A Neural Question Answering System for Subroutines

Author One, Author Two, Author Three, and Author Four
{one,two,three,four}@anonymous
Anonymous
Anonymous
Anonymous, Anon, Anon

ABSTRACT

A question answering (QA) system if a type of conversational AI that generates natural language answers to questions posed by human users. QA systems often form the backbone of interactive dialogue systems, and have been studied extensively for a wide variety of tasks ranging from restaurant recommendations to medical diagnostics. Dramatic progress has been made in recent years, especially from the use of encoder-decoder neural architectures trained with big data input. In this paper, we take initial steps to bringing state-of-the-art neural QA technologies to Software Engineering applications. We target the problem of QA about subroutines in source code, a common information need for SE tasks such as API learning and program comprehension. We curate a training dataset of 10.9 million question/context/answer tuples based on rules we extract from recent empirical studies. Then, we train a custom neural QA model with this dataset and evaluate the model in a study with professional programmers. We demonstrate the strengths and weaknesses of the system, and lay the groundwork for its use in eventual dialogue systems for software engineering.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**;

KEYWORDS

neural networks, question/answer dialogue, artificial intelligence

ACM Reference Format:
Author One, Author Two, Author Three, and Author Four. 2020. A Neural Question Answering System for Subroutines. In *ICPC'20, May 23-24, 2020, Seoul, South Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/00.0000/0000000.0000000

1 INTRODUCTION

A question answering (QA) system is a type of conversational AI that focuses on generating natural language answers to questions posed by human users. QA is defined as single-turn dialogue, in that there are only two participants in the conversation (the human and the machine) and each participant speaks for only one turn (the human asks a question which the machine answers). In practice, a complete conversational machine agent would discuss several topics over an arbitrary number of turns, detect when a question has been asked, and the use a QA system to generate an answer to the question. Thus, QA systems are key components necessary for building usable conversational agents.

In general, QA systems generate an answer given a context about which the question is being asked. For example, a Yin *et al.* [60] describe an approach that parses a knowledge base of facts about famous people to generate English answers about birthdates, political offices held, awards received, etc. Malinowski *et al.* [38] present a system that answers questions about images, such as which objects are red or green in the image. Weston *et al.* [56] provide a dataset of twenty tasks for training QA systems (the so-called bAbI tasks) ranging from positional reasoning to path finding, for which the context is a knowledge base of facts about objects and how they relate to each other (e.g. Context: 1. Lily handed the baby to Philip. 2. Philip walked outside. Question: Where is the baby? Answer: Outside with Philip.).

As the above examples show and as chronicled in several survey papers [11, 18, 34], scientific literature from the areas of Natural Language Processing (NLP) and AI is replete with QA systems designed to answer questions about a context. The overall structure of these approaches is fairly consistent: A large dataset is collected including question, answers, and related contexts. Then a model is trained and tested using the dataset. Typically, a neural model of the encoder-decoder design is employed, in which the model learns to connect features in the questions to features in the context via an attention mechanism. However, there are always numerous domain-specific customizations required to model the context (as a general rule, the question and answer can be modeled using language features such as from a recurrent neural net). For example, Malinowski's work connecting words in questions to features in images uses a typical RNN-based model of questions and answers, but depends on a custom model for extracting those features from the images [38]. In short, the key difficulty in implementing QA systems boils down to: 1) obtaining a proper dataset, and 2) designing a suitable domain-specific model of the context.

In this paper, we present a QA system for answering programmer questions about subroutines in programs (the subroutines are the context about which questions are asked). We construct a dataset of programmer questions based on recent experimental results released by Eberhart *et al.* [17] – that paper isolated five types of questions that programmers asked about Java methods during actual programming tasks. For example, "what are the parameters to the method convertWavToMp3?" We built question and answer templates and paraphrases based on these question types, to construct a dataset of questions and answers for 1.56m Java methods. We then designed a custom QA system based on a neural encoder-decoder model. We model the subroutine context as an Abstract

ICPC'20, May 23-24, 2020, Seoul, South Korea

ACM ISBN 000-0-0000-0000-0/00/00…$15.00
https://doi.org/00.0000/0000000.0000000

Syntax Tree (AST), motivated by recent models of source code [2, 3] and using an AST flattening encoding described at ICSE'19 [28].

We evaluated our work in two ways. First, we used automated metrics over a large testing set of around 67k Java methods, to estimate how our approach would generalize. Second, we performed an experiment with 20 human experts, to determine how well our model responds to actual human input for a subset of 100 methods out of the 67k test set. We explore evidence of *how* our model learns to recognize pertinent facts in source code and generate readable English responses (in the spirit of explainable AI [44, 49]).

2 PROBLEM, SIGNIFICANCE, SCOPE

The problem definition of this paper is fairly straightforward: given a natural language question from a programmer about a program subroutine, we seek to provide a natural language answer to that question. This is referred to as a "question answering system" or QA system in the relevant NLP and AI literature [11, 18, 34, 38, 60]. A QA system involves single turn dialogue: one question from a user and one answer from the machine. This is distinct from other conversational AI such as task-oriented or open-ended dialogue.

A predictable critique of this paper is that programmers probably would not use a QA system alone for basic informational questions about source code. After all, the return type, parameter list, etc., of a function is readily available from reading the source code or summarizing documentation. However, it is important to recognize that a QA system is usually not intended to be used on its own. Instead, a QA system for these questions is a key component in the big picture of conversational AI systems for programmers. Robillard, with thirteen co-authors leading in the area of program comprehension, make the case clearly in a paper summarizing the outcomes of a relevant workshop in 2017 [48]: they "advocate for a new vision for satisfying the information needs of developers" which they call on-demand developer documentation. The idea is that we as a research field should move towards machine responses to programmer information needs that are customized to that programmers' software context and individual questions. But to get to that point, we (the research community) need to solve a few smaller problems that are currently barriers to continued progress. This argument mirrors those made repeatedly in the AI research community generally [24, 55], that smaller problems must be solved and used as a wedge against larger ones, towards the long-term goal of a meaningful conversational AI.

A QA system for basic programming information about subroutines is one of those wedge problems in program comprehension. A successful system would not only answer the narrow problem at hand, but offer insights into issues of how to model and extract features from source code, how to interpret programmer information needs, and how to understand the vocabulary that programmers use that is different from general word use. In the long run, our plan is to include this work as part of a larger interactive dialogue system for helping programmers read and understand source code[1].

3 BACKGROUND & RELATED WORK

This section covers background technologies and closely-related work in both NLP/AI and SE research venues.

[1]Some citations omitted to comply with double-blind review policy.

3.1 Interactive Dialogue Systems

The anatomy of an interactive dialogue system is neatly articulated in a recent book by Rieser and Lemon [46] and summarized in Figure 1 below. There are essentially four components. First, a knowledge base is created to hold information relevant to the conversation, such as images about which questions are asked [38], or maps about which directions may be obtained [19, 31, 59], or restaurants which may be recommended [30]. Second, a natural language understanding component is responsible for converting incoming text into an internal representation of what was said. Often this starts with labeling the text with a dialogue act type [7, 9, 13, 26, 58] (e.g., as a question, a followup statement, a positive or negative comment). But it also includes extracting relevant information necessary to form a response. For example, whether a user wants to know about the return type or parameter list of a subroutine.

The third component is dialogue strategy management. This component decides how to respond as well as how to extract information necessary to make the response. It uses the knowledge base to help make this decision and searches the knowledge base for information relevant to the response. Note that the notion of "strategy" refers to the decision-making process that the machine follows, and is distinct from the natural language in the conversation [21]. For example, if presented with a comment about the weather, some agents would respond with a summary of the predicted weather, some would respond with a suggestion to take an umbrella, while still others would ask a question about the user's preference for summer or fall. But the decision about how to respond is not related to the words actually used to render a response.

Fourth, natural language generation techniques lie along a spectrum, one extreme of which is a templated, rule-based approach [45] while the other extreme is a purely data-driven (usually deep learning-based) approach [15]. An example of a hybrid system is one in which canned responses are used to train a neural net (which allows more flexible combinations of the responses), or data-driven selection from a set of candidate template responses. For a time, there was a belief that language understanding, strategy, and generation could be combined into a single module based on deep learning, but that belief is in strong decline for most applications [19, 21, 55].

QA systems fit into this anatomy of interactive dialogue systems in two ways. First, as mentioned above, a conversational system providing ongoing discussion with a user may include several subsystems to handle different situations, and pass control to a QA

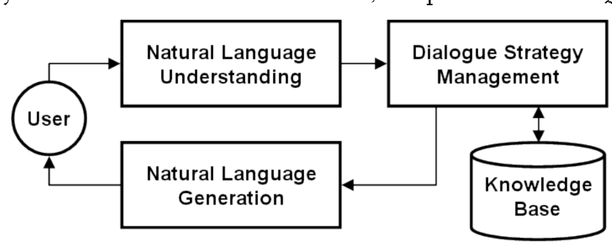


Figure 1: Stereotyped dialogue system described by Rieser and Lemon [46]. In this paper, the knowledge base consists of the source code of subroutines, while the understanding and generation components are learned via a neural net from a dataset we create. We pre-define the strategy based on experimental findings reported by Eberhart *et al.* [17].

subsystem from time to time. Second, a QA system itself generally follows the same design. The strategy component tends to be simpler than most systems because of the assumption that a single question from a user will be provided a single answer. However, a QA system may need to cope with different types of questions and extract information from different types of artifacts – both decisions that fall into the category of strategy. In practice, the strategy tends to be encoded into the model based on dataset design, rather than manual modification of the model.

Research into dialogue systems for software engineering is generally either foundational / dataset generation and analysis, or implementations of experimental dialogue systems. Key foundational and dataset analysis work includes Maalej *et. al* [36], Eberhart *et al.* [17], and several others [1, 22, 37, 40, 41, 47, 58]. A recent survey discusses dialogue systems in SE [4], which includes several experimental systems [8, 25, 43, 53]. These systems are related to this paper in the sense that they are prototypes of dialogue systems for SE problems, but are not directly comparable because they solve very specific problems and are based on scrutiny of highly-specialized domain knowledge. While one may perform quite well in one situation, it is almost guaranteed to fail for other situations. This specialization is typical of dialogue systems in all domains [46], so the way to evaluate an approach is to compare implementation alternatives rather than different dialogue systems [6, 54].

3.2 Neural Encoder-Decoder

Our approach is based on the neural encoder-decoder model. This model is the current standard for QA systems, as described in several surveys [11, 15, 18]. To pick one very recent and related paper that exemplifies how dialogue systems based on the encoder-decoder model work, consider Lin *et al.* [32]. The paper presents a new memory model to augment the encoder of a typical encoder-decoder design, then compares it to alternative encoder-decoder models over publicly-available datasets. This paper is similar, except that rather than a model tuned for general conversations, we propose an encoder model specific to this SE problem, and focus on SE domain knowledge gained via our evaluation.

The encoder-decoder design itself has been clearly described in many papers, and we discuss details in our approach section. In general, the design includes an encoder, which receives as input the natural language from the user plus the knowledge base. The encoder outputs a vector representation of the input natural language, usually via a recurrent net (RNN). The decoder receives the example desired output during training. It generates a vector representation of this desired output. During inference, the model outputs one word at a time of the language to be sent to the user. The decoder receives the output predicted "so far" and uses it to help the model predict the next word.

The encoder-decoder design ballooned in popularity after Bahdanau *et al.* [5] introduced an "attentional" variant that allows the decoder's vector representation to focus on sections of the encoder's representation during training, i.e. to create a dictionary of words in one language in the decoder to another language in the encoder. Specific designs such as the famed seq2seq model have motivated thousands of papers, well beyond what we can describe in this section. Thus we direct readers to several surveys [42, 50, 61]. Within software engineering literature, the encoder-decoder design is seeing increased use for tasks such as code completion [20], code summarization [28], and automated repair [12].

4 APPROACH

Our approach aligns with the related work described in the previous section: the overall architecture is based on the dialogue system design in Figure 1, and the implementation is based on a neural encoder-decoder model. The key novelty in the model is the representation of the knowledge base. The key novelty in the overall architecture is the crafting of our dataset to train the neural model. These set up the novelty of the evaluation, which is showing how these models work in a QA system for program comprehension of functions. In the long run, we plan for this QA system to be a component of a much larger dialogue agent, but that agent is beyond the scope of this paper. An overview of the components of our dialogue system follows:

Dialogue Strategy Management Recall that dialogue strategy management involves decisions both on 1) how to respond, and 2) how to extract the information necessary to make a response. For (1), we craft a dataset that includes either types of questions that we found in recently-released simulation experiments with programmers. While those experiments were performed by others, we completed the analysis of the eight questions for this paper. The dataset design represents our manual effort in designing the strategy the system should follow, but the strategy itself will be learned during training and encoded in a neural model. For (2), we use an attention mechanism in our neural model between the input question and the knowledge base, to learn during training which components of the knowledge base pertain to which questions. Details of our dataset design are in Section 4.1. Details of the attention mechanism are intertwined with the neural model in Section 4.3.

Knowledge Base The knowledge base consists of the source code of the subroutines. We use a collection of Java methods provided by Linstead *et al.* [33] and further processed by LeClair *et al.* [29]. In total, the knowledge base includes 2.1m Java methods from over 10k projects. We represent each subroutine as an abstract syntax tree (AST). Then, we use a graph neural network to model each subroutine's AST and provide a vectorized representation of the subroutine. We train this GNN while we train the other components of the neural model (i.e. it is supervised by the dataset we create, we do not pretrain it using an unsupervised procedure). We were inspired to use an AST representation by recent work in code summarization [3, 23] and we use a flattened tree approach inspired by LeClair *et al.* [28], though our application in this paper is novel. Details of the model of the knowledge base are in Section 4.3.

Natural Language Understanding / Generation We use recurrent neural networks with word embedding vector spaces to implement the encoder and decoder. The encoder is essentially the component that implements the natural language understanding, and the decoder implements the language generation. This structure is closely in line with a vast majority of recent data-driven QA systems (see Section 3.2). We describe details of these components as part of the code implementation in Section 4.3.

4.1 Dataset Preparation

We prepare a dataset that we use to train the neural model described in the next section. This section describes how we structure our dataset so that it represents knowledge about how programmers ask questions and how to respond. Note that while we do not explicitly write rules into our dialogue strategy management, this dataset contains those rules implicitly from which the neural model learns later. We mention this in order to be clear that we do not merely feed the network all data collected during empirical studies and expect the model to learn proper behavior, and to justify our overall posture towards dataset design: the decisions we make in creating the dataset *are* the decisions that will be encoded as dialogue strategy management.

We build the rules for generating our dataset based on empirical data made available to us on pre-release. Eberhart *et al.* [17] conducted an experiment in which 30 programmers solved programming challenges with the help of a simulated interactive dialogue agent (a so-called "Wizard of Oz" study design). The authors of that paper then annotated each question asked by programmers with one of twelve types of API information needs (these twelve types of API info needs were determined in an earlier TSE paper by Maalej and Robillard [36]). Eberhart *et al.* found that over 90% of questions fell into one of three information needs: `functionality`, `patterns`, or `basic`. In the long run, a dialogue agent will need to handle all three types of question. But the scope of that challenge is far too much for one paper. Since this is an early attempt at the problem, we focus on `basic` questions which tend to be more self-contained, have concrete single-turn answers, and overall likelier to be answerable with current technology than other categories.

A `basic` question is one in which a programmer asks for key information about the components of code. The "components" were almost always subroutines rather than classes etc. The "key information" included things like the return type, the function parameters, or a high level description (such as a summary comment from JavaDocs). Approximately 20% of the questions asked by programmers in the study by Eberhart *et al.* [17] were `basic` questions.

We (independent of the analysis by Eberhart *et al.*) examined all questions that were labeled `basic`. The first and second authors of this paper created eight categories of `basic` question. The procedure was an open coding process in which the authors labeled each question with a specific information need from a subroutine (since practically all questions were related to subroutines). The authors worked together to resolve disagreements, rather than work independently and compute an agreement metric, in order to ensure maximum reliability of the data[2].

In the end we had eight types of `basic` question. An important distinction is that six of the questions involved **known** subroutines i.e. the programmer already knew he had the correct method for his task. For example, asking what the return type of method `X` is. Three of the questions involved **unknown** subroutines i.e. the programmer did not know if she had the correct method. For example, asking which method takes an int as a parameter and returns a string. We call questions with a known subroutine "type K" and questions with an unknown subroutine "type U."

Type K questions (subroutine known):

(1) What is the return type of *method*?
(2) What are the parameters of *method*?
(3) Give me the definition of *method*.
(4) What is the signature of *method*.
(5) What does *method* do?
(6) Can *method*, *short task description*?

Type U questions (subroutine unknown):

(7) How do I *short task description*?
(8) What method takes parameter type *P* and returns type *R*?

The scope of our QA system only includes type K questions. Type K questions involve a question, answer, and known context, which is in line with what QA models in NLP are equipped to solve (though, those models have not been adapted to source code). Type U questions involve a search process for the correct subroutine, which would include code search and even dialogue between machine and programmer to decide on the correct subroutine. These search tasks are research problems of their own and are too much to include in one paper. Therefore, we confine ourselves to the problem of answering `basic` questions about known subroutines. Integrating code search, grounding dialogue, etc., is an area of our future work to build on this paper.

4.2 Dataset Generation

The next step is to generate a dataset, now knowing the question types. At a high level, what we do is obtain a large repository of Java methods, then generate example questions and answers for each question type using heuristics to automatically extract information from the methods. The repository of Java methods is a set of 2.1m methods already filtered for duplicates and other errors, and paired with summary descriptions, provided at NAACL'19 [29]. We further filtered this dataset for methods with duplicate and non descriptive comments to 1563197 methods.

Generating text for questions (1-4) is straightforward: just extract information from each method e.g. the return type. For question (5), we used the summary description as the answer.

We used the summary description and method name in the question for question type (6), and the answer was simply "yes" or "no." However, for every positive example for each method, we added a negative example to maintain a balanced dataset. This negative example consisted of a random summary description from another method (of a different name, to avoid picking an overloaded method name) in the same project paired with the method. So for each of the 1.56m methods, we had one positive example and one negative example for question type (6).

To limit the vocabulary size, we replaced some information with tokens that direct the output interface to copy the information from the context directly, rather than learn to predict the information as part of the model. We have a token for $<funcode>$ for the answer of question type (3) that is essentially the whole context and is unnecessary for the model to learn to retrieve. So when the model predicts this token, it can simply copy this from the interface. This allows the user to have the same experience while reducing the vocabulary that the model has to learn.

The last step in our dataset generation was to paraphrase each question and answer. The example questions above are the primary form we used based on the underlying empirical data. However,

[2] Agreement metrics quantify reliability, but do not resolve disagreements. Because we ultimately had to make decisions to create a dataset, we elected to resolve disagreements at the cost of a reliability metric, as suggested by Craggs and McGee [14].

there is no guarantee that a programmer will use exactly that language when asking a question, otherwise we could just use a templated QA system and avoid the complexity of a neural model. So we wrote 15-25 paraphrases of each question, and randomly chose one of them when generating questions and answer for each question. The number of details behind the vocabulary replacements and paraphrases would exceed space limitations to print in this paper, but all are available via our online appendix (see Section 7).

To summarize, our procedure is:

for *each of the 1.56m methods M* **do**
  for *each question type T* **do**
    1. randomly select paraphrase template for T
    2. generate question and answer using template
    3. preprocess code of M to serve as context
    4. create 3-tuple: (question, answer, context)
    if *T == 6* **then**
      5. randomly select summary of different method
      6. create 3-tuple: (question, "no", context)

The result of our dataset generation is a set of 10.88 million 3-tuples. Each 3-tuple contains a question, an answer, and a context Java method. For each of the 1.56m Java methods, we generated 7 type K questions and answers (one for question types 1-5, two for question 6). To ensure maximum reproducibility, we maintained the training/validation/test splits provided by LeClair *et al.* [29].

4.3 Neural Model

Rationale The rationale for using a neural model is, essentially, that neural models enable more flexible natural language understanding and generation in fewer steps, without the need for manually-written rule to extract information from context. A traditional alternative to a neural model is a simple approach based on classification of incoming questions and rules to extract information. However, it is important to realize that this seemingly-obvious alternative is not in line with recent work from the NLP research area for context-based Q/A systems. As Wiese *et al.* [57] point out, recent advances in neural models have led to "impressive performance gains over more traditional systems."

In contrast, our model falls clearly in line with related work from the NLP research area on context-based Q/A systems (see Section 3.2): there is an encoder with question and context inputs, and a decoder with the answer input. From an ML perspective, one novel aspect to this paper is that we show how the neural model can learn features in the source code when given only that code as a context, and questions/answers about the context. This is important novelty, along the lines of Wiese *et al.* [57] when they showed how neural Q/A models can learn from biomedical text data versus other highly specific areas e.g. technical support conversations [10] or even religious texts [62]. The point is that domain adaptions are considered important contributions and are not merely applying technology X to data Y.

Overview Our neural model is, at a high level, similar to context-based Q/A systems described in related work and summarized in Section 3.2. The structure of these systems is basically a question and context as "encoder" input and an answer as "decoder" input (during training). The model is trained so that during inference, the model will output one word of the predicted answer at a time. Our model follows this same basic structure. The question and answer are generated for each function as described in the previous section. The context is the source code of the function.

At a technical level, our approach is based on the encoder-decoder model released by LeClair *et al.* [28] at ICSE 2019. We chose that model because: 1) it was designed to accommodate source code as input instead of only text, and 2) a thorough reproducibility package is available. That model was designed to generate natural language descriptions of source code (so-called "source code summarization"). The inputs to the model's encoder were preprocessed source code, a flattened abstract syntax tree. The input for training for the decoder was the example summary.

Our modifications, in a nutshell, are to make the model's encoder inputs the raw source code (not preprocessed), to add an input for the user query/question to the encoder, and to change the decoder's training input to example answers to the questions. We used raw source code instead of preprocessed source code because we are interested in the model's ability to learn where code features are such as the return type, parameters, etc., unlike LeClair *et al.* who were more interested in extracting text features such as identifier names. Their preprocessing steps removed information that we found to be critical in helping the model learn features about code.

Details We explain our model as a walkthrough of our actual Keras implementation to maximize clarity and reproducibility, following the successful example of LeClair *et al.* [28]. The code in this section is in file `qamodel.py` in our online appendix (Section 7.

```
qe = Embedding(output_dim=self.embdims,
input_dim=self.quesvocabsize)(ques_input)
ce = Embedding(output_dim=self.embdims,
input_dim=self.codevocabsize)(code_input)
```

The first step is to create a word embedding space for the question and code encoder inputs. The question vocabulary size we used was 20K, which is typical for text inputs, but we used a much larger vocab size of 100K for the source code context. Programmers tend to use domain specific words that expand the vocabulary.

```
ques_enc = CuDNNGRU(self.rnndims,
return_state=True, return_sequences=False)
quesout, qs = ques_enc(qe)
code_enc = CuDNNGRU(self.rnndims,
return_state=True, return_sequences=True)
codeout, cs = enc(ce, initial_state=qs)
```

We use a GRU to encode the question and source code, with the question and source code embedding spaces serving as input. We set the initial state of the code encoder to the end state of the question encoder, in line with other neural QA model designs in which the question state is used to start the state of the context encoding.

```
ae = Embedding(output_dim=self.embdims,
input_dim=self.ansvocabsize)(ans_input)
aec = CuDNNGRU(self.rnndims,
return_sequences=True)
aout = aec(ae, initial_state=cs)
```

The decoder follows the same basic structure: an embedding space as input to a GRU. The decoder input is the answer. The answer vocab size is 20K.

```
ques_attn = dot([aout, quesout], axes=[2, 2])
ques_attn = Activation('softmax')(ques_attn)
```

```
ques_context=dot([ques_attn, quesout],axes=[2, 1])
code_attn = dot([aout, codeout], axes=[2, 2])
code_attn = Activation('softmax')(code_attn)
code_context=dot([code_attn, codeout], axes=[2, 1])
```

Our attention mechanism consists of attention applied from the decoder (aout) to both the question and source code context. The attention to code context is especially important because this is how the model emphasizes context features – this is chiefly what papers mean when they say that the model "learns to comprehend" the context. We will show in our experimental results how the model learns different code features relevant to different questions.

```
context = concatenate(
[ques_context, code_context, aout])
out = TimeDistributed(Dense(self.rnndims,
activation="relu"))(context)
```

The next step is to create a context matrix by combining the attended question and context matrices with the answer context from the decoder. After this step, the models uses the combined context matrix to predict the next word in the answer.

4.4 Training Procedures

Our training procedure is based on the "teacher forcing" technique [16, 27, 35] in which the model receives only correct examples from the training set and is not exposed to its own errors (the technique helps keep the model reinforcing mistakes). To understand how the procedure works for our approach, recall that an encoder-decoder architecture typically (as in our approach and others related to a seq2seq model) predicts output sequences one item at a time. For example, given a question "what is the return type of function X?", the model would generate an answer by predicting the first word of the answer:

```
[ question ] + [ code ]
    => [ "the" ]
```

Then it would use the first word prediction as a new input to the decoder, to predict the second word:

```
[ question ] + [ code ] + [ "the" ]
    => [ "method" ]
```

And the process would continue to predict the entire response:

```
[ question ] + [ code ] + [ "the method" ]
    => [ "returns" ]
[ question ] + [ code ] + [ "the method returns" ]
    => [ "a" ]
[ question ] + [ code ] + [ "the method returns an" ]
    => [ "unsigned" ]
[ question ] + [ code ] + [ "the method returns an unsigned" ]
    => [ "long" ]
```

Yet this is how the model behaves during inference. To train the model, following the teacher forcing procedure, we provide the model each example one word at a time. So, in the above example, we would provide the model with "the" followed by the reference output "method", then "the method" with the reference output "returns", and so on. If the model makes an incorrect prediction, we use back propagation to correct the model, and then substitute the correct reference output for the next step – the model is not permitted to use its own erroneous prediction as the next input. However, a caveat is that the procedure slows training because each example must pass through the model for every word in the output.

5 EVALUATION

We conduct an experiment with human users to evaluate our QA system. Note that our ultimate intent for this QA system is to serve as a component of a much larger conversational AI (see Sections 2 and 3.1). Therefore, our experimental setup is a controlled environment in which we test specific inputs and outputs generated by human users. We are *not* attempting to evaluate the system "in the wild" because the system is not intended to be used standalone, and because the larger conversational AI system does not yet exist.

5.1 Research Questions

Our research objective is to determine the degree to which our QA system is able to answer the eight questions about subroutines we determined in Section 4.1. We ask the following Research Questions (RQs) towards this objective:

RQ$_1$ What is the performance of our QA system in terms of relevance, accuracy, completeness, and conciseness?

RQ$_2$ How does the performance vary across the six question types for which we designed the system?

RQ$_3$ What features in the context are the most important for the model to use when answering a question?

The rationale behind RQ$_1$ is that good responses by any QA system should score well across at least three degrees: relevance, accuracy, completeness, and conciseness. Accuracy, because independent of any other factors the response should not contain false information. Completeness, because responses should contain *all* information needed to answer the question. Conciseness, because responses should contain only the information necessary to answer a question. We derived these four degrees of text generation quality from related SE literature on code description generation [39, 52]. The rationale behind RQ$_2$ is that the system may perform well for some questions but not others. In particular, it may perform well at extracting information such as the return type of a subroutine, but struggle for other questions such as returning a description of a subroutine. RQ$_3$ relates to the explainability of the neural model. Neural models tend to be highly effective for text comprehension and generation tasks, but are notorious for producing black box responses that are difficult to understand. We ask RQ$_3$ to provide a few insights into the model's behavior, within the constraints of a single conference paper.

5.2 Methodology

Our methodology for answering RQ$_1$ and RQ$_2$ is to conduct a user study in which human programmers evaluate the output of the QA system for questions that they generate. To limit the scope of the experiment, we control the study conditions so that the programmers only ask questions related to information needs we highlight in the six questions in Section 4.1. We recruited professional programmers from around the United States via an online job platform (demographics of study population are in the next section). We also created a web interface with which the programmers could communicate with the QA system. A screenshot of this interface is in Figure 2. The interface also provided a space for the programmers to rate the responses on a 1-4 scale ranging from Strongly Agree, Agree, Neutral, Disagree, or Strongly Disagree for the quality prompts shown in Table 1.

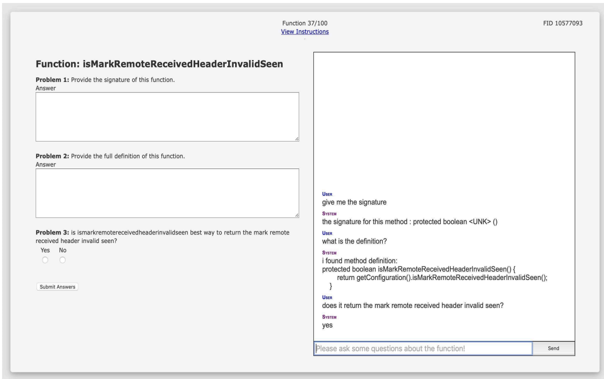


Figure 2: The interface that programmers used to communicate with the QA system during our experiment.

Rationale Our study design is similar to previous experiments by Sridhara *et al.* [52] and McBurney *et al.* [39]. We used similar wording of our prompts to study participants and the same four options. The only difference we made was to add another option for Neutral in case the model returns a nonsensical reply (which can happen for our neural model but was very unlikely in the templated systems of code comment generation in those previous studies). We added the Neutral option as a middle ground between the four main choices, to avoid forcing participants to make decisions on possible nonsensical responses.

Another similarity is that we find ourselves in the same situation as Sridhara *et al.* [51] in their ASE paper: no baseline exists for comparison. To our knowledge, no QA system has been designed to answer these specific questions in a natural language format. Different tools do exist for some questions. For example, question (5) could be thought of as a code summarization question, while questions (1-4) could be answered by just reading the subroutine itself. Yet recall that we are not seeking an "in the wild" evaluation – we need to evaluate the input and output of the model in situ with the natural language understanding and generation components of the approach. Therefore, we follow the example of these earlier papers and focus on a deeper analysis of the responses across multiple quality criteria, instead of comparing metrics across competing approaches (since they do not yet exist).

Table 1: Quality prompts (P1-5) in the user study. These correspond to the quality criteria (relevance, accuracy, completeness, and conciseness) discussed in Section 5.2. The first four prompts are answerable as "Strongly Agree", "Agree", "Neutral", "Disagree", or "Strongly Disagree."

P_1	Independent of other factors, the response is relevant to my question, even if the information it contains is inaccurate.
P_2	The response is accurate, even if it is not relevant to my question.
P_3	The response is missing important information, and that can hinder my understanding.
P_4	The response contains a lot of unnecessary information.
P_5	Do you have any general comments about the response?

Note also that we do not use BLEU scores or other automated metrics. A human study is vital for two reasons. First, we need to evaluate specific subjective qualities rather than an overall similarity metric to a ground truth (like BLEU would do). Second, the ground truth in our dataset (i.e. the answer component of the question, context, answer tuples, see Section 4.1) is generated by us. We use it as training data, but it would not be appropriate to use as testing data since it would include our own biases.

Experiment Procedure In the experiment, we gave each programmer a "quiz" to fill out with the assistance of the QA system (see Figure 2). Each page of the quiz gave the name of a particular Java method. Only the method name was shown, not the method body. For each method, three Type K questions (see Section 4.1) were chosen randomly. Below the method name, there were three prompts, derived from the chosen Type K questions. We phrased the prompts as imperative statements (e.g. "Provide the return type of this function") to avoid priming the programmers with a particular question format. We instructed programmers to use their own words to ask for information from the QA system. We asked programmers not to copy questions, but we allowed them to copy answers from the QA system for the quiz. A programmer could ask the QA system as many queries as he or she wanted.

After answering the question prompts for a particular method, programmers were brought to a new page that asked them to rate each of their interactions with the QA system for that method. For each interaction (consisting of a user query and the QA system's response), we asked the programmers to answer the five quality prompts listed in Table 1. When they were done, they could press a button to bring up the next method, and a new set of prompts.

In short, we used a quiz format to encourage programmers to ask the QA system certain types of questions, but in their own words. Then they rated the responses using the quality prompts. They also completed the quiz, so we could determine whether they obtained the correct information in the end, independent of how what ratings they chose for the quality prompts. Space constraints prevent us from including the quiz and other materials, but we provide these via our online appendix (Section 7).

For clarity in the experimental results section, we use the following vocabulary to refer to the various parts of our study: 1) a "question", Q1-6, is one of the six Type K question types we use in our experiment and described in Section 4.1, 2) a "query" is text typed by the user into the experiment interface separated by striking the return key, since hitting the return key triggers the interface to send the text to the prediction model and receive an answer back, and 3) a "quality prompt", P1-5, is one of the requests we make of users to rate the model's answer. The users see three questions per function. They may write as many queries as they wish to help them answer each question. Then they respond to five quality prompts for each answer they see to a query.

5.3 Participants

We recruited 30 participants for our experiment. These participants had professional experience ranging from three to 15 years. We compensated programmers at a flat rate of US$60/hr, market rate in our region, regardless of performance speed. Each programmer worked for a total of 40 minutes to answer as many quiz pages as possible in that time.

5.4 Subject Java Methods

We used a total of 100 Java methods in our experiment. We sourced these methods from the test set of the dataset split – the model had not seen them during training. We rotated these at random so that no programmer saw the same method more than once, but that each method was shown to at least three programmers. But given the vicissitudes of any study with humans (fatigue, differing speeds, skipped pages), not all methods ended with three ratings.

5.5 Threats to Validity

Like any paper, this experiment carries threats to the validity of its conclusions. One threat is the dataset we use. One of the disadvantages of human studies is that the number of functions that we could ask any one person to evaluate is quite limited – we cannot merely calculate a metric over thousands of examples. We chose a random selection from a large, curated dataset, and we ensured that each function was seen by more than one person, but it is still possible that a different selection would result in a different result. Likewise, another threat is that a different set of programmers might give different answers. We attempted to mitigate this risk by asking over 20 participants. Also, we attempted to mitigate a risk of varying results from the model itself by ensuring consistent random seeds and experimental conditions (all available via our online appendix), though it is always a risk that random factors in GPU hardware or software could lead to slightly different results.

6 EXPERIMENTAL RESULTS

This section describes the results of the experiment: our answers to our RQs and supporting evidence.

6.1 RQ_1: Overall Performance

In general, we found the model's overall performance to be good. Figure 3 gives an overview. The figure is a histogram of all user answers to the quality prompts from Table 1. Recall that 1="Strongly Agree", 2="Agree", 3="Neutral", 4="Disagree", and 5="Strongly Disagree" to the prompt text. Prompts 1 and 2 are worded positively (so agreement is better) while prompts 3 and 4 are worded negatively (so disagreement is better). For example, for P_1 about how relevant the response is, a vast majority of responses received a score of Strongly Agree or Agree. Likewise, for P_3, a vast majority of responses received a score indicated disagreement to a prompt about missing important information. Also note that only a small percent of responses were rated as neutral, meaning that, in general, responses were clear enough for participants to form an opinion – upon inspection a vast majority of responses rated as neutral were gibberish output from the neural model. Still, in terms of overall performance, the model does tend to generate reasonable responses.

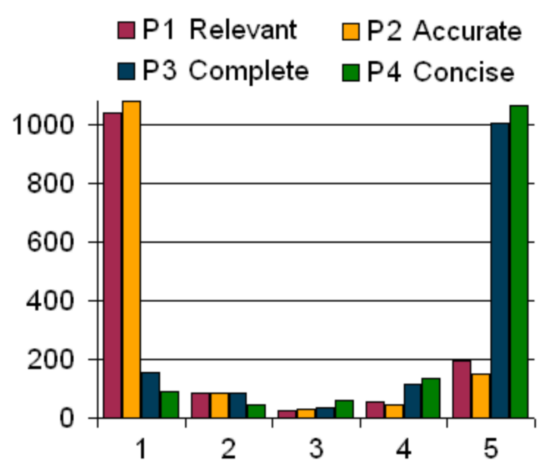


Figure 3: Histograms of the user responses to the quality prompts in Table 1. Recall that P1 and P2 are asked in a positive tone (so 1-2 scores are better) while P3 and P4 are in a negative tone (so 4-5 are better). Participants tended to find the model's responses to be of good quality.

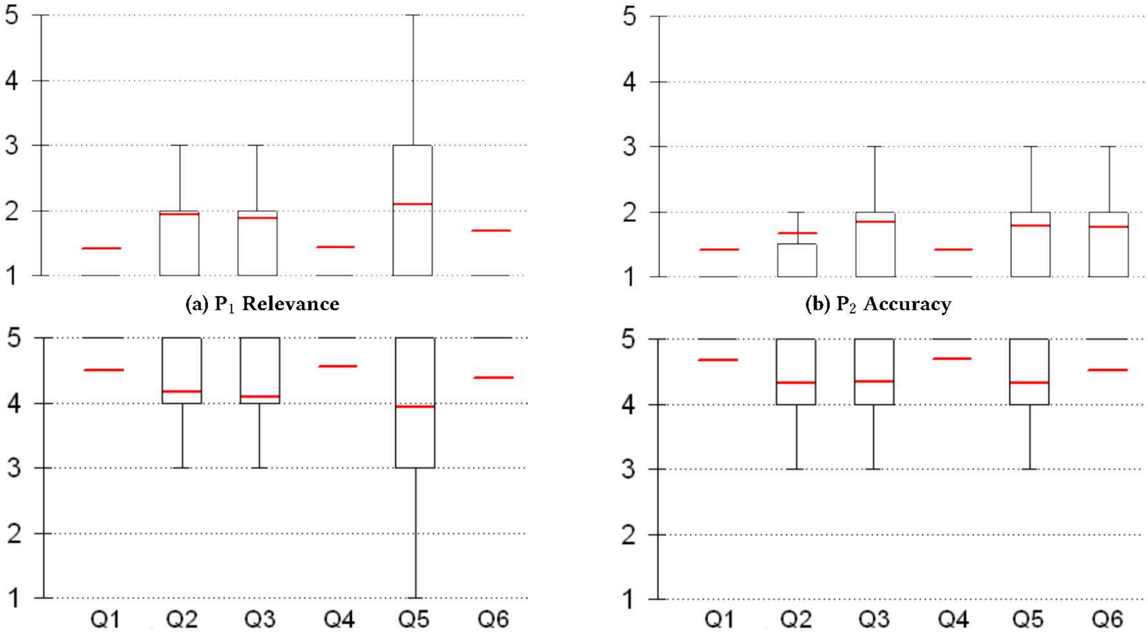
Table 2: Performance statistics of participants in the experiment. Each participant worked for 40 minutes. We asked three questions for each method. However, participants worked at their own speeds and were allowed to ask any number of queries they wanted.

	Mean	Min	Max
Methods Evaluated per Participant	19	7	38
Queries per Participant	70	37	117
Queries per Method	3.8	2	10
Queries until "Correct" Response	1.2	1	8

Two caveats should be understood. First, different participants worked at different rates, so some participants are represented more in the data than others. Table 2 quantifies these differences. Almost all participants evaluated between 15 and 20 methods, but there were a few outliers as is natural in samples of human populations (mean speed of 19 methods per 40 minute study is about 2 minutes per method, while 38 is a rate of about 1 minute/method). Nonetheless we found the number of queries required to answer each question to be quite stable, with one query usually sufficing and two or more queries being quite rare. In other words, the time required by each participant seemed to have more to do with time required by the participant to read and understand the questions, than with the number of queries required per participant.

Second, the responses to each quality prompt are independent of other prompts. So it is possible that a response receives a good score for P_2 and a poor score for P_1, i.e. the response is accurate but not relevant. To study this caveat, we derived a metric we call "correctness" by combining P_1 and P_2 scores. The metric is binary. A response receives a 1 if and only if both P_1 and P_2 scores are one or two – that is, a response is only "correct" if the participant strongly agrees or agrees that it is both relevant and accurate. We found that 79% of responses were "correct" and that it usually took only one query to receive a correct response.

We found that a key factor in the 21% of incorrect responses to be the vocabulary size. As mentioned in Section 4.3, GPU memory limitations restrict both the input and output vocab size, despite our attempts to extend these by using GPUs with 16gb VRAM and low training batch sizes. This limit affected our results. A vast majority of the responses that were relevant but not accurate were ones with UNK tokens in the answer. Likewise, responses that were accurate byt not relevant almost always had UNK tokens in the question (i.e. the participant wrote a query with out-of-vocab words in it) – these UNK tokens likely caused the model to misunderstand the question and give an accurate response that was nonetheless irrelevant.



(a) P_1 Relevance (b) P_2 Accuracy (c) P_3 Completeness (d) P_4 Concision

Figure 4: Boxplots of answers to quality prompts (relevance, accuracy, completeness, and concision) for each of the question types from Section 4.1. The model performs differently for different question types. Performance is highest for Q1 and Q4 and worst for Q5: ratings of relevance and completeness tend to be worse for Q5 than for other question types.

6.2 RQ$_2$: Variation among Question Types

We observe a small degree of variation among the question types in our experiment. Recall from Section 4.1 that we have a variety of question templates that we derived from six different question types corresponding to six key information needs programmers have. (In the experiment, we confined participants to these information needs, but we had no restriction on the language that they could use to render a question.) Recall that the rationale of this RQ is that the model may be better at understanding some information needs than others. For the convenience of understanding the results in this section and Figure 4, we reprint the question types below:

Q1 What is the return type of *method*?
Q2 What are the parameters of *method*?
Q3 Give me the definition of *method*.
Q4 What is the signature of *method*.
Q5 What does *method* do?
Q6 Can *method, short task description*?

Figure 4 contains boxplots of the answers for each quality prompt, divided across each question type. For example, column Q2 of Figure 4(a) shows that for Question 2, the mean of all responses to queries is about 2 (the red line), the interquartile range is 1 to 2. The way to interpret this is that, among all queries written for Q2, participants either Strongly Agreed or Agreed that the query was relevant about half of the time. Note that outliers are excluded for readability, but we did have at least one instance of each score.

In general, the model performs very well for Q1 and Q4. For both question types, the responses are dominated by optimal scores (1 for relevance and completeness, 5 for accuracy and concision). This result implies that the model is successfully learning to recognize when participants were asking for those information needs, and also learns how to extract that information from the source code and place it in a natural language response. Q1 and Q4 correspond to the return type and signature of the source code. We will show in the next section how the model learns to find this information in source code quite reliably.

The model performed slightly less well for Q2 and Q3, for which the model learns to find the method parameters and definition. These information needs are slightly more difficult to learn because they vary more in size and vocabulary. The return type (Q1) can always be found in the same place at the start of the method signature, it is always exactly one word long, and the vocabulary is limited to type names. The parameter list can also always be found in the same place, but it varies in length and includes identifier names that may be specific to that method. Thus the model struggles slightly more to learn to find it.

The model performs the least well on Q5, especially in terms of relevance and completeness. This result may be expected, however, since the model is expected to provide a short description of the method's behavior. We do give the model a short description in the context (see an example in the next section), and the model does learn to use this description in its response. But the size and vocabulary of the description vary considerably, and the model is prone to use incorrect words not in the actual description. In addition, the description we provide originates in the JavaDocs, which could have varying quality.

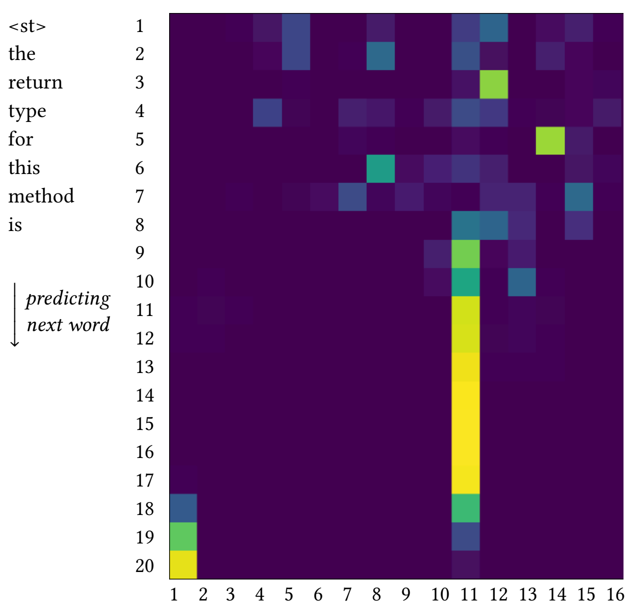
The responses to Q6 are a bit of a special case since they are always either "yes" or "no." Therefore it is relatively easy for the model to score well in terms of e.g. relevance. Still, the model is sometimes wrong, which is reflected in column Q6 of Figure 4(b).

6.3 RQ$_3$: Effects of Context Features

We provide evidence of the effects of the features in the context via an example of the model's behavior. "Explainable AI" is a highly controversial topic, which much agreement that it is necessary but practically no consensus on the best strategies – neural networks in particular have a reputation for producing results that are difficult to explain [44, 49]. However, one source of evidence is the attention network. The attention mechanisms in most encoder-decoder models, include ours, are responsible for connecting pieces of the decoder inputs to pieces of the encoder inputs. Frequently attention provides clues on why the model makes a particular decision. For example in NMT the word "hund" in a German sentence will receive high attention to the English word "dog", while in computer vision the word "dog" may receive high attention to the area in an image where a dog appears.

In our approach, the attention mechanism connects output words to words in the input context sequence. Consider Example 1 below. The user study participant writes a query requesting the return type. The heatmap shows the state of the attention network just prior to predicting the word "vertex" (layer `code_attn` from Section 4.3, recall from Section 4.4 that the model predicts output one word at a time). This example is typical of almost all queries about the return type: the model has learned where to find the return type in code. Of course, it is not always in position 11, but the model "knows" to look for the signature, and where to look in the signature for the return type. Note that the model is not attending to earlier uses of the word vertex in the method description, since that wording may change. Likewise, it is not attending to the word vertex in after the actual return in the code, since that is a variable name which may not be the actual return type. The model has learned these patterns from the training set.

user query Give me the return type
model output the return type for this method is vertex
context
```
<st> returns the next vertex of a polygon nl
public vertex nextvertex ( vertex v ) {
int ind = vertices . indexof ( v ) ;
return ( vertex ) ( ind == -1 ?
null : vertices . get ( ( ind + 1 )
% vertices . size () ) ) ; } <et>
```



Example 1: User study participant asking for the return type of the method. The model creates a response based on the query and the source code sequence (including summary). A heatmap of the attention network shows how the model attends heavily to the word "vertex" in the context (position 11) just prior to predicting the last word in the output.

While space restrictions prevent us from printing numerous examples, we include several more in our online appendix cited below. The behavior is quite consistent: for queries about e.g. parameters, the model attends to the parameters area of the signature, and outputs the relevant information.

7 CONCLUSION

We have presented a QA system for programmer questions about subroutines. We design a neural model based on the encoder-decoder structure that can extract information about Java methods directly from the source code of those methods. We designed our system to distinguish between and answer questions for six different information needs, which we derived from recent related work on dialogue systems for programmers. In an experiment with 20 professional programmers, we show that our approach is able to reliably answer these six questions.

Throughout our paper, we note that this QA system is not intended for use on its own. Instead, it would serve as a component of a hypothetical much larger interactive dialogue system. Virtual agents are anticipated for many tasks including as assistants for software engineering. However, it is unreasonable to expect to create such a system in one step – research into subsystems and supporting components is required first. This paper fills that role towards virtual agents for SE tasks. Important next steps include both designing other subsystems and expanding the number of question types that this QA system is able to handle.

To promote continued research, we release all our data, approach source code, and a working interactive demonstration via our online appendix:

https://github.com/paqs2020/paqs2020

8 ACKNOWLEDGMENTS

Redacted to comply with double-blind review process.

REFERENCES

[1] Emad Aghajani, Csaba Nagy, Gabriele Bavota, and Michele Lanza. 2018. A large-scale empirical study on linguistic antipatterns affecting apis. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 25–35.

[2] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to represent programs with graphs. *International Conference on Learning Representations* (2018).

[3] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating sequences from structured representations of code. *International Conference on Learning Representations* (2019).

[4] Venera Arnaoudova, Sonia Haiduc, Andrian Marcus, and Giuliano Antoniol. 2015. The use of text retrieval and natural language processing in software engineering. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*. IEEE Press, 949–950.

[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] Nicole Beringer, Ute Kartal, Katerina Louka, Florian Schiel, Uli Türk, et al. 2002. PROMISE: A procedure for multimodal interactive system evaluation. In *Proceedings of the WorkshopâĂŹMultimodal Resources and Multimodal Systems Evaluation*. Citeseer, 90–95.

[7] Phil Blunsom, Nal Kalchbrenner, and Nal Kalchbrenner. 2013. Recurrent convolutional neural networks for discourse compositionality. In *Proceedings of the 2013 Workshop on Continuous Vector Space Models and their Compositionality*. Proceedings of the 2013 Workshop on Continuous Vector Space Models and their Compositionality.

[8] Nick C Bradley, Thomas Fritz, and Reid Holmes. 2018. Context-aware conversational developer assistants. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 993–1003.

[9] Susanne Burger, Karl Weilhammer, Florian Schiel, and Hans G Tillmann. 2000. Verbmobil data collection and annotation. In *Verbmobil: Foundations of speech-to-speech translation*. Springer, 537–549.

[10] Vittorio Castelli, Rishav Chakravarti, Saswati Dana, Anthony Ferritto, Radu Florian, Martin Franz, Dinesh Garg, Dinesh Khandelwal, Scott McCarley, Mike McCawley, et al. 2019. The TechQA Dataset. *arXiv preprint arXiv:1911.02984* (2019).

[11] Hongshen Chen, Xiaorui Liu, Dawei Yin, and Jiliang Tang. 2017. A survey on dialogue systems: Recent advances and new frontiers. *Acm Sigkdd Explorations Newsletter* 19, 2 (2017), 25–35.

[12] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering* (2019).

[13] Zheqian Chen, Rongqin Yang, Zhou Zhao, Deng Cai, and Xiaofei He. 2018. Dialogue Act Recognition via CRF-Attentive Structured Network. In *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*. ACM, 225–234.

[14] Richard Craggs and Mary McGee Wood. 2005. Evaluating discourse and dialogue coding schemes. *Computational Linguistics* 31, 3 (2005), 289–296.

[15] Li Deng and Yang Liu. 2018. *Deep Learning in Natural Language Processing*. Springer.

[16] Kenji Doya. 2003. Recurrent networks: learning algorithms. *The Handbook of Brain Theory and Neural Networks,* (2003), 955–960.

[17] Zachary Eberhart, Aakash Bansal, and Collin McMillan. 2020. The Apiza Corpus: API Usage Dialogues with a Simulated Virtual Assistant. arXiv:cs.SE/2001.09925

[18] Jianfeng Gao, Michel Galley, Lihong Li, et al. 2019. Neural approaches to conversational AI. *Foundations and Trends® in Information Retrieval* 13, 2-3 (2019), 127–298.

[19] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. 2016. Hybrid computing using a neural network with dynamic external memory. *Nature* 538, 7626 (2016), 471.

[20] Shirley Anugrah Hayati, Raphael Olivier, Pravalika Avvaru, Pengcheng Yin, Anthony Tomasic, and Graham Neubig. 2018. Retrieval-Based Neural Code Generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. 925–930.

[21] He He, Derek Chen, Anusha Balakrishnan, and Percy Liang. 2018. Decoupling Strategy and Generation in Negotiation Dialogues. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. 2333–2343.

[22] Andrew Head, Caitlin Sadowski, Emerson Murphy-Hill, and Andrea Knight. 2018. When not to comment: questions and tradeoffs with API documentation for C++ projects. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 643–653.

[23] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th International Conference on Program Comprehension*. ACM, 200–210.

[24] Matthew Johnson and Alonso Vera. 2019. No AI Is an Island: The Case for Teaming Intelligence. *AI Magazine* 40, 1 (2019), 16–28.

[25] Andrew J Ko and Brad A Myers. 2004. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 151–158.

[26] Harshit Kumar, Arvind Agarwal, Riddhiman Dasgupta, Sachindra Joshi, and Arun Kumar. 2018. Dialogue Act Sequence Labeling using Hierarchical encoder with CRF. *AAAI* (2018).

[27] Alex M Lamb, Anirudh Goyal Alias Parth Goyal, Ying Zhang, Saizheng Zhang, Aaron C Courville, and Yoshua Bengio. 2016. Professor forcing: A new algorithm for training recurrent networks. In *Advances In Neural Information Processing Systems*. 4601–4609.

[28] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 795–806.

[29] Alexander LeClair and Collin McMillan. 2019. Recommendations for Datasets for Source Code Summarization. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. 3931–3937.

[30] Oliver Lemon. 2011. Learning what to say and how to say it: Joint optimisation of spoken dialogue management and natural language generation. *Computer Speech & Language* 25, 2 (2011), 210–221.

[31] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2015. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493* (2015).

[32] Zehao Lin, Xinjing Huang, Feng Ji, Haiqing Chen, and Ying Zhang. 2019. Task-Oriented Conversation Generation Using Heterogeneous Memory Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*.

[33] Erik Linstead, Sushil Bajracharya, Trung Ngo, Paul Rigor, Cristina Lopes, and Pierre Baldi. 2009. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery* 18 (2009), 300–336.

[34] Weibo Liu, Zidong Wang, Xiaohui Liu, Nianyin Zeng, Yurong Liu, and Fuad E Alsaadi. 2017. A survey of deep neural network architectures and their applications. *Neurocomputing* 234 (2017), 11–26.

[35] Antonette M Logar, Edward M Corwin, and William JB Oldham. 1993. A comparison of recurrent neural network learning algorithms. In *IEEE International Conference on Neural Networks*. IEEE, 1129–1134.

[36] Walid Maalej and Martin P Robillard. 2013. Patterns of knowledge in API reference documentation. *IEEE Transactions on Software Engineering* 39, 9 (2013), 1264–1282.

[37] Walid Maalej, Rebecca Tiarks, Tobias Roehm, and Rainer Koschke. 2014. On the comprehension of program comprehension. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23, 4 (2014), 31.

[38] Mateusz Malinowski, Marcus Rohrbach, and Mario Fritz. 2015. Ask your neurons: A neural-based approach to answering questions about images. In *Proceedings of the IEEE international conference on computer vision*. 1–9.

[39] Paul W McBurney and Collin McMillan. 2016. Automatic source code summarization of context for java methods. *IEEE Transactions on Software Engineering* 42, 2 (2016), 103–119.

[40] Michael Meng, Stephanie Steinhardt, and Andreas Schubert. 2018. Application programming interface documentation: what do software developers want? *Journal of Technical Writing and Communication* 48, 3 (2018), 295–330.

[41] Martin Monperrus, Michael Eichberg, Elif Tekes, and Mira Mezini. 2012. What should developers be aware of? An empirical study on the directives of API documentation. *Empirical Software Engineering* 17, 6 (2012), 703–737.

[42] Samira Pouyanfar, Saad Sadiq, Yilin Yan, Haiman Tian, Yudong Tao, Maria Presa Reyes, Mei-Ling Shyu, Shu-Ching Chen, and SS Iyengar. 2018. A survey on deep learning: Algorithms, techniques, and applications. *ACM Computing Surveys (CSUR)* 51, 5 (2018), 92.

[43] Piotr Pruski, Sugandha Lohar, William Goss, Alexander Rasin, and Jane Cleland-Huang. 2015. TiQi: answering unstructured natural language trace queries. *Requirements Engineering* 20, 3 (2015), 215–232.

[44] Gabriëlle Ras, Marcel van Gerven, and Pim Haselager. 2018. Explanation methods in deep learning: Users, values, concerns and challenges. In *Explainable and Interpretable Models in Computer Vision and Machine Learning*. Springer, 19–36.

[45] Ehud Reiter and Robert Dale. 2000. *Building natural language generation systems*. Cambridge University Press, New York, NY, USA.

[46] Verena Rieser and Oliver Lemon. 2011. *Reinforcement learning for adaptive dialogue systems: a data-driven methodology for dialogue management and natural language generation*. Springer Science & Business Media.

[47] Martin P Robillard and Robert Deline. 2011. A field study of API learning obstacles. *Empirical Software Engineering* 16, 6 (2011), 703–732.

[48] Martin P Robillard, Andrian Marcus, Christoph Treude, Gabriele Bavota, Oscar Chaparro, Neil Ernst, Marco Aurélio Gerosa, Michael Godfrey, Michele Lanza, Mario Linares-Vásquez, et al. 2017. On-demand developer documentation. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 479–483.

[49] Wojciech Samek, Thomas Wiegand, and Klaus-Robert Müller. 2017. Explainable artificial intelligence: Understanding, visualizing and interpreting deep learning models. *arXiv preprint arXiv:1708.08296* (2017).

[50] Ajay Shrestha and Ausif Mahmood. 2019. Review of Deep Learning Algorithms and Architectures. *IEEE Access* 7 (2019), 53040–53065.

[51] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. 2010. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 43–52.

[52] Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. 2011. Automatically detecting and describing high level actions within methods. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 101–110.

[53] Yuan Tian, Ferdian Thung, Abhishek Sharma, and David Lo. 2017. APIBot: Question answering bot for API documentation. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 153–158.

[54] Marilyn A Walker, Diane J Litman, Candace A Kamm, and Alicia Abella. 1997. Evaluating interactive dialogue systems: Extending component evaluation to integrated system evaluation. In *Interactive Spoken Dialog Systems: Bringing Speech and NLP Together in Real Applications*.

[55] Nigel G Ward and David DeVault. 2016. Challenges in building highly-interactive dialog systems. *AI Magazine* 37, 4 (2016), 7–18.

[56] Jason Weston, Antoine Bordes, Sumit Chopra, Alexander M Rush, Bart van Merriënboer, Armand Joulin, and Tomas Mikolov. 2015. Towards ai-complete question answering: A set of prerequisite toy tasks. *arXiv preprint arXiv:1502.05698* (2015).

[57] Georg Wiese, Dirk Weissenborn, and Mariana Neves. 2017. Neural domain adaptation for biomedical question answering. *arXiv preprint arXiv:1706.03610* (2017).

[58] Andrew Wood, Paige Rodeghero, Ameer Armaly, and Collin McMillan. 2018. Detecting speech act types in developer question/answer conversations during bug repair. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 491–502.

[59] Caiming Xiong, Stephen Merity, and Richard Socher. 2016. Dynamic memory networks for visual and textual question answering. In *International conference on machine learning*. 2397–2406.

[60] Jun Yin, Xin Jiang, Zhengdong Lu, Lifeng Shang, Hang Li, and Xiaoming Li. 2016. Neural generative question answering. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*. AAAI Press, 2972–2978.

[61] Tom Young, Devamanyu Hazarika, Soujanya Poria, and Erik Cambria. 2018. Recent trends in deep learning based natural language processing. *ieee Computational intelligenCe magazine* 13, 3 (2018), 55–75.

[62] Helen Jiahe Zhao and Jiamou Liu. 2018. Finding Answers from the Word of God: Domain Adaptation for Neural Networks in Biblical Question Answering. In *2018 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–8.