# A Comprehensive Study of StaQC for Deep Code Summarization

Jayavardhan Reddy Peddamail
The Ohio State University
Columbus, Ohio
peddamail.1@osu.edu

Ziyu Yao
The Ohio State University
Columbus, Ohio
yao.470@osu.edu

Zhen Wang
The Ohio State University
Columbus, Ohio
wang.9215@osu.edu

Huan Sun
The Ohio State University
Columbus, Ohio
sun.397@osu.edu

## ABSTRACT

Learning the mapping between natural language (NL) and programming language, such as retrieving or generating code snippets based on NL queries and annotating code snippets using NL, has been explored by lots of research works [2, 19, 21]. At the core of these works are machine learning and deep learning models, which usually demand for large datasets of <NL, code> pairs for training. This paper describes an experimental study of StaQC [50], a large-scale and high-quality dataset of <NL, code> pairs in Python and SQL domain, systematically mined from the Stack Overflow forum (SO). We compare StaQC with two other popular datasets mined from SO on the code summarization task, showing that StaQC helps achieve substantially better results, improving the current state-of-the-art model by an order of 8% ∼ 9% in BLEU metric.

## CCS CONCEPTS

• **Information systems** → **Web searching and information discovery**; • **Software and its engineering** → **Documentation**; • **Computing methodologies** → Neural networks;

## KEYWORDS

Code Summarization; Question-Code Pairs; Web Mining; Deep Neural Networks; Stack Overflow

## 1 INTRODUCTION

Stack Overflow (SO) [41] as a website has helped software development greatly over the past few years. The understanding and re-usability of crowd sourced programming solutions can help improve the overall process of software development. Over the past few years, great research has been performed towards generation or retrieval of code snippets from a natural language description [1, 24, 28, 38, 53] and summarizing code snippets using natural language [1–3, 10, 19, 21, 47]. The underlying machine learning models for these tasks fall into the category of deep learning models [13], which are inherently data hungry. Generating large datasets containing high quality code snippets paired with a natural language description from SO becomes an important task in building many such data-hungry downstream applications for software engineering.

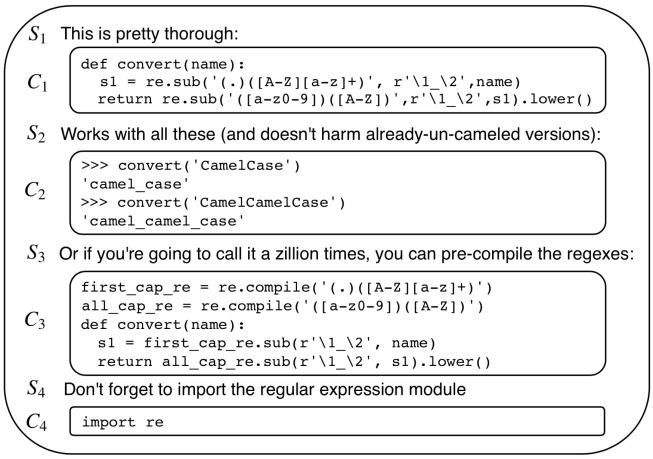



**Figure 1: The accepted answer post to question "*Elegant Python function to convert CamelCase to snake_case?*" on SO. In the figure, $S_i$ $(i = 1, 2, 3, 4)$ and $C_j$ $(j = 1, 2, 3, 4)$ denote sentence blocks and code blocks respectively [50].**

Figure 1 shows an example question "*Elegant Python function to convert CamelCase to snake_case?*" on SO and its accepted solution post. The accepted solution post contains four code snippets {C1, C2, C3, C4} but only C1 and C3 are *standalone solutions*, solely based on which the questioner is able to solve the problem [50]. In such a multi-code setting, it becomes a challenge to mine good quality <natural language question, code solution> pairs. Researchers in the past usually collected such pairs in heuristic ways: Simply pairing the question title with the first code snippet, or with each code snippet, or with the concatenation of all code snippets in the post [4, 54]. Iyer et al. [21] merely employed accepting answer posts that contain exactly one code snippet, and discarded all others with multiple code snippets. Such heuristicly mined <question, code> datasets suffer from low recall/precision problem, as they tend to either drop many useful code solutions to a question or contain noisy <question, code> pairs [50].


*KDD'18 Deep Learning Day, August 2018, London, UK*

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

In fact, multi-code answer posts are very common in SO, which makes the low-precision and low-recall issues even more prominent when we need to create large-scale <NL, code> datasets. To address these problems, Yao et al. [50] proposed a novel Bi-View Hierarchical Neural Network model (i.e., BiV-HNN) to systematically mine question-code pairs from multi-code question posts, with high precision and recall. The BiV-HNN model is able to capture features from both the textual context as well as the programming content of each code snippet, which are combined into a deep neural network architecture to predict whether this code snippet is a standalone solution or not. The model substantially outperforms heuristic approaches by more than 15% in F1 and accuracy, and is then applied to automatically mine code solutions for SO questions in Python and SQL domains, which results in *StaQC*, comprising ~**148K** Python and ~**120K** SQL <NL, code> pairs. According to Yao et al. [50], models trained on StaQC for code retrieval task [4, 21, 25] achieved impressive improvement in Mean Reciprocal Rank [46] of about 6% over the baseline dataset [21]. The experiment showed a glimpse into the potential of StaQC to improve state-of-the-art performance on downstream tasks of software engineering without considerable modifications to the model, which is an interesting research question to explore, *"Can StaQC show similar gains in performance on other downstream applications in Software Engineering?*

In this paper, we investigate the task of *Code Summarization*, which is to automatically generate a natural language summary for a given code snippet. We test StaQC [50] against CODE-NN, a heuristically mined dataset presented in [21] and CoNaLa [52], which also utilizes machine learning models to systematically collect question-code pairs from SO, on Code Summarization task. We compare the performances of two code summarization models [19, 21] for SQL and Python programming languages across different datasets. Through experiments, we show that StaQC helps achieve significantly better results on SQL code summarization, improving the current state-of-the-art model by 8% ~ 9% in BLEU metric. We also show that StaQC consistently performs better on Python code summarization, when compared to CoNaLa.

**Paper organization**. The remainder of this paper is organized as follows. Section 2 presents statistical analysis of the CODE-NN dataset, the CoNaLa dataset and the StaQC dataset. Section 3 provides brief descriptions on the code summarization models. Section 4 presents the experimental setup and results for the code summarization models. Finally, Section 5 concludes the paper and points out potential future directions.

## 2 STATISTICAL ANALYSIS OF THE DATASETS

### 2.1 CODE-NN Dataset

The CODE-NN dataset[1] is heuristically mined from SO by Iyer et al. [21], containing <NL, code> pairs of SQL and C# domains. In order for a straightforward comparison with StaQC (which covers SQL and Python domains), in this paper, we focus on the SQL dataset in CODE-NN. The authors first extracted SO question posts tagged by "sql", "database" or "oracle" from the archived Stack Exchange Dump[2] to be in SQL domain. Among them, the authors only considered questions whose accepted answer post contains exactly one code snippet. The CODE-NN dataset was then generated by pairing the question title with the one code snippet in its accepted answer post. The dataset is further processed using a semi-supervised bootstrap approach to filter titles that bear no relation to the corresponding code snippet [21]. The final cleaned SQL dataset contains **32,337** <question, code> pairs. Complete statistics of the CODE-NN dataset are provided in Table 1[3].

### 2.2 StaQC Dataset

For larger scale and higher quality, the StaQC dataset [4] is systematically mined from SO by Yao et al. [50], covering Python and SQL domain. Similar to CODE-NN, the authors identified SQL and Python posts by their tags on SO. They further cleaned them by using a supervised binary classifier to select only the "how-to-do-it" questions, since answers to other types of questions are not very likely to be standalone code solutions. From this cleaned dataset, answer posts containing exactly one code snippet are directly mined similar to CODE-NN.

To extract code solutions from multi-code answer posts (i.e., answer posts containing multiple code snippets), Yao et al. [50] proposed a novel Bi-View Hierarchical Neural Network (BiV-HNN) model. The BiV-HNN model consists of two different modules that capture features from the textual contexts and the code content of a code snippet, and combines them into a deep neural network architecture, which finally predicts whether a code snippet is a standalone solution or not. The model was shown to outperform both the widely adopted heuristic methods (e.g., pairing the question title with the first code snippet or all code snippets in its answer post) and traditional classifiers (e.g., Logistic Regression and Support Vector Machines) by more than **15%** higher F1 and accuracy in identifying code solutions. The final mined StaQC dataset contains **119,519** SQL domain and **147,546** Python domain <question, code> pairs, making it the largest-to-date dataset for SQL Domain. Figure 2 shows an example of two code solutions in the StaQC dataset, which are mined from one multi-code SQL answer post. Complete statistics of the StaQC dataset are provided in Table 1.

### 2.3 CoNaLa Dataset

Similar to StaQC, CoNaLa [52][5] is also extracted by a machine learning model from SO, resulting in the largest-to-date dataset of **598,237** question-code pairs in Python domain. As in the two previous work [21, 50], Yin et al. [52] collected SO Python questions and filtered down to only "how-to" questions from them. Different from [50], they considered every line (or fragment) in a code block as a candidate code solution, and built a logistic regression classifier to decide whether the candidate aligns well with the question title or not. The classifier utilizes two kinds of features to capture both the syntactic and the semantic information in a code candidate, leading

[1] Available at https://github.com/sriniiyer/codenn

[2] https://archive.org/details/stackexchange

[3] To have a direct comparison, code snippets are all tokenized by the preprocessing steps mentioned in Section 4.2 and 4.3. Therefore, the dataset statistics are different from the reported statistics in [21, 50].

[4] Available at https://github.com/LittleYUYU/StackOverflow-Question-Code-Dataset

[5] Available at https://conala-corpus.github.io/.

**Table 1: Statistics of datasets.**

| Programming Language | Dataset | # of QC pairs | Question | | Code | |
|---|---|---|---|---|---|---|
| | | | Average length | # of tokens | Average Length | # of tokens |
| SQL | CODE-NN | 32,337 | 9 | 10,086 | 27 | 141,018 |
| | StaQC | 119,519 | 9 | 35,722 | 37 | 740,837 |
| Python | CoNaLa | 598,237 | 9 | 25,582 | 8 | 394,777 |
| | StaQC | 147,546 | 9 | 59,906 | 39 | 1,171,117 |

**Question:** Can I use TOP and Count in a single SELECT of SQL?

**Code solution 1**

```
select top 5 count(*) over () as cnt, column1, max(column2),
min(column3) from myTable
group by column1
order by column1;
```

**Code solution 2**

```
select top 5 *
from (select count(*) over () as cnt, column1, max(column2),
min(column3) from myTable
group by column1) t
order by column1;
```

**Figure 2: An example of two code solutions in StaQC [50], which were mined from a multi-code answer post.**

to much better recall (precision) at the same level of precision (recall) as the heuristic approaches. Complete statistics of the CoNaLa dataset are provided in Table 1.

## 2.4 Question-Code Distribution of the Datasets

StaQC and CoNaLa enjoy great diversity in the sense that they contain multiple code solutions for a question. We count the number of code solutions for each question and show the "question-# of code" distribution for each dataset in Table 2. It is observed that about 15% of the SQL questions in StaQC have multiple code solutions, while the same statistic for CODE-NN is only 0.08%. StaQC achieves such rich diversity due to the BiV-HNN model, which analyzes different <question, code> pairs from each multi-code post and can identify code solutions with high precision. To verify, we present the statistics of StaQC-multi, which is the subset of StaQC mined solely from SO multi-code answer posts. About 57% of the SQL questions and 42% of the Python questions in StaQC-multi datasets contain multiple code solutions and 9% of the questions have more than 2 solutions in both SQL and Python. Due to the mining of all fragments in each code snippet, the CoNaLa dataset shows even greater diversity, around 87% of which contain more than 3 solutions. This "question-# of code" distribution for StaQC and CoNaLa reflects the advantage of systematical mining in terms of contribution to dataset diversity.

A dataset with rich surface variations is beneficial for the development of complex deep learning models [13]. When a model does not observe certain data patterns in the training phase, it becomes less capable to predict them during testing. StaQC can alleviate this issue by enabling a model to learn from alternative code solutions to the same question. Owing to its large scale and diversity, Yao et al. [50] showed that a model trained on StaQC dataset can outperform the one trained on the CODE-NN dataset by **6%** Mean Reciprocal Rank [46] on the code retrieval task. In this paper, we further demonstrate the strength of StaQC through experiments on the code summarization task.

# 3 CODE SUMMARIZATION MODELS

## 3.1 Background

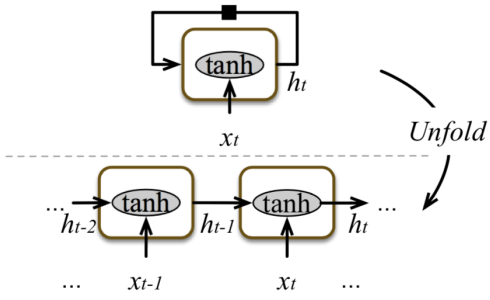
Code Summarization is a task to generate a natural language summary given a code snippet. Current state-of-the-art architectures for code summarization model are built upon the recent advancements in deep recurrent neural network (RNN) [12] architectures. Specifically, Long Short-Term Memory (LSTM) [11, 18] has become the core component of many tasks involving sequential data due to its capability to learn long-term dependencies. It has also been used to better handle long dependencies in source code (e.g., a Python function is used far away from its definition) [19]. The details of RNN and LSTM are shown in Figure 3a and 3b respectively.

*3.1.1* ***Recurrent Neural Networks.*** RNNs are intimately related to sequences and lists because of their chain-like architecture. As illustrated in Figure 3a, At each time step $t$, the RNN cell takes as input, the current input token as well as the previous hidden state outputted by its previous time step $t-1$ and updates the current hidden state namely, $h_t = tanh(Wx_t + Uh_{t-1} + b)$ where $W$, $U$, and $b$ are the trainable parameters and $tanh$ is the activation function [19] . Though, in theory, RNNs are capable to capture long-distance dependencies, in practice, they fail due to the gradient vanishing/exploding problems [6, 36].

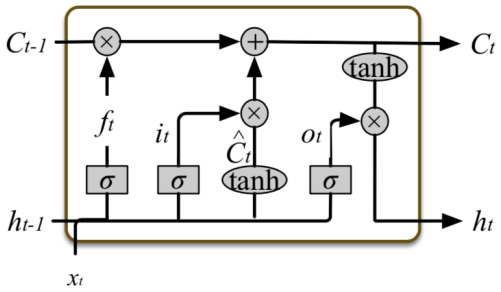
*3.1.2* ***Long Short-Term Memory.*** LSTM introduces a structure called "memory cell" to solve the problem. Basically, a LSTM unit is composed of three multiplicative gates which control the proportions of information to forget and to pass on to the next time step. Figure 3b gives the basic structure of an LSTM unit. The LSTM is trained to selectively "forget" information from the hidden states, thus allowing room to take in more important information [18].

**Table 2: Question - # of code distributions.**

| | Programming Language | Dataset | # of code solutions | | | |
|---|---|---|---|---|---|---|
| | | | 1 | 2 | 3 | >3 |
| # of Questions | SQL | CODE-NN | 25,625 | 16 | 2 | 2 |
| | | StaQC | 86,788 | 12,536 | 2,003 | 455 |
| | | StaQC-multi | 11,266 | 12,333 | 1,992 | 449 |
| | Python | CoNaLa | 2,264 | 1,233 | 1,832 | 35,178 |
| | | StaQC | 108,653 | 13,300 | 2,807 | 892 |
| | | StaQC-multi | 23,403 | 13,291 | 2,800 | 891 |




**(a) Standard RNN model and its unfolded architecture through time steps.**




**(b) Basic LSTM Cell.**

**Figure 3: An illustration of the standard RNN and LSTM [19].**

LSTM has been widely used to solve semantically related tasks like speech recognition [14], sequence tagging [9, 20, 32], machine translation [5, 48] and etc., leading to state-of-the-art performances.

*3.1.3* ***Language Models.*** Language models have been successfully applied to solve a variety of problems in NLP, e.g., machine translation [5, 7, 16, 31], speech recognition [8, 29], named entity recognition [17, 27, 37] and question answering [51]. Language models are trained to perceive language patterns from its training corpus. Specifically, given a sequence of words $x = (x_1, x_2, ..., x_n)$ in a sentence, a language model estimates the probability of this sentence by computing a product of conditional probabilities of predicting the next word based on the prior words, as shown in Eq 1.

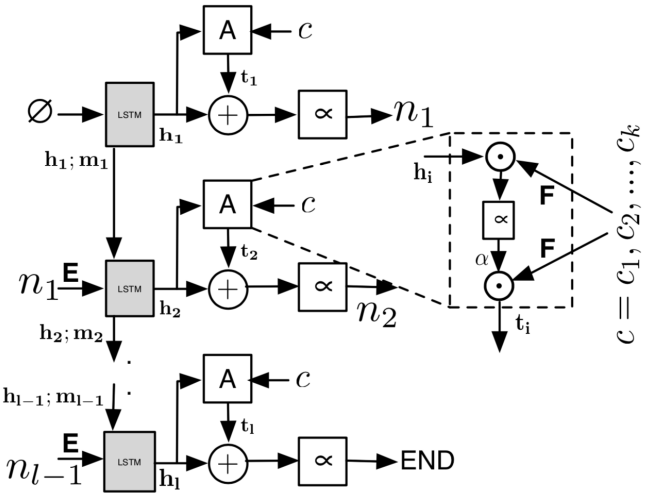



**Figure 4: The Codenn model [21] that generates a NL summary $n = n_1, ...,$ END given code snippet $c_1, ..., c_k$. The vector $t_i$ encodes the code snippet through an attention mechanism based on the current LSTM hidden state $h_i$. The two vectors are then combined to generate the next word $n_i$, which will be taken as input and fed into the next LSTM cell. The $\propto$ blocks denote softmax operations.**

LSTM's have become an important building block for language models due to their capability to remember long range dependencies [23, 26, 33, 42, 43].

$$P(x) = P(x_1)P(x_2|x_1)...P(x_n|x_1...x_{n-1}) \quad (1)$$

## 3.2 Codenn

Codenn summarization model, published in [21], uses an end-to-end generation system to perform content selection and surface realization jointly. The core component of Codenn is a LSTM-based recurrent neural network with an attention mechanism [14, 30], which models the probability of a natural language summary conditioned on the given code snippet, as shown in Figure 4. Formally, given a NL summary $n = n_1, ..., n_l$, each word $n_i$ is represented by

a 1-hot vector $n_i \in \{0,1\}^{|N|}$, where $N$ is the vocabulary size. The model computes the probability of $n$ as a product of the conditional next-word probabilities as shown in Eq 2:

$$s(c,n) = \prod_{i=1}^{l} p(n_i|n_1,...,n_{i-1}) \tag{2}$$

with

$$p(n_i|n_1,...,n_{i-1}) \propto \mathbf{W}\tanh(\mathbf{W_1 h_i} + \mathbf{W_2 t_i}) \tag{3}$$

where $\mathbf{W} \in \mathbb{R}^{|N|xH}$ and $\mathbf{W_1}, \mathbf{W_2} \in \mathbb{R}^{HxH}$ are trainable parameters, and $H$ is the embedding dimensionality of the summaries. $\mathbf{h_i}$ is the LSTM hidden state at time step $i$. $\mathbf{t_i}$ is the contribution from the attention model on the source code. The generation of each word is guided by a global attention model [30], which computes a weighted sum of the embeddings of the code snippet tokens based on the current LSTM state. A code snippet $c$ is represented as a set of 1-hot vectors $c_1,...,c_k \in 0,1^{|C|}$, where C is the vocabulary of all tokens in the code snippets. The attention model computes,

$$\mathbf{t_i} = \sum_{j=1}^{k} \alpha_{i,j} \cdot \mathbf{c_j F} \tag{4}$$

where $\mathbf{F} \in \mathbb{R}^{|C|xH}$ is a token embedding matrix and each $\alpha_{i,j}$ is the attention weight for each code token $\mathrm{c_j}$ w.r.t current hidden state $\mathrm{h_i}$:

$$\alpha_{i,j} = \frac{\exp(\mathrm{h_i}^{\mathrm{T}}\mathrm{c_j F})}{\sum_{j=1}^{k}\exp(\mathrm{h_i}^{\mathrm{T}}\mathrm{c_j F})} \tag{5}$$

### 3.3 DeepCom

DeepCom [19] is built upon advances in Neural Machine Translation (NMT). Typical NMT aims to automatically translate from one language to another language [5, 44]. Intuitively, Hu et al. [19] considered generating NL summaries or comments as a variant of the NMT problem, where source code written in a programming language needs to be translated to text in natural language. Compared with Codenn which only builds a LSTM based language model for NL summaries, the NMT model builds language models for both source code and summaries.

Specifically, DeepCom models both the code snippets and the text summaries as sequences, and uses the Sequence-to-Sequence (Seq2Seq) approach to learn the translation between them. Seq2Seq has been widely used for machine translation [44], text summarization [39], dialogue system [45], etc. The model consists of three components: a LSTM encoder, a LSTM decoder, and an attention component. Figure 5 illustrates the detailed Seq2Seq model.

At each time step $t$ , the encoder reads one token $x_t$ of the source code sequence, then updates and records the current hidden state $s_t$. The source code features are finally encoded into a context vector $c$ through an attention mechanism [5]. Specifically, DeepCom defines individual $c_i$ for predicting each target word $i$ as a weighted sum of all hidden states $s_1,..,s_m$ in encoder:

$$c_i = \sum_{j=1}^{m} \alpha_{ij} s_j \tag{6}$$

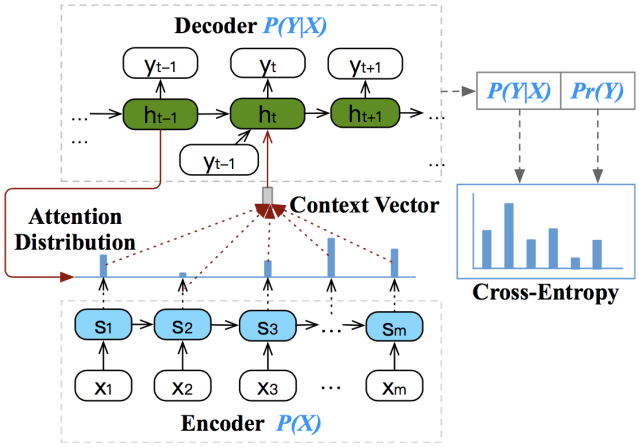



**Figure 5: The DeepCom model [19] with a Sequence-to-Sequence modeling.**

The weight $\alpha_{ij}$ of each hidden state $s_j$ is computed as

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{m}\exp(e_{ik})} \tag{7}$$

and

$$e_{ij} = a(h_{i-1}, s_j) \tag{8}$$

is an alignment model which scores how well the inputs around position $j$ and the output at position $i$ match. The decoder aims to generate the target summary by sequentially predicting the probability of a word $y_i$ conditioned on the context vector $c_i$ and its previous generated words $y_1,...,y_{i-1}$ i.e.,

$$p(y_i|y_1,...,y_{i-1},x) = g(y_{i-1}, h_i, c_i) \tag{9}$$

where $g$ is used to estimate the probability of the word $i$ [19].

Other ideas published in [19] include using Abstract Syntax Trees (AST) to capture structure in code. The authors propose a new AST traversal method (namely structure-based traversal) and a domain-specific method to deal with out-of-vocabulary tokens better in ASTs. However, since most of the code snippets on SO are not directly parsable [21, 49], we do not perform AST parsing and simply traverse each code snippet line by line into a sequence, similar to the experiment performed in [19].

## 4 EXPERIMENTAL SETUP AND RESULTS

### 4.1 Evaluation

We measure the quality of the generated code summaries by the BLEU-4 metric [35]. BLEU score has been the widely-used accuracy measure for NMT [5] and has also been used in software tasks evaluation [15, 22]. It calculates the similarity between the generated sequence and reference sequence. BLEU score has proven to be a good measure of accuracy for generated sequences [15, 22], so we adopt it has the main metric for model evaluation as in [19, 21].

BLEU uses a modified form of precision to compare a candidate sequence against multiple reference sequences. BLEU measures the

average n-gram precision on a set of reference sentences, with a penalty for overly short sentences. The score is computed as:

$$BLEU = BP \cdot \exp(\sum_{n=1}^{N} w_n \log(p_n)) \quad (10)$$

where $p_n$ is the ratio of length $n$ subsequences in the candidate that are also in the reference. For BLEU-4, we set N to 4, which is the maximum number of grams. BP is brevity penalty,

$$BP = \begin{cases} 1 & if \quad c > r \\ e^{(1-r/c)} & if \quad c \leq r \end{cases} \quad (11)$$

where $c$ is the length of the candidate translation and $r$ is the effective reference sequence length.

## 4.2 SQL-Domain Experimental Setup

For the experiments on the Codenn model and the DeepCom model for SQL domain, we keep the implementation details exactly the same as mentioned in [21] and [19], only changing the training dataset for each experimental setting. In the first and second setting, we use CODE-NN and StaQC (SQL) datasets as training data respectively. To emphasize the importance of surface variation in StaQC, we just added the **41,826** <natural language question, code solution> pairs, automatically mined from Stack Overflow SQL multi-code answer posts, to the CODE-NN dataset (i.e., CODENN-multi = CODE-NN + StaQC-multi), which becomes the training data in our third experimental setting. For all SQL domain experiments, we used the DEV (valid) set and the EVAL (test) set in [21] for development and evaluation, respectively. All questions and code snippets occurring in the these two sets were removed from training data.

For the experiments on the Codenn model, all the hyper-parameters are kept the same as mentioned in [21], except for the dropout rate [40], which was chosen from {0.4, 0.7} for each experimental setting, to avoid over-fitting and achieve a better performance. For the experiments on DeepCom, almost all the hyper-parameters are as mentioned in [19] except for the LSTM hidden dimension, which was chosen from {128,512}, and the dropout rate [40], which was chosen from {0.4, 0.7} for each experimental setting. Hu et al. [19] use a vocabulary size of 30,000 for code snippets as well as NL summaries. Instead, we experimented with vocabulary sizes of {30,000, 50,000} for code snippets, in order to accommodate the considerable difference in token numbers of code and summaries, as shown in Table 1. We used simple "space" tokenization to create vocabulary for both code snippets and natural language.[6] The best model was selected as the one achieving the highest BLEU score on DEV sets.

## 4.3 Python-Domain Experimental Setup

We experimented with the DeepCom model [19] for Python domain. In the first and second experimental settings, we use StaQC-python and CoNaLa datasets as training data respectively. As the CoNaLa dataset (~600k) is more than ×3 times the size of StaQC Python dataset (~148k), we perform a third experiment using CoNaLa-reduced dataset (~148k), which is a random subset of CoNaLa and has the same size as the StaQC-python dataset, in order to make a direct comparison. We randomly sampled 10% of the CoNaLa dataset as the development (DEV) set across all experiments. We test all models with the CoNaLa test set [52] (denoted as "EVAL"). All questions and code snippets occurring in the DEV and EVAL set were removed from the training data.

For all experiment, we set the the maximum length of a code sequence to 100 and the maximum question length to 30. The vocabulary sizes for code and natural language questions are set to 50,000 and 30,000 respectively. We used simple "space" tokenization to create vocabulary for both code and natural language. We experimented with different hyper-parameters: Number of LSTM layers {2, 3}, hidden dimension {128, 256, 512} and dropout rate {0.3, 0.8} in each setting. The best model was selected as the one achieving the highest BLEU score on the DEV set.

[6] Although we use a simpler data processing method, we got comparable experimental results as the reported in [19].

**Table 3: Performance of different SQL code summarization models across different experimental settings on the EVAL set. Performance on DEV is indicated in parentheses.**

| Model | Dataset | BLEU-4 score(%) |
|---|---|---|
| Codenn | CODE-NN | 19.64 (21.42) |
| | StaQC | 21.41 **(21.67)** |
| | CODENN-multi | **21.44** (21.37) |
| DeepCom | CODE-NN | 32.55 (32.91) |
| | StaQC | **40.95 (41.44)** |
| | CODENN-multi | 40.02 (40.36) |

**Table 4: Performance of DeepCom Python code summarization model across different experimental settings on the EVAL set. Performance on DEV is indicated in parentheses.**

| Dataset | Size of Training Data | BLEU-4 score(%) |
|---|---|---|
| CoNaLa | 481,872 | 43.95 (**40.16**) |
| CoNaLa-reduced | 145,353 | 42.07 (38.50) |
| StaQC-python | 145,353 | **44.49** (40.15) |

## 4.4 Experimental Results

Table 3 shows the BLEU score of each SQL Code summarization model on EVAL and DEV set across different experimental settings. Across different model architectures, we can consistently observe that models trained on StaQC and CODENN-multi showed improved performance over models trained on CODE-NN. For the Codenn summarization model, we can observe an improvement of 2% in BLEU metric over the CODE-NN dataset. Specifically for DeepCom, StaQC helps achieve a substantial 8% ~ 9% improvement in BLEU metric on the current state-of-the-art model [19].

By comparing the results of experimental setting 1 (using the CODE-NN dataset) to experimental setting 3 (using the CODENN-multi dataset), we can clearly see the importance of the mined multi-code questions. The rich surface variations in the mined multi-code questions helps improve the performance across all models. Note that the performance gains shown here are still conservative,

since we adopted almost the same hyper-parameters and a small evaluation set, in order to see the direct impact of StaQC. Using more challenging evaluation sets and by conducting systematic hyper-parameter selection, we expect models trained on StaQC to be more advantageous.

Table 4 presents the BLEU scores for different experiments on the Python Code Summarization task. The DeepCom model trained on StaQC [50] consistently performs similar or slightly better than the model trained on CoNaLa [52], even though CoNaLa is ×3 size of StaQC. We further make a direct comparison between StaQC and CoNaLa-reduced, which are of the same size, and observe an improvement of ∼ 2.5% in BLEU metric provided by StaQC over CoNaLa-reduced. In the future, we plan to conduct more qualitative study over different experiments.

## 5 CONCLUSION

This paper explores an experimental study of StaQC [50], a large-scale and high-quality dataset of <natural language question, code snippet> pairs in Python and SQL domain, in comparison with two other existing datasets, CODE-NN [21] and CoNaLa [52]. We show that the systematically mined StaQC can greatly help downstream tasks aiming to associate natural language with programming language, by performing experiments on the code summarization task. In the future, we plan to perform more qualitative analysis across these datasets and also perform more experiments on other downstream code-language tasks like code generation.

## ACKNOWLEDGMENTS

This research was sponsored in part by the Army Research Office under cooperative agreements W911NF-17-1-0412, Fujitsu gift grant, and Ohio Supercomputer Center [34]. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Office or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notice herein.

## REFERENCES

[1] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2017. A Survey of Machine Learning for Big Code and Naturalness. *arXiv preprint arXiv:1709.06182* (2017).

[2] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning*. 2091–2100.

[3] Miltos Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. 2015. Bimodal Modelling of Source Code and Natural Language. In *Proceedings of the 32nd International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Francis Bach and David Blei (Eds.), Vol. 37. PMLR, Lille, France, 2123–2132. http://proceedings.mlr.press/v37/allamanis15.html

[4] Miltos Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. 2015. Bimodal modelling of source code and natural language. In *International Conference on Machine Learning*. 2123–2132.

[5] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).

[6] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. 1994. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks* 5, 2 (1994), 157–166.

[7] Thorsten Brants, Ashok C Popat, Peng Xu, Franz J Och, and Jeffrey Dean. 2007. Large language models in machine translation. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*.

[8] Ciprian Chelba, Dan Bikel, Maria Shugrina, Patrick Nguyen, and Shankar Kumar. 2012. Large scale language modeling in automatic speech recognition. *arXiv preprint arXiv:1210.8440* (2012).

[9] Jason PC Chiu and Eric Nichols. 2015. Named entity recognition with bidirectional LSTM-CNNs. *arXiv preprint arXiv:1511.08308* (2015).

[10] Jaroslav Fowkes, Pankajan Chanthirasegaran, Razvan Ranca, Miltiadis Allamanis, Mirella Lapata, and Charles Sutton. 2016. TASSAL: Autofolding for source code summarization. In *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, 649–652.

[11] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. 1999. Learning to forget: Continual prediction with LSTM. (1999).

[12] Christoph Goller and Andreas Kuchler. 1996. Learning task-dependent distributed representations by backpropagation through structure. In *Neural Networks, 1996., IEEE International Conference on*, Vol. 1. IEEE, 347–352.

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. The MIT Press.

[14] Alex Graves. 2013. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850* (2013).

[15] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 631–642.

[16] Di He, Yingce Xia, Tao Qin, Liwei Wang, Nenghai Yu, Tieyan Liu, and Wei-Ying Ma. 2016. Dual learning for machine translation. In *Advances in Neural Information Processing Systems*. 820–828.

[17] Djoerd Hiemstra. 2001. Using language models for information retrieval. (2001).

[18] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[19] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep Code Comment Generation. In *Proceedings of the 2017 26th IEEE/ACM International Conference on Program Comprehension*. ACM.

[20] Zhiheng Huang, Wei Xu, and Kai Yu. 2015. Bidirectional LSTM-CRF models for sequence tagging. *arXiv preprint arXiv:1508.01991* (2015).

[21] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Vol. 1. 2073–2083.

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

[23] Rafal Jozefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. 2016. Exploring the limits of language modeling. *arXiv preprint arXiv:1602.02410* (2016).

[24] Neel Kant. 2018. Recent Advances in Neural Program Synthesis. *CoRR* abs/1802.02353 (2018). arXiv:1802.02353 http://arxiv.org/abs/1802.02353

[25] Iman Keivanloo, Juergen Rilling, and Ying Zou. 2014. Spotting Working Code Examples. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 664–675. https://doi.org/10.1145/2568225.2568292

[26] Yoon Kim, Yacine Jernite, David Sontag, and Alexander M Rush. 2016. Character-Aware Neural Language Models.. In *AAAI*. 2741–2749.

[27] Onur Kuru, Ozan Arkan Can, and Deniz Yuret. 2016. Charner: Character-level named entity recognition. In *Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: Technical Papers*. 911–921.

[28] Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D. Ernst. 2018. NL2Bash: A Corpus and Semantic Parser for Natural Language Interface to the Linux Operating System. *CoRR* abs/1802.08979 (2018). arXiv:1802.08979 http://arxiv.org/abs/1802.08979

[29] Xunying Liu, James L Hieronymus, Mark JF Gales, and Philip C Woodland. 2013. Syllable language models for Mandarin speech recognition: Exploiting character language models. *The Journal of the Acoustical Society of America* 133, 1 (2013), 519–528.

[30] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. 2015. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025* (2015).

[31] Thang Luong, Michael Kayser, and Christopher D Manning. 2015. Deep neural language models for machine translation. In *Proceedings of the Nineteenth Conference on Computational Natural Language Learning*. 305–309.

[32] Xuezhe Ma and Eduard Hovy. 2016. End-to-end sequence labeling via bi-directional lstm-cnns-crf. *arXiv preprint arXiv:1603.01354* (2016).

[33] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. 2017. Regularizing and optimizing LSTM language models. *arXiv preprint arXiv:1708.02182* (2017).

[34] Ohio Supercomputer Center. 1987. Ohio Supercomputer Center. http://osc.edu/ark:/19495/f5s1ph73.

[35] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*. Association for Computational Linguistics, 311–318.

[36] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. 2013. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*. 1310–1318.

[37] Matthew E Peters, Waleed Ammar, Chandra Bhagavatula, and Russell Power. 2017. Semi-supervised sequence tagging with bidirectional language models. *arXiv preprint arXiv:1705.00108* (2017).

[38] Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract Syntax Networks for Code Generation and Semantic Parsing. *CoRR* abs/1704.07535 (2017). arXiv:1704.07535 http://arxiv.org/abs/1704.07535

[39] Alexander M Rush, Sumit Chopra, and Jason Weston. 2015. A neural attention model for abstractive sentence summarization. *arXiv preprint arXiv:1509.00685* (2015).

[40] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research* 15, 1 (2014), 1929–1958.

[41] Stack Overflow. 2018. Stack Overflow. https://stackoverflow.com/.

[42] Martin Sundermeyer, Hermann Ney, and Ralf Schlüter. 2015. From feedforward to recurrent LSTM neural networks for language modeling. *IEEE Transactions on Audio, Speech, and Language Processing* 23, 3 (2015), 517–529.

[43] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. 2012. LSTM neural networks for language modeling. In *Thirteenth Annual Conference of the International Speech Communication Association*.

[44] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*. 3104–3112.

[45] Oriol Vinyals and Quoc Le. 2015. A neural conversational model. *arXiv preprint arXiv:1506.05869* (2015).

[46] Ellen M Voorhees et al. 1999. The TREC-8 Question Answering Track Report.. In *Trec*, Vol. 99. 77–82.

[47] Xiaoran Wang, Yifan Peng, and Benwen Zhang. 2018. Comment Generation for Source Code: State of the Art, Challenges and Opportunities. *CoRR* abs/1802.02971 (2018). arXiv:1802.02971 http://arxiv.org/abs/1802.02971

[48] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).

[49] Di Yang, Aftab Hussain, and Cristina Videira Lopes. 2016. From query to usable code: An analysis of Stack Overflow code snippets. In *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, 391–402.

[50] Ziyu Yao, Daniel S Weld, Wei-Peng Chen, and Huan Sun. 2018. StaQC: A Systematically Mined Question-Code Dataset from Stack Overflow. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 1693–1703.

[51] Jun Yin, Xin Jiang, Zhengdong Lu, Lifeng Shang, Hang Li, and Xiaoming Li. 2015. Neural generative question answering. *arXiv preprint arXiv:1512.01337* (2015).

[52] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to Mine Aligned Code and Natural Language Pairs from Stack Overflow. In *International Conference on Mining Software Repositories (MSR)*. ACM, 476–486. https://doi.org/10.1145/3196398.3196408

[53] Pengcheng Yin and Graham Neubig. 2017. A Syntactic Neural Model for General-Purpose Code Generation. *CoRR* abs/1704.01696 (2017). arXiv:1704.01696 http://arxiv.org/abs/1704.01696

[54] Meital Zilberstein and Eran Yahav. 2016. Leveraging a corpus of natural language descriptions for program similarity. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. ACM, 197–211.