StaQC: A Systematically Mined Question-Code Dataset from Stack Overflow

Ziyu Yao[†], Daniel S. Weld[#], Wei-Peng Chen[††], Huan Sun[†]

[†]The Ohio State University, [#]University of Washington, [††]Fujitsu Labs of America
{yao.470,sun.397}@osu.edu,weld@cs.washington.edu,wei-peng.chen@us.fujitsu.com

ABSTRACT

Stack Overflow (SO) has been a great source of natural language questions and their code solutions (i.e., question-code pairs), which are critical for many tasks including code retrieval and annotation. In most existing research, question-code pairs were collected *heuristically* and tend to have low quality. In this paper, we investigate a new problem of *systematically mining* question-code pairs from Stack Overflow (in contrast to *heuristically collecting* them). It is formulated as predicting whether or not a code snippet is a standalone solution to a question. We propose a novel Bi-View Hierarchical Neural Network which can capture both the programming content and the textual context of a code snippet (i.e., two views) to make a prediction. On two manually annotated datasets in Python and SQL domain, our framework substantially outperforms heuristic methods with at least **15%** higher F_1 and accuracy. Furthermore, we present StaQC (Stack Overflow Question-Code pairs), the largest dataset to date of ~**148K** Python and ~**120K** SQL question-code pairs, automatically mined from SO using our framework. Under various case studies, we demonstrate that StaQC can greatly help develop data-hungry models for associating natural language with programming language[1].

CCS CONCEPTS

• **Information systems** → **Web searching and information discovery**; **Question answering**; **Information extraction**; • **Software and its engineering** → *Automatic programming*;

KEYWORDS

Natural Language Question Answering; Question-Code Pairs; Deep Neural Networks; Stack Overflow

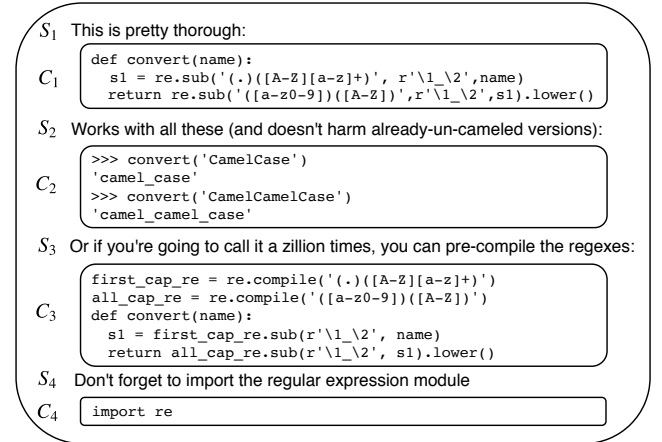[1]Available at https://github.com/LittleYUYU/StackOverflow-Question-Code-Dataset.

Figure 1: The accepted answer post to question *"Elegant Python function to convert CamelCase to snake_case?"* in SO. S_i ($i = 1, 2, 3, 4$) and C_j ($j = 1, 2, 3, 4$) denote sentence blocks and code blocks respectively, which can be trivially separated based on the HTML format.

1 INTRODUCTION

Online forums such as Stack Overflow (SO) [40] have contributed a huge number of code snippets, understanding and reuse of which can greatly speed up software development. Towards this goal, a lot of research work have been developed recently, such as retrieving or generating code snippets based on a natural language query, and annotating code snippets using natural language [2, 21, 28, 44, 52, 61, 64]. At the core of these work are machine learning models that map between natural language and programming language, which are typically data-hungry [19, 24, 46] and require large-scale and high-quality <natural language question, code solution> pairs (i.e., question-code or QC pairs).

In our work, *we define a code snippet as a code solution when the questioner can solve the problem solely based on it* (also named as "standalone" solution). Take Figure 1 as an example, which shows the accepted answer post[2] to question "*Elegant Python function to convert CamelCase to snake_case*". Among the four code snippets {C_1, C_2, C_3, C_4}, only C_1 and C_3 are standalone code solutions to

[2]In SO, an accepted answer post is marked with a green check by the questioner, if he/she thinks it solves the problem. Following previous work [21, 59], although there can be multiple answer posts to a question, *we only consider the accepted one because of its verified quality*, and use "accepted answer post" and "answer post" interchangeably.

the question while the rest are not, because C_2 only gives an input-output demo of the "convert" function without its definition and C_4 is a reminder of an additional detail. Given an answer post with multiple code snippets (i.e., a multi-code answer post) like Figure 1, previous work usually collected question-code pairs in heuristic ways: Simply pair the question title with the first code snippet, or with each code snippet, or with the concatenation of all code snippets in the post [2, 64]. Iyer et al. [21] merely employed accepted answer posts that contain exactly one code snippet, and discarded all others with multiple code snippets. Such heuristic question-code collection methods suffer from at least one of the following weaknesses: (1) Low precision: Questions do not match with their paired code snippets, when the latter serve as background, explanation, or input-output demo rather than as a solution (e.g., C_2 in Figure 1); (2) Low recall: If one only selects the first code snippet to pair with a question, other code solutions in an answer post (e.g., C_3) will be unemployed.

In fact, multi-code answer posts are very common in SO, which makes the low-precision and low-recall issues even more prominent. In the Stack Exchange Data dump[51], among all accepted answer posts for Python and SQL "how-to-do-it" questions (to be introduced in Section 2), 44.66% and 34.35% contain more than one code snippets respectively. Note that an accepted answer post was verified only *as an entirety* by the questioner, and labels on whether each individual code snippet serves as a standalone solution or not are not readily available. Moreover, it is not feasible to obtain such labels by simply running each code snippet in a programming environment for two reasons: (1) A runnable code snippet is not necessarily a code solution (e.g., C_4 in Figure 1); (2) It was reported that around 74% of Python and 88% of SQL code snippets in SO are not directly parsable or runnable [21, 59]. Nevertheless, many of them usually contain critical information to answer a question. Therefore, they can still be used in semantic analysis for downstream tasks [2, 21, 59, 64] once paired with natural language questions.

To systematically mine question-code pairs with high precision and recall, we propose a novel task: *Given a question[3] in SO and its accepted answer post with multiple code snippets, how to predict whether each code snippet is a standalone solution or not?* In this paper, we focus on "how-to-do-it"-type of questions which ask how to implement a certain task like in Figure 1, since answers to such questions are most likely to be standalone code solutions. The definition and classification of different types of questions will be discussed in Section 2. We identify two challenges in our task: (1) As shown in Figure 1, code snippets in an answer post can play many non-solution roles such as serving as an input-output demo or reminder (e.g., C_2 and C_4), which calls for a statistical learning model to make accurate predictions. (2) Both the textual context and the programming content of a code snippet can be predictive, but an effective model to jointly utilize them needs careful design. Intuitively, a text block with patterns like "you can do ..." and "this is one thorough solution ..." is more likely to be followed by a code solution. For example, given S_1 and S_3 in Figure 1, a code solution is likely to be introduced after them. On the other hand, by inspecting the code content, C_2 is probably not a code solution to the question,

since it contains special Python console patterns like ">>> ... >>>" and no particular definition of "convert".

To tackle these challenges, we explore a series of models including traditional classifiers and deep learning models, and propose a novel model, named Bi-View Hierarchical Neural Network (BiV-HNN), to capture both the textual context and the programming content of each code snippet (which make the two views). In BiV-HNN, we design two different modules to learn features from text and code respectively, and combine them into a deep neural network architecture, which finally predicts whether a code snippet is a standalone solution or not. To summarize, our contributions lie in three folds:

First, to the best of our knowledge, we are the first to investigate *systematically mining large-scale high-quality question-code pairs*, which are critical for developing learning-based models aiming to map between natural language and programming language.

Second, we extensively explore various models including traditional classifiers and deep learning models to predict whether a code snippet is a solution or not, and propose a novel Bi-View Hierarchical Neural Network which considers both text- and code-based views. On two manually labeled datasets in Python and SQL domain, BiV-HNN outperforms both the widely adopted heuristic methods and traditional classifiers by a large margin in terms of F_1 and accuracy. Moreover, BiV-HNN does not rely on any prior knowledge and can be easily applied to other programming domains.

Last but not least, we present StaQC, the largest dataset to date of ~**148K** Python and ~**120K** SQL question-code pairs, systematically mined by our framework. Using multiple case studies, we show that (1) StaQC is rich in surface variation: A question can be paired with multiple code solutions, and semantically the same code snippets can have different/paraphrased natural language descriptions. (2) Owing to such diversity as well as its large scale, StaQC is a much better data resource than existing ones for constructing models to map between natural language and programming language. In addition, we can continue to grow StaQC in both size and diversity, by regularly applying our framework to the fast-growing SO. Question-code pairs in other programming languages can also be mined similarly and included in StaQC.

2 PRELIMINARIES

In this section, we first clarify our task definition, and then describe how we annotated datasets for model development.

2.1 Task Definition

Given a question and its accepted answer post which contains multiple code snippets in Stack Overflow, we aim at predicting whether each code snippet in the answer post is a *standalone* solution to the question or not. As explained in Section 1, we focus on "accepted" answer posts and "standalone" solutions.

Users can ask different types of questions in SO such as "how to implement X" and "what/why is Y". Following previous work [12, 13, 32], we divide questions into five types: "How-to-do-it", "Debug/corrective", "Conceptual", "Seeking something, e.g., advice, tutorial", and their combinations. In particular, a question is of type "how-to-do-it" when the questioner provides a scenario and asks how to implement it like in Figure 1.

---

[3]Following previous work [2, 6, 21], we only use the title of a question post in this work, and leave incorporating the question post content for future work.

For collecting question-code pairs, we target at "how-to-do-it" questions, because answers to other types of questions are not very likely to be standalone code solutions (e.g., answers to "Conceptual" questions are usually text descriptions). Next, we describe how to distinguish "how-to-do-it" questions from others.

2.2 "How-to-do-it" Question Collection

2.2.1 Question Type Classification.

At the high level, we combined the other four question types apart from "how-to-do-it" into one category named "non-how-to" and built a binary question type classifier.

We first collected Python and SQL questions from SO based on their tags, which are available for all question posts. Specifically, we considered questions whose tags contain the keyword "python" to be in Python domain and questions tagged by "sql", "database" or "oracle" to be in SQL domain. For each domain, we randomly sampled and labeled 250 questions for training (150), validating (20) and testing (80) the classifier[4]. Among the 250 questions, around 45% in Python and 57% in SQL are "how-to-do-it" questions. We built one Logistic Regression classifier respectively for each domain, based on simple features extracted from question and answer posts as in [13], such as keyword-occurrence features, the number of code blocks in question/answer posts, the maximum length of code blocks, etc. Hyperparameters in classifiers were tuned based on validation sets.

Finally, we obtained a question-type classification accuracy of 0.738 (precision: 0.653, recall: 0.889, and F_1: 0.753) for Python and an accuracy of 0.713 (precision: 0.625, recall: 0.946, and F_1: 0.753) for SQL. The classification of question types may be further improved with more advanced features and algorithms, which is not the focus of this paper.

2.2.2 "How-to-do-it" Question Set Collection.

Using the above classifiers, we classified all Python and SQL questions in SO whose accepted answer post contains code blocks and collected a large set of "how-to-do-it" questions in each domain. Among these "how-to-do-it" questions, around 44.66% (68, 839) Python questions and 34.45% (39, 752) SQL questions have an accepted answer post with more than one code snippets, from which we will systematically mine question-code pairs.

2.3 Annotating QC Pairs for Model Training

To construct training/validation/testing datasets for our task, we hired four undergraduate students familiar with Python and SQL to annotate answer posts in these two domains. For each code snippet in an answer post, annotators can assign "1" to it if they think they can solve the problem based on the code snippet alone (i.e., it is a standalone code solution), and "0" otherwise. We ensured each code snippet is annotated by two annotators and adopted the label only when both annotators agreed on it. For each programming language, around 85% code snippets were labeled. The average Cohen's kappa agreement [9] is around 0.658 for Python and 0.691 for SQL. The statistics of our manually annotated datasets are summarized in Table 1, which will be used to develop our models.

---

[4]Despite of the small amount of training data, no overfitting was observed in our experiments partly because the features are very simple.

	Python		SQL	
	# of QC pairs	% of QC pairs with label "1"	# of QC pairs	% of QC pairs with label "1"
Training	2,932	43.89%	2,183	56.12%
Validation	976	43.14%	727	55.98%
Testing	976	47.23%	727	58.32%

Table 1: Statistics of manually annotated datasets.

3 BI-VIEW HIERARCHICAL NN

Without loss of generality, let us assume an answer post of a given question has a sequence of *blocks* $\{S_1, C_1, S_2, ..., S_i, C_i, S_{i+1}, ..., S_{L-1}, C_{L-1}, S_L\}$ with L text blocks (S_i's) and $L - 1$ code blocks (C_i's) interleaving with each other. Our task is to automatically assign a binary label to each code snippet C_i, where 1 means a standalone solution while 0 otherwise. In this work, we model each code snippet independently and predict the label of C_i based on its textual context (i.e., S_i, S_{i+1}) and programming content. If either S_i or S_{i+1} is empty, we insert an empty dummy text block to make our model applicable. One can extend our formulation to a more complicated sequence labeling problem where a sequence of code snippets can be modeled simultaneously, which we leave for future work.

3.1 Intuition

We first analyze at the high level how each individual block contributes to elaborating the entire answer fluently. For example, in Figure 1, the first text block S_1 suggests its followed code block C_1 (which implements a function) is "thorough" and thus might be a solution. S_2 subsequently connects C_1 to examples it can work with in C_2. In contrast, S_3 starts with the conjunction word "Or" and possibly will introduce an alternative solution (e.g., C_3). This observation inspires us to first model the meaning of each block separately using a *token-level sequence encoder*, then model the block sequence S_i-C_i-S_{i+1} using a *block-level encoder*, from which we finally obtain the semantic representation of C_i.

Figure 2 shows our model, named Bi-View Hierarchical Neural Network (BiV-HNN). It progressively learns the semantic representation of a code block from token level to block level, based on which we predict it to be a standalone solution or not. On the other hand, BiV-HNN naturally incorporates two views, i.e., textual context and code content, into the model structure. We detail each component as follows.

3.2 Token-level Sequence Encoder

Text block. Given a sentence block S_i with a sequence of words w_{it}, $t \in [1, T_i]$, we first embed the words into vectors through a pretrained word embedding matrix W_e, i.e., $x_{it} = W_e w_{it}$. We then use a bidirectional Gated Recurrent Unit (GRU) based Recurrent Neural Network (RNN) [8] to learn the word representation by summarizing the contextual information from both directions. The GRU tracks the state of sequences by controlling how much information is updated into the new hidden state from previous states. Specifically, given the input word vector x_t in the current step and the hidden state h_{t-1} from the last step, the GRU first computes a *reset gate r* for resetting information from previous steps in order
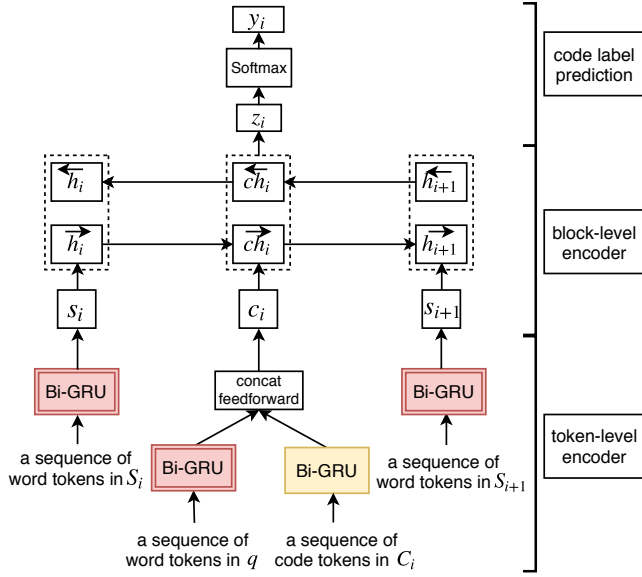
Figure 2: Our Bi-View Hierarchical Neural Network (BiV-HNN). Text block S_i and question q are encoded by a bidirectional GRU-based RNN (Bi-GRU) module and code block C_i is encoded by another Bi-GRU with different parameters.

to learn a new hidden state $\tilde{h}_t$:

$$r = \sigma(W_r[x_t, h_{t-1}] + b_r),$$
$$\tilde{h}_t = \phi(W[x_t, r \odot h_{t-1}] + b),$$

where $[x_t, h_{t-1}]$ is the concatenation of x_t and h_{t-1}, σ and ϕ are the *sigmoid* and *tanh* activation function respectively. W_r, W are two weight matrices in $R^{d_h \times (d_x + d_h)}$ and b_r, b are the biases in R^{d_h}, where d_x, d_h is the dimension of x_t and the hidden state h_{t-1} respectively. Intuitively, if r is close to 0, then the information in h_{t-1} will not be passed into the current step when learning the new hidden state. The GRU also defines an *update gate u* for integrating hidden states h_{t-1} and $\tilde{h}_t$:

$$u = \sigma(W_u[x_t, h_{t-1}] + b_u),$$
$$h_t = u h_{t-1} + (1 - u)\tilde{h}_t.$$

When u is closer to 0, h_t contains more information about the current step $\tilde{h}_t$; otherwise, it memorizes more about previous steps. Onwards, we denote the above calculation by $h_t = GRU(x_t, h_{t-1})$ for convenience.

In our work, the bidirectional GRU (i.e., Bi-GRU) contains a forward GRU reading a text block S_i from w_{i1} to w_{iT_i} and a backward GRU which reads from w_{iT_i} to w_{i1}:

$$\overrightarrow{h}_{it} = GRU(x_{it}, \overrightarrow{h}_{i,t-1}), t \in [1, T_i],$$
$$\overleftarrow{h}_{it} = GRU(x_{it}, \overleftarrow{h}_{i,t+1}), t \in [T_i, 1],$$
$$\overrightarrow{h}_{i0} = \overrightarrow{0}, \quad \overleftarrow{h}_{i,T_i+1} = \overleftarrow{0},$$

where the hidden states in both directions are initialized with zero vectors. Since the forward and backward GRU summarize the context information from different perspectives, we concatenate their last hidden states (i.e., $\overrightarrow{h}_{iT_i}$, $\overleftarrow{h}_{i1}$) to represent the meaning of the text block S_i:

$$s_i = [\overrightarrow{h}_{iT_i}, \overleftarrow{h}_{i1}].$$

Code block. Similarly, we employ another Bi-GRU RNN module to learn a vector representation v_c for code block C_i based on its code token sequence. One may directly take this code vector v_c as the token-level representation of a code block. However, since the goal of our model is to decide whether a code snippet answers a certain question, we associate C_i with the question title q to capture their semantic correspondences in the learnt vector representation c_i. Specifically, we first learn the *question vector v_q* by applying the token-level *text* encoder to the word sequence in q. The concatenation of v_q and v_c is then fed into a feedforward *tanh* layer (i.e., "concat feedforward" in Figure 2) for generating c_i:

$$c_i = \phi(W_c[v_q, v_c] + b_c).$$

We will verify the effect of incorporating q in our experiments.

Unlike modeling a code block, we do not associate a text block with question q when learning its representation, because we observed no direct semantic matching between the two. For example, in Figure 1, a text block can hardly match the question by its content. However, as we discussed in Section 1, a text block with patterns like "you can do ..." or "This is one thorough solution ..." can imply that a code solution will be introduced after it. Therefore, we model each text block per se, without incorporating question information.

3.3 Block-level Sequence Encoder

Given the sequence of token-level representations s_i-c_i-s_{i+1}, we use a bidirectional GRU-based RNN to build a block-level sequence encoder and finally obtain the code block representation:

$$\overrightarrow{h}_i = GRU(s_i, \overrightarrow{0}), \overleftarrow{h}_i = GRU(s_i, \overleftarrow{ch}_i),$$
$$\overrightarrow{ch}_i = GRU(c_i, \overrightarrow{h}_i), \overleftarrow{ch}_i = GRU(c_i, \overleftarrow{h}_{i+1}),$$
$$\overrightarrow{h}_{i+1} = GRU(s_{i+1}, \overrightarrow{ch}_i), \overleftarrow{h}_{i+1} = GRU(s_{i+1}, \overleftarrow{0}),$$

where the encoder is initialized with zero vectors (i.e., $\overrightarrow{0}$ and $\overleftarrow{0}$) in both directions. We concatenate the forward state $\overrightarrow{ch}_i$ and the backward state $\overleftarrow{ch}_i$ of the code block as its semantic representation:

$$z_i = [\overrightarrow{ch}_i, \overleftarrow{ch}_i].$$

3.4 Code Label Prediction

The representation z_i of code block C_i is then used for prediction:

$$y_i = \text{softmax}(W_y z_i + b_y),$$

where $y_i = [y_{i0}, y_{i1}]$ represents the probability of predicting C_i to have label 0 or 1 respectively.

We define the loss function using cross entropy [19], which is averaged over all the N code snippets during training:

$$\mathcal{L} = -\frac{1}{N} \sum_{i=1}^{N} p_{i0} \log(y_{i0}) + p_{i1} \log(y_{i1}),$$

where $p_{i0} = 0$ and $p_{i1} = 1$ if the i-th code snippet is manually annotated as a solution; otherwise, $p_{i0} = 1$ and $p_{i1} = 0$.

4 TRADITIONAL CLASSIFIERS WITH FEATURE ENGINEERING

In addition to neural network based models like BiV-HNN, we also explore traditional classifiers like Logistic Regression (LR) [11] and Support Vector Machine (SVM) [10] for our task. Features are manually crafted from both text- and code-based views:

Textual Context. (1) Token: The unigrams and bigrams in the context. (2) FirstToken: If a sentence starts with phrases like "try this" or "use it", then the following code snippet is very likely to be the solution. Inspired by this idea, we discriminate the first token from others in the context. (3) Conn: Boolean features indicating whether a connective word/phrase (e.g., "alternatively") occurs in the context. We used the common connective words and phrases from Penn Discourse Tree Bank [43].

Code Content. (1) CodeToken: All code tokens in a code snippet. (2) CodeClass: To discriminate code snippets that function and can be considered for learning and pragmatic reuse (i.e., "working code" [22]) from input-output demos, we introduce CodeClass, which is the probability of a code snippet being a working code. Specifically, from all the "how-to-do-it" Python questions in SO, we first collected totally 850 code snippets following text blocks such as "output:" and "output is:" as input-output code snippets. We further randomly selected 850 accepted answer posts containing exactly one code snippet and took their code snippets as the working code. We then extracted a set of features like the proportion of numbers and parenthesis and constructed a binary Logistic Regression classifier, which obtains 0.804 accuracy and 0.891 F_1 on a manually labeled testing set. Finally, the trained classifier outputs the probability for each code snippet in Python being a "working code" as the CodeClass feature. For SQL, a working code can usually be detected by keywords like "SELECT" and "DELETE", which have been included in the CodeToken feature. Thus, we did not design the CodeClass feature for it.

There could be other features to incorporate into traditional classifiers. However, coming up with useful features is anything but an easy task. In contrast, neural network models can automatically learn advanced features from raw data and have been broadly and successfully applied in different areas [8, 24, 30, 50, 53]. Therefore, in our work, we choose to design the neural network based model BiV-HNN. We will compare different models in experiments.

5 EXPERIMENTS

In this section, we conduct extensive experiments to compare various models and show the advantages of our proposed BiV-HNN.

5.1 Experimental Setup

Dataset Summarization. Section 2 discussed how we manually annotated question-code pairs for training, validation and testing. Statistics were summarized in Table 1. To evaluate different models, we adopt precision, recall, F_1, and accuracy, which are defined in the same way as in a typical binary classification setting.

Data Preprocessing. We tokenized Python code snippets with best efforts: We first applied Python built-in tokenizer and for code lines that remain untokenized after that, we adopted the "word-punct_tokenizer" in NLTK toolkit [27] to separate tokens and symbols (e.g., "." and "="). In addition, we detected variables, numbers and strings in a code snippet by traversing its Abstract Syntax Tree (AST) parsed with Python built-in AST parser, and replaced them with special tokens "VAR", "NUMBER" and "STRING" respectively, to alleviate data sparsity. For SQL, we followed [21] to perform the tokenization, which replaced table/column names with placeholder tokens and numbered them to preserve their dependencies. Finally, we collected 4,557 (3,191) word tokens and 6,581 (1,200) code tokens from Python (SQL) training set.

Implementation Details. We used Tensorflow[55] to implement our BiV-HNN and its variants to be introduced in Section 5.2. The embedding size of word and code tokens was set at 150. The embedding vectors were pre-trained using GloVe [42] on all Python or SQL posts in SO. Parameters were randomly initialized following [18]. We started the learning rate at 0.001 and trained neural network models in mini batch of size 100 with the Adam optimizer [23]. The size of the GRU units was chosen from {64, 128} for token-level encoders and from {128, 256} for block-level encoders. Following the convention [20, 21, 29], we selected model parameters based on their performance on validation sets. The Logistic Regression and Support Vector Machine models were implemented with Python Scikit-learn library [41].

5.2 Baselines and Variants of BiV-HNN

Baselines. We compare our proposed model with two commonly used heuristics for collecting QC pairs: (1) *Select-First*: Only treat the first code snippet in an answer post as a solution; (2) *Select-All*: Treat every code snippet in an answer post as a solution and pair each of them with the question. In addition, we compare our model with traditional classifiers like LR and SVM based on hand-crafted features (Section 4).

Variants of BiV-HNN. *First*, to evaluate the effectiveness of combining two views (i.e., textual context and code content), we adapt BiV-HNN to consider only one single view: (1) *Text-HNN* (Figure 3a): In this model, we only utilize textual contexts of a code snippet. We mask all code blocks with a special token CODEBLOCK and represent them with a unified vector. (2) *Code-HNN* (Figure 3b): We only feed the output of the token-level code encoder (i.e., c_i) into the "code label prediction" layer in Section 3, and do not model textual contexts. In addition, to evaluate the effect of question q when encoding a code block, we compare BiV-HNN with *BiV-HNN-nq*, which directly takes the code vector v_c as the code block representation c_i, without associating question q, for further learning. These three models are all input-level variants of BiV-HNN.

Second, to evaluate the hierarchical structure in BiV-HNN, we compare it with "flat" RNN models, which model word and code tokens as a single sequence. The comparison is conducted in both text-only and bi-view settings: (1) *Text-RNN* (Figure 4a): Compared with Text-HNN, we concatenate all words in context blocks S_i and S_{i+1} as well as the unified code vector CODEBLOCK as a single sequence, i.e., $\{w_{i1}, ..., w_{i,T_i}, \text{CODEBLOCK}, w_{i+1,1}, ..., w_{i+1,T_{i+1}}\}$, using Bi-GRU RNN. The concatenation of the forward and backward
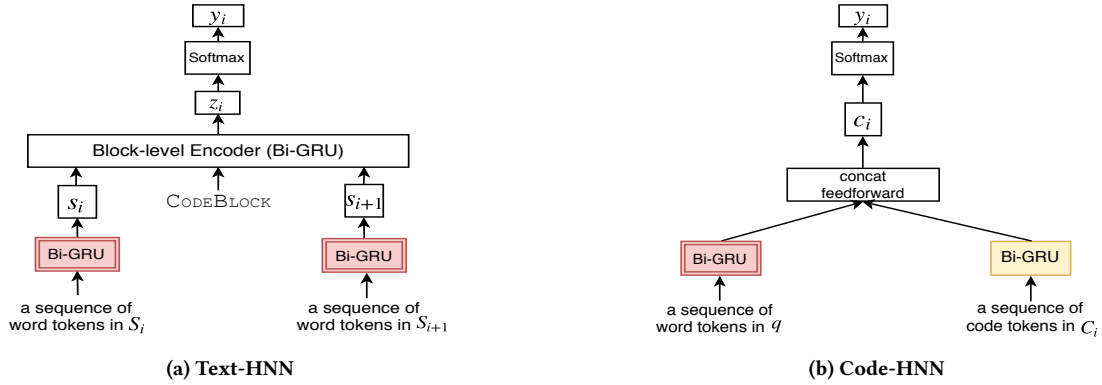
(a) Text-HNN

(b) Code-HNN

Figure 3: Single-view variants of BiV-HNN: (a) Text-HNN, without code content; (b) Code-HNN, without contextual text.
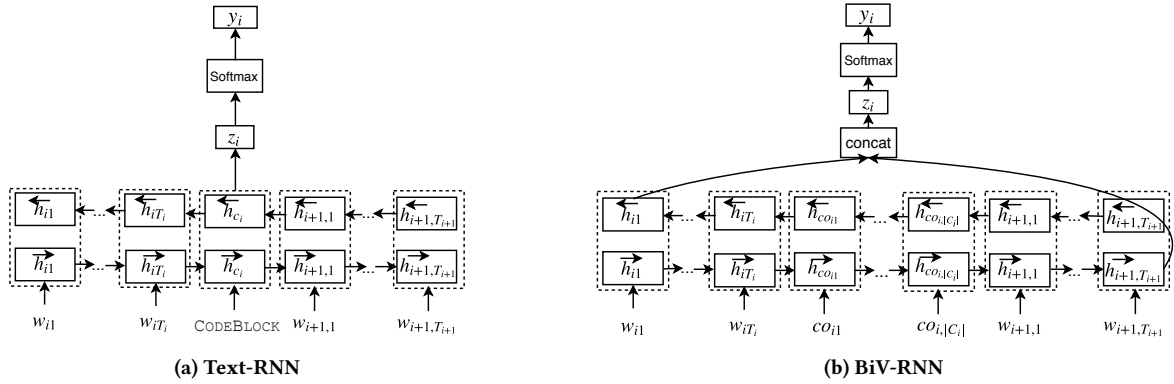

(a) Text-RNN

(b) BiV-RNN

Figure 4: "Flat"-structure variants of BiV-HNN, without differentiating token- and block-level: (a) Text-RNN; (b) BiV-RNN.

hidden states of CODEBLOCK is considered as its final semantic vector z_i, which is then fed into the code label prediction layer. (2) *BiV-RNN* (Figure 4b): In contrast to BiV-HNN, BiV-RNN models all word and code tokens in S_i-C_i-S_{i+1} as a single sequence, i.e., $\{w_{i1}, ..., w_{iT_i}, co_{i1}, ..., co_{ij}, ..., co_{i,|C_i|}, w_{i+1,1}, ..., w_{i+1,T_{i+1}}\}$, where co_{ij} denotes the j-th token in code C_i and $|C_i|$ is the number of code tokens in C_i. BiV-RNN concatenates the last hidden states in two directions as the final semantic vector z_i for prediction. We also tried directly "flattening" BiV-HNN by concatenating tokens in S_i-q-C_i-S_{i+1}, but observed worse performance, perhaps because transitioning from S_i to question q is less natural.

Finally, at the block level, instead of using an RNN, one may apply a feedforward neural network [47] to the concatenated token-level output $[s_i, c_i, s_{i+1}]$. Specifically, the block-level Bi-GRU in BiV-HNN can be replaced with a one-layer[5] feedforward neural network, denoted as *BiV-HFF*. Intuitively, modeling the three blocks as a sequence is more consistent with the way humans read a post. We will verify this intuition in experiments.

While there could be other variants of our model, the above ones are related to the most critical designs in BiV-HNN. We only show their performance due to space constraints.

---

[5]For fair comparison, we only use one layer since the Bi-GRU in BiV-HNN only has one hidden layer.

5.3 Results

Our experimental results in Table 2 show the effectiveness of our BiV-HNN. On both datasets, BiV-HNN substantially outperforms heuristic baselines Select-First and Select-All by more than **15%** in F_1 and accuracy. This demonstrates that our model can collect QC pairs with much higher quality than heuristic methods used in existing research. In addition, when compared with LR and SVM, BiV-HNN achieves 7%~9% higher F_1 and accuracy on Python dataset, and 3%~5% better F_1 and accuracy on SQL dataset. The gain on SQL data is relatively smaller, probably because interpreting SQL programs is a relatively easier task, implied by the observation that both simple classifiers and BiV-HNN can have around 85% F1.

Results in Table 3 show the effect of key components in BiV-HNN in comparison with alternatives. Due to space constraints, we do not show the accuracy of each model, which has roughly the same pattern as F_1. We have made the following observations: (1) *Single-view variants*. BiV-HNN outperforms Text-HNN and Code-HNN by a large margin on both datasets, showing that both views are critical for our task. In particular, by incorporating code content information, BiV-HNN is able to improve Text-HNN by 7% on Python dataset and around 5% on SQL dataset in F_1. (2) *No-query variant*. On Python dataset, the integration of the question information in BiV-HNN brings 3% F_1 improvements over BiV-HNN-nq,

Model	Python Testing Set				SQL Testing Set			
	Precision	Recall	F_1	Accuracy	Precision	Recall	F_1	Accuracy
Heuristics Baselines								
Select-First	0.676	0.551	0.607	0.663	0.755	0.517	0.613	0.620
Select-All	0.472	**1.000**	0.642	0.472	0.583	**1.000**	0.737	0.583
Classifiers based on simple features								
Logistic Regression	0.801	0.733	0.766	0.788	0.843	0.849	0.846	0.820
Support Vector Machine	0.701	0.813	0.753	0.748	0.843	0.858	0.850	0.824
BiV-HNN	**0.808**	0.876	**0.841**	**0.843**	**0.872**	0.903	**0.888**	**0.867**

Table 2: Comparison of BiV-HNN and baseline methods.

which shows the effectiveness of associating the question with the code snippet for identifying code answers. For SQL dataset, adding the question gives no obvious benefit, possibly because the code content in each SQL program already carries critical information for making a prediction (e.g., a SQL program containing the command keyword "SELECT" is very likely to be a solution to the given question, regardless of the question content). (3) *"Flat"-structure variants.* On both datasets, the hierarchical structure leads to 1%~2% improvements against the "flat" structure in both bi-view (BiV-HNN vs. BiV-RNN) and single-view setting (Text-HNN vs. Text-RNN). (4) *Non-sequence variant.* On Python dataset, BiV-HNN outperforms BiV-HFF by around 2%, showing the block-level Bi-GRU is preferable over feedforward neural networks. The two models get roughly the same performance on SQL, probably because our task is easier in SQL domain than in Python domain as we mentioned earlier.

In summary, our BiV-HNN is much more effective than widely-adopted heuristic baselines and traditional classifiers. The key components in BiV-HNN, such as bi-view inputs, hierarchical structure and block-level sequence encoding, are also empirically justified.

Error Analysis. There are a variety of non-solution roles that a code snippet can play, such as being only one step of a multi-step solution, an input-output example, etc. We observe that more than half of the wrong predictions were false positives (i.e., predicting a non-solution code snippet as a solution), correcting which usually requires integrating information from the entire answer post. For example, when a code snippet is the first step of a multi-step solution, BiV-HNN may mistakenly take it as a complete and standalone solution, since BiV-HNN does not simultaneously take into account follow-up code snippets and their context to make predictions. In addition, BiV-HNN may make mistakes when a correct prediction requires a close examination of the *content* of a question post (besides its title). Exploring these directions in the future may lead to further improved model performance on this task.

Model Combination. When experimenting with the single-view variants of BiV-HNN, i.e., Text-HNN and Code-HNN, we observed that the three models complement each other in making accurate predictions. For example, on Python validation set, around 70% mistakes made by Text-HNN or Code-HNN can be corrected by considering predictions from the other two models. Although BiV-HNN is built based on both text- and code-based views, 60% of its wrong predictions can be remedied by Text-HNN and Code-HNN. The same pattern was also observed on SQL dataset.

Therefore, we further tested the effect of combining the three models via a simple heuristic: The label of a code snippet is predicted

Model	Python Testing Set			SQL Testing Set		
	Prec.	Rec.	F_1	Prec.	Rec.	F_1
Single-view Variants						
Text-HNN	0.723	0.826	0.771	0.798	0.887	0.840
Code-HNN	0.770	0.859	0.812	0.848	0.854	0.851
No-query Variant						
BiV-HNN-nq	0.802	0.818	0.810	**0.883**	0.892	0.887
"Flat"-structure Variants						
Text-RNN	0.693	0.824	0.753	0.773	0.894	0.829
BiV-RNN	0.760	**0.887**	0.819	0.869	0.880	0.875
Non-sequence Variant						
BiV-HFF	0.787	0.859	0.822	0.845	**0.939**	**0.889**
BiV-HNN	**0.808**	0.876	**0.841**	0.872	0.903	0.888

Table 3: Comparison of BiV-HNN and its variants.

only when the three models agree on it. Using this heuristic, 69.2% code blocks on the annotated Python testing set are labeled with 0.916 F_1 and 0.911 accuracy. Similarly, on SQL testing set, 78.7% code blocks are labeled with 0.943 F_1 and 0.926 accuracy. The combined model further improves BiV-HNN by around 6% while still being able to label a large portion of the code snippets. Thus, we apply this combined model to those SO answer posts that are not manually annotated yet to obtain large-scale QC pairs, to be discussed next.

6 STAQC: A SYSTEMATICALLY MINED DATASET OF QUESTION-CODE PAIRS

In this section, we present StaQC (Stack Overflow Question-Code pairs), a large-scale and diverse set of question-code pairs automatically mined using our framework. Under various case studies, we demonstrate that StaQC can greatly help tasks aiming to associate natural language with programming language.

6.1 Statistics of StaQC

In Section 5, we showed that a combination of BiV-HNN and its variants can reliably identify standalone code solutions with > 90% F_1 and accuracy from a large portion of the testing set. Thus we applied this combined model to all unlabeled multi-code answer posts that correspond to "how-to-do-it" questions in Python and SQL domain, and finally collected **60,083** and **41,826** question-code pairs respectively. Additionally, there are 85,294 Python and 75,637 SQL "how-to-do-it" questions whose answer post contains exactly one code snippet. For them, as in [21], we paired the question title

	# of QC pairs	Question		Code	
		Average length	# of tokens	Average length	# of tokens
Python	**147,546**	9	17,635	86	137,123
SQL	**119,519**	9	9,920	60	21,143

Table 4: Statistics of StaQC.

Question: "*How to limit a number to be within a specified range? (Python)*"
Code answers:

```
def clamp(n, minn, maxn):
  return max(min(maxn, n),
minn)                    (1st)
```

```
clamp = lambda n, minn,
maxn: max(min(maxn, n),
minn)                    (2nd)
```

```
def clamp(n, minn, maxn):
  if n < minn:
    return minn
  elif n > maxn:
    return maxn
  else:
    return n             (5th)
```

```
n = minn if n < minn else maxn if n > maxn else n    (4th)
```

Figure 5: StaQC contains four alternative code solutions to question "*How to limit a number to be within a specified range? (Python)*" [38] whose answer post contains five code snippets. The number at the bottom right denotes the position of each code snippet in the answer post.

with the one code snippet as a question-code pair. Together with 2,169 and 2,056 manually annotated QC pairs with label "1" for each domain (Table 1), we collected a dataset of **147,546** Python and **119,519** SQL QC pairs, named as StaQC. Table 4 shows its statistics.

Note that we can continue to expand StaQC with minimal efforts, since it is automatically mined by our framework, and more and more posts will be created in SO as time goes by. QC pairs in other programming languages can also be mined similarly to further enrich StaQC beyond Python and SQL domain.

6.2 Diversity of StaQC

Besides the large scale, StaQC also enjoys great diversity in the sense that it contains multiple textual descriptions for semantically similar code snippets and multiple code solutions to a question. For example, considering question "*How to limit a number to be within a specified range? (Python)*" whose answer post contains five code snippets (Figure 5), our framework is able to correctly mine four alternative code answers. Heuristic methods may either miss some of them or mistakenly include a false solution (i.e., the 3rd code snippet). Therefore, our framework is able to obtain more alternative solutions for the same question more accurately. Moreover, Figure 6 shows two question-code pairs included in StaQC, which we easily located by comparing code solutions of *relevant* questions in SO (i.e., questions manually linked by SO users). Note that the two code snippets have a very similar functionality but two different text descriptions.

Figure 5 and 6 show that StaQC is highly diverse and rich in surface variation. Such a dataset is beneficial for model development. Intuitively, when certain data patterns are not observed in the training phase, a model is 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 or from different

Question A: "*How to find a gap in range in SQL*"

```
SELECT id + 1
FROM test mo
WHERE NOT EXISTS
(
   SELECT NULL
   FROM test mi
   WHERE mi.id = mo.id + 1
   ) and mo.id> 100
ORDER BY
       id
LIMIT 1
```

Question B: "*How do I find a "gap" in running counter with SQL?*"

```
SELECT id + 1
FROM mytable mo
WHERE NOT EXISTS
(
   SELECT NULL
   FROM mytable mi
   WHERE mi.id = mo.id + 1
   )
ORDER BY
       id
LIMIT 1
```

Figure 6: StaQC has different text descriptions, e.g., "*How to find a gap in range in SQL*" [37] and "*How do I find a "gap" in running counter with SQL?*" [36], for two code snippets bearing a similar functionality.

text descriptions to similar code snippets. Next we demonstrate this benefit using an exemplar downstream task.

6.3 Usage Demo of StaQC on Code Retrieval

To further demonstrate the usage of StaQC, we employ it to train a deep learning model for the code retrieval task [2, 21, 22]. *Given a natural language description and a set of code snippet candidates, the task is to retrieve code snippets that can match the description.* In particular, an effective model should rank matched code snippets as high as possible. Models are evaluated by Mean Reciprocal Rank (MRR) [58]. In [21], the authors proposed a neural network based model, *CODE-NN*, which outputs a matching score between a natural language question and a code snippet. We choose CODE-NN as it is one of the state of the arts for code retrieval and improved previous work by a large margin. For training, the authors collected around 25,870 SQL QC pairs from answer posts containing exactly one code snippet (which is paired with the question title). They manually annotated two datasets DEV and EVAL for choosing the best model parameters and for final evaluation respectively, both containing around 100 QC pairs. The final evaluation is conducted in 20 runs. In each run, for every QC pair in DEV or EVAL, [21] randomly selected 49 code snippets from SO as non-answer candidates, and ranked all 50 code snippets based on their scores output by CODE-NN. The averaged MRR is computed as the final result.

Improved Retrieval Performance. We first trained CODE-NN using the *original* training set in [21]. We denote this setting as *CODE-NN (original)*. Then we used StaQC to upgrade the training data in two most straightforward ways: (1) We directly took all the 119,519 SQL QC pairs in StaQC to train CODE-NN, denoted as *CODE-NN (StaQC)*. (2) To emphasize the effect of our framework, we just added the 41,826 QC pairs, automatically mined from SO multi-code answer posts, to the original training set and retrained the model, which is denoted as *CODE-NN (original + StaQC-multi)*. In both (1) and (2), questions and code snippets occurring in the DEV/EVAL set were removed from training.

In all three settings, we used the same DEV/EVAL set and the same hyper-parameters as in [21] except the dropout rate, which was chosen from {0.5, 0.7} for each model to obtain better performance. Like [21], we decayed the learning rate in each epoch and terminated the training when it was lower than 0.001. The best

Model Setting	MRR
CODE-NN (original)	0.51 ± 0.02
CODE-NN (StaQC)[*]	**0.57 ± 0.02**
CODE-NN (original + StaQC-multi)[*]	0.54 ± 0.02

Table 5: Performance of CODE-NN [21] on code retrieval, with and without StaQC for training. [*]denotes statistically significant w.r.t. *CODE-NN (original)* under one-tailed Student's t-test ($p < 0.05$).

model was selected as the one achieving the highest average MRR on DEV set. When using this strategy, we observed better results on the EVAL set than those reported in [21] (around 0.44).

Table 5 shows the average MRR score and standard deviation of each model on EVAL set. We can see that directly using StaQC for training leads to a substantial 6% improvement over using the original dataset in [21]. By adding QC pairs we mined from multi-code posts to the original training data, CODE-NN can be significantly improved by 3%. Note that the performance gains shown here are still conservative, since we adopted 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. StaQC can also be used to train other code retrieval models besides CODE-NN, as well as models for other related tasks like code generation or annotation.

7 DISCUSSION AND FUTURE WORK

Besides boosting relevant tasks using StaQC, future work includes: (1) We currently only consider a code snippet to be a standalone solution or not. In many cases, code snippets in an answer post serve as multiple steps and should be merged to form a complete solution [39]. This is a more challenging task and we leave it to the future. (2) In our experiments, we combined BiV-HNN and its two variants using a simple heuristic to achieve better performance. In the future, one can also use StaQC to retrain the three models, similar to self-training [34], or jointly train the three models in a tri-training framework [63]. (3) One may also employ Convolutional Neural Networks [1, 24, 49], which have shown great power on representation learning, to encode text and code blocks. Moreover, we can consider encoders similar to [31, 33] for capturing the intrinsic structure of programming language.

8 RELATED WORK

***Language + Code* Tasks and Datasets**. Tasks that map between natural language and programming language, referred to *Language + Code* tasks here, such as code annotation and code retrieval/generation, have been popularly studied in recent years [2, 15, 21, 22, 26, 35, 45, 57, 64]. In order to train more advanced yet data-hungry models, researchers have collected data either automatically from online communities [2, 5, 21, 22, 26, 45, 57, 64] or with human intervention [16, 35]. Like our work, [2, 21, 57, 64] utilized SO to collect data. Particularly, [2] merges code snippets in its answer post as the target source code and pair it with the question title. [21] only employs accepted answer posts containing exactly one code snippet. Other interesting datasets include ~19K <English pseudo-code, Python code snippet> pairs manually annotated by

[35], and ~114K pairs of Python functions and their documentation strings heuristically collected by [5] from GitHub [17]. Unlike their work, we systematically mine high-quality question-code pairs from SO using advanced machine learning models. Our mined dataset StaQC, the largest to date of around 148K Python and 120K SQL question-code pairs, has been shown to be a better resource. Moreover, StaQC is easily expandable in terms of both scale and programming language types.

Recurrent Neural Networks for Sequential Data. Recurrent Neural Networks have shown great success in various natural language tasks [4, 8, 20, 29]. In an RNN, terms are modeled sequentially without discrimination. Recently, in order to handle information at different levels, [25, 48, 54, 60] stack multiple RNNs into a hierarchical structure. For example, [60] incorporates the attention mechanism in a hierarchical RNN model to pick up important words and sentences. Their model finally aggregates all sentence vectors to learn the document representation. In comparison, we utilize the hierarchical structure to first learn the semantic meaning of each block individually, and then predict the label of a code snippet by combining two views: textual context and programming content.

Mining Stack Overflow. Stack Overflow has been the focus of the Mining Software Repositories (MSR) challenge for years [3, 62]. A lot of work [12–14, 32, 56, 59] have been done on exploring the categories of questions, mining source codes, etc. We follow [12, 13, 32] to categorize SO questions into 5 classes but only focus on the "how-to-do-it" type (Section 2). [14, 59] analyzes the quality of code snippets (e.g., readability) or explores "usable" code snippets that could be parsed, compiled and run. Different from their work, we are interested in finding standalone code solutions, which are not necessarily directly parsable, compilable or runnable, but can be semantically paired with questions. To the best of our knowledge, we are the first to study the problem of systematically mining high-quality question-code pairs.

9 CONCLUSION

This paper explores systematically mining question-code pairs from Stack Overflow, in contrast to heuristically collecting them. We focus on the "how-to-do-it" questions since their answers are more likely to be code solutions. We present the largest-to-date dataset of diversified question-code pairs in Python and SQL domain (StaQC), systematically collected by our framework. StaQC can greatly help downstream tasks aiming to associate natural language with programming language. We will release it together with our source code for future research.

ACKNOWLEDGMENTS

REFERENCES

[1] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *ICML*. 2091–2100.
[2] Miltos Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. 2015. Bimodal modelling of source code and natural language. In *ICML*. 2123–2132.
[3] Alberto Bacchelli. 2013. Mining Challenge 2013: Stack Overflow. In *The 10th Working Conference on Mining Software Repositories*. to appear.
[4] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural Machine Translation by Jointly Learning to Align and Translate. *CoRR* abs/1409.0473 (2014). arXiv:1409.0473 http://arxiv.org/abs/1409.0473
[5] Antonio Valerio Miceli Barone and Rico Sennrich. 2017. A parallel corpus of Python functions and documentation strings for automated code documentation and code generation. *arXiv preprint arXiv:1707.02275* (2017).
[6] Brock Angus Campbell and Christoph Treude. 2017. NLP2Code: Code Snippet Content Assist via Natural Language Tasks. *arXiv preprint arXiv:1701.05648* (2017).
[7] Ohio Supercomputer Center. 1987. Ohio Supercomputer Center. http://osc.edu/ark:/19495/f5s1ph73. (1987).
[8] Kyunghyun Cho, Bart van Merriënboer, Çağlar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *EMNLP*. Association for Computational Linguistics, Doha, Qatar, 1724–1734.
[9] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.
[10] Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. *Machine learning* 20, 3 (1995), 273–297.
[11] David R Cox. 1958. The regression analysis of binary sequences. *Journal of the Royal Statistical Society. Series B (Methodological)* (1958), 215–242.
[12] Lucas BL de Souza, Eduardo C Campos, and Marcelo de A Maia. 2014. Ranking crowd knowledge to assist software development. In *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, 72–82.
[13] Fernanda Madeiral Delfim, Klérisson VR Paixão, Damien Cassou, and Marcelo de Almeida Maia. 2016. Redocumenting APIs with crowd knowledge: a coverage analysis based on question types. *Journal of the Brazilian Computer Society* 22, 1 (2016), 9.
[14] Maarten Duijn, Adam Kučera, and Alberto Bacchelli. 2015. Quality questions need quality code: classifying code fragments on stack overflow. In *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE Press, 410–413.
[15] Alessandra Giordani and Alessandro Moschitti. 2009. Semantic mapping between natural language questions and SQL queries via syntactic pairing. In *International Conference on Application of Natural Language to Information Systems*. Springer, 207–221.
[16] Alessandra Giordani and Alessandro Moschitti. 2010. Corpora for Automatically Learning to Map Natural Language Questions into SQL Queries.. In *LREC*.
[17] GitHub. 2017. GitHub. (2017). https://github.com/
[18] Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. 249–256.
[19] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.
[20] Karl Moritz Hermann, Tomas Kocisky, Edward Grefenstette, Lasse Espeholt, Will Kay, Mustafa Suleyman, and Phil Blunsom. 2015. Teaching machines to read and comprehend. In *NIPS*. 1693–1701.
[21] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *ACL*, Vol. 1. 2073–2083.
[22] Iman Keivanloo, Juergen Rilling, and Ying Zou. 2014. Spotting working code examples. In *ICSE*. ACM, 664–675.
[23] Diederik Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).
[24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *NIPS*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 1097–1105.
[25] Jiwei Li, Minh-Thang Luong, and Dan Jurafsky. 2015. A hierarchical neural autoencoder for paragraphs and documents. *arXiv preprint arXiv:1506.01057* (2015).
[26] Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiský, Andrew Senior, Fumin Wang, and Phil Blunsom. 2016. Latent predictor networks for code generation. *arXiv preprint arXiv:1603.06744* (2016).
[27] Edward Loper and Steven Bird. 2002. NLTK: The Natural Language Toolkit. In *Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics - Volume 1 (ETMTNLP '02)*. Association for Computational Linguistics, Stroudsburg, PA, USA, 63–70. https://doi.org/10.3115/1118108.1118117
[28] Pablo Loyola, Edison Marrese-Taylor, and Yutaka Matsuo. 2017. A Neural Architecture for Generating Natural Language Descriptions from Source Code Changes. *arXiv preprint arXiv:1704.04856* (2017).
[29] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective Approaches to Attention-based Neural Machine Translation. In *EMNLP*.
[30] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *NIPS*. 3111–3119.
[31] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *AAAI*.
[32] Seyed Mehdi Nasehi, Jonathan Sillito, Frank Maurer, and Chris Burns. 2012. What makes a good code example?: A study of programming Q&A in StackOverflow. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 25–34.
[33] Anh Tuan Nguyen and Tien N Nguyen. 2015. Graph-based statistical language model for code. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 858–868.
[34] Kamal Nigam and Rayid Ghani. 2000. Analyzing the effectiveness and applicability of co-training. In *CIKM*. ACM, 86–93.
[35] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 574–584.
[36] Stack Overflow. 2017. How do I find a "gap" in running counter with SQL? (2017). https://stackoverflow.com/a/1312137/4941215
[37] Stack Overflow. 2017. How to find a gap in range in SQL. (2017). https://stackoverflow.com/a/17782635/4941215
[38] Stack Overflow. 2017. How to limit a number to be within a specified range? (Python). (2017). https://stackoverflow.com/a/5996949/4941215
[39] Stack Overflow. 2017. Splitting a dataframe based on column values. (2017). https://stackoverflow.com/a/33973304/4941215
[40] Stack Overflow. 2017. Stack Overflow. (2017). https://stackoverflow.com/
[41] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.
[42] Jeffrey Pennington, Richard Socher, and Christopher D Manning. 2014. Glove: Global Vectors for Word Representation.. In *EMNLP*, Vol. 14. 1532–1543.
[43] Rashmi Prasad, Nikhil Dinesh, Alan Lee, Eleni Miltsakaki, Livio Robaldo, Aravind Joshi, and Bonnie Webber. 2008. The Penn Discourse TreeBank 2.0. In *In Proceedings of LREC*.
[44] Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract Syntax Networks for Code Generation and Semantic Parsing. In *ACL*.
[45] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. 2016. SWIM: synthesizing what I mean: code search and idiomatic snippet synthesis. In *ICSE*. ACM, 357–367.
[46] Alexander J Ratner, Christopher M De Sa, Sen Wu, Daniel Selsam, and Christopher Ré. 2016. Data programming: Creating large training sets, quickly. In *NIPS*. 3567–3575.
[47] David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. 1988. Learning representations by back-propagating errors. *Cognitive modeling* 5, 3 (1988), 1.
[48] Iulian Vlad Serban, Alessandro Sordoni, Yoshua Bengio, Aaron C Courville, and Joelle Pineau. 2016. Building End-To-End Dialogue Systems Using Generative Hierarchical Neural Network Models.. In *AAAI*. 3776–3784.
[49] Yelong Shen, Xiaodong He, Jianfeng Gao, Li Deng, and Grégoire Mesnil. 2014. A latent semantic model with convolutional-pooling structure for information retrieval. In *CIKM*. ACM, 101–110.
[50] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).
[51] Inc Stack Exchange. 2017. Stack Exchange Data Dump. (2017). https://archive.org/details/stackexchange
[52] Yu Su, Ahmed Hassan Awadallah, Madian Khabsa, Patrick Pantel, and Michael Gamon. 2017. Building Natural Language Interfaces to Web APIs. In CIKM.
[53] Christian Szegedy, Alexander Toshev, and Dumitru Erhan. 2013. Deep neural networks for object detection. In *NIPS*. 2553–2561.
[54] Duyu Tang, Bing Qin, and Ting Liu. 2015. Document Modeling with Gated Recurrent Neural Network for Sentiment Classification.. In *EMNLP*. 1422–1432.
[55] TensorFlow. 2017. TensorFlow. (2017). https://www.tensorflow.org/
[56] Christoph Treude, Ohad Barzilay, and Margaret-Anne Storey. 2011. How do programmers ask and answer questions on the web?: Nier track. In *ICSE*. IEEE, 804–807.
[57] Venkatesh Vinayakarao, Anita Sarma, Rahul Purandare, Shuktika Jain, and Saumya Jain. 2017. Anne: Improving source code search using entity retrieval approach. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*. ACM, 211–220.
[58] Ellen M Voorhees et al. 1999. The TREC-8 Question Answering Track Report.. In *Trec*, Vol. 99. 77–82.
[59] 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 Workshop on Mining Software Repositories*. ACM, 391–402.

[60] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alex Smola, and Eduard Hovy. 2016. Hierarchical attention networks for document classification. In *Proceedings of NAACL-HLT*. 1480–1489.

[61] Pengcheng Yin and Graham Neubig. 2017. A Syntactic Neural Model for General-Purpose Code Generation. In *ACL*. Vancouver, Canada.

[62] Annie T. T. Ying. 2015. Mining Challenge 2015: Comparing and combining different information sources on the Stack Overflow data set. In *The 12th Working Conference on Mining Software Repositories*. to appear.

[63] Zhi-Hua Zhou and Ming Li. 2005. Tri-training: Exploiting unlabeled data using three classifiers. *IEEE Transactions on knowledge and Data Engineering* 17, 11 (2005), 1529–1541.

[64] 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.