. Jones, "The good, the bad and their kins: Identifying questions with negative scores in stackoverflow," in *2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*. IEEE, 2015, pp. 1232–1239.

[58] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," *Communications of the ACM*, vol. 59, no. 5, pp. 122–131, 2016.

[59] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 419–428.

[60] J. Li, Y. Wang, M. R. Lyu, and I. King, "Code completion with neural attention and pointer networks," in *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, 2018, pp. 4159–25.

[61] S. Kim, J. Zhao, Y. Tian, and S. Chandra, "Code prediction by feeding trees to transformers," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 150–162.

[62] F. Liu, G. Li, B. Wei, X. Xia, Z. Fu, and Z. Jin, "A self-attentional neural architecture for code completion with multi-task learning," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 37–47.