NLP4SE: Evaluation of Natural Language Processing Techniques for Software Engineering Tasks

Zhen Zhang, Qifan Lu
University of Washington

1 Introduction

Natural language processing (NLP) techniques have been applied more and more in the field of Software Engineering (SE) today (Ernst, 2017), due to the following needs:

1. **For professional programmers**: read and build on on top of an increasingly huge codebase, which is written in a mixture of natural language (NL) and programming language (PL), as in application code, open source libraries, documentation, issue tracker etc.

2. **For non-professional programmers**: write relatively simple and small code snippets with a more friendly/natural interface

Motivated by these needs, there are many interesting tasks being defined in the intersection of NLP and SE. We categorize them as follows (the UPPER-CASED names is their corresponding acronyms for later references):

- **NL2PL**: Program/query synthesis from natural language specification or examples (GEN)

- **PL2NL**: Automatic documentation or commenting.

- {**PL+NL**}**2NL**: Code review, code summarization (SUM).

- {**PL+NL**}**2PL**: Code completion (COMPL), bug fix (BUGFIX), type inference.

- {**PL+NL**} **Comprehension**: Code search (SEARCH), sentimental analysis (SENTI), code/repo topic modelling (TOPIC), type inference (SEM), semantic/syntax defect detection (BUGLOC)

Since around 2015, the researchers of NLP and SE communities started work together closely, and produced a line of exploratory work in this area. There are two flavors of NLP techniques being applied to these problems: (1) **Training-light**, or *traditional NLP*: semantic parsing (SemPar) and language modelling (LM). (2) **Training-heavy**, or *deep learning NLP*: sequence/tree-shaped encoder and decoder based on recurrent neural network (RNN) architecture, attention mechanism etc. However, these two flavors are not totally separate. Instead, we think that the essential difference is in how much manual feature engineering is applied and how much data is available to train a huge yet high quality matrix of parameters.

However, this is a young field and naturally, we are wondering what is proven to be working and what is not, and what can we learn from that. Specifically, in this report, we ask the following two questions:

1. Is SE community using NLP techniques to its full potential?

2. What is the quality of existing artifacts? Are the results on paper easily reproducible? Can their system be easily extended? or more specifically,

    - **Reproducibility** Is it difficult to reproduce the evaluation results on paper, especially for the work based on neural networks (well-known for its lack of robustness) even with moderate computation power?
    - **Extensibility** What result can we get from customizing the model? What could we learn from improving the previous work?

In the end of the day, we hope to gain better insights about what kind of techniques/model/hyper-

parameter is better suited for what kind of SE task, and why it is so.

Contributions

- **Design key attributes** of a new NLP4SE taxonomy
- **Verify reproducibility** of a selected subset of existing work
- **Extend Jdoctor** by replacing hardcoded rules with a general paraphrasing-based model

2 Background

2.1 Language Modeling

A **statistical language model** is a probability distribution over characters, words or part of speeches. Among traditional statistical language models, N-gram is the simplest yet most classical one. Along with Hidden Markov Models it can be used to solve part-of-speech tagging problem, which enables applications like chunking and machine translation. Probabilistic Context-Free Grammars (PCFGs) further take the hierarchy of speech into account, and are more powerful of examining the rationality of sentences. There also exists more powerful language models such as log-linear modules and Conditional Random Fields (CRF).

In recent years, **neural language models**, many of which powered by CNN or LSTM, have been widely adopted by NLP community. CNN neural language models are often used for classification tasks (Kim, 2014), while LSTM-based sequence-to-sequence models (Sutskever et al., 2014) enable generative tasks like neural machine translation (Bahdanau et al., 2014). To combat the sparsity problem caused by one-hot encoding, word embedding techniques are utilized to reduce the dimension of word vectors and establish their relations in lower-dimensional space. Word2vec (Mikolov et al., 2013) is the first word embedding technique to see widespread use, which includes two models with similar structure, known as Continuous Bag-of-Words (CBOW) and skip-gram models. GloVe (Pennington et al., 2014) is another popular word embedding technique focusing on the co-occurrence of words. Finally, attention mechanism, originally proposed by Vaswani et al. (Vaswani et al., 2017), became a research hotspot in the last few years since it empowers the neural network to "focus" on particular context information from input or network memory.

2.2 Semantic Parsing

Semantic parsing is the task of converting natural language utterances to a machine-understandable logic form. It is commonly used in situations where formalized user intention is needed, such as question answering or code generation. Traditionally, semantic parsers are trained in a supervised way, using natural language utterances and their corresponding machine-understandable representation. However, because of the variety and ambiguity of natural language, traditional semantic parsers often fail to translate user input accurately. Various researches have tried to tackle this problem using advanced parsing techniques.

Artzi et al. propose inducing semantic parsers from unannotated conversational logs (Artzi and Zettlemoyer, 2011). They argue that users' response to machine's remediation requests can be used to avoid original confusion in user future conversations, and as such meanings can be modeled from latent variables without any explicit annotation. They conduct experiments on logs from the DARPA Communicator corpus (Walker et al., 2002), which contain 376 conversations between the system and the users. The proposed method is compared against supervised method and a no-conversation baseline by measuring the proportion of correctly generated logic forms. The authors find out that the accuracy of their method is significantly higher than no-conversation baseline and is only slightly worse then supervised method.

Berant et al. suggest that paraphrasing can be used to handle myriad expressions of equivalent knowledge base predicates (Berant and Liang, 2014). When given an input utterance, a small and deterministic set of candidate logical forms is generated, along with the canonical utterances for these logical forms. Then, a paraphrase model is used to choose the realization that best paraphrases user input, which in turn determines the best logic form. Since the paraphrasing model is decoupled from the parsing process, the authors present two paraphrase models, an association model and a vector space model, and train them jointly during their experiments. The datasets they use are WebQuestions (Berant et al., 2013), which contains 5,810 question-answer pairs with common questions asked by web users, and Free917

(Cai and Yates, 2013), which includes 917 questions manually authored by annotators. They compare their method with a number of previous works, as well as their own model with ablations and they are able to show various degrees of improvements over previous methods. They have open-sourced their semantic parser at https://nlp.stanford.edu/software/sempre/.

Semantic parsing can be applied to a variety of applications. Dhamdhere et al. show that semantic parsing could play an important role in **data exploration** (Dhamdhere et al., 2017). Data exploration process can guide the user to find and filter the data he/she wants, aggregate and visualize data in an understandable and straightforward way, and enable the user to share data for further analysis. Throughout the paper the authors mainly show the internals of their data exploration system. As the most complex part of the system, the semantic parser first annotates the query with entities and intent word types, then parse the query with a context-free grammar. The query is examined against semantic predicates which invalidates unreasonable queries, and finally each parse is given a score. The parses then goes through a series of steps within the answering engine in order to find the answer for the query and present the answer to the user.

Besides, semantic parsing is also useful for **code generation tasks**. One of the typical example is the Seq2SQL paper (Zhong et al., 2017), which formulates the problem of translating natural language questions to corresponding SQL queries. Seq2SQL also provides the WikiSQL dataset, one of the largest hand-annotated semantic parsing dataset available now, containing 80,654 pairs of natural language utterances and corresponding SQL queries from 24,241 tables extracted from Wikipedia. Numerous later works aim to improve the performance of question parsing on this dataset. Hwang et al. (Hwang et al., 2019) designed a novel semantic parser called SQLOVA which combines BERT-based word representations, sequence-to-SQL generation and execution-guided decoding all together. They compare their parser to the baseline and Seq2SQL method proposed in the original paper, as well as eight related works. They are able to show that their method outperform any other existing methods and achieves 80% to 90% accuracy on both logical form and during execution.

2.3 Recurrent Neural Networks

Recurrent Neural Network (RNN) is conceptually used as a variant length decoder-encoder in NLP "X2Y" flavor of tasks, e.g. SEQ2ONE (classification), ONE2SEQ (image captioning), SEQ2SEQ (translation, tagging, or predicting next word in LM task).

In the context of NLP, RNN is essentially a way to generate the next word using previous words, and the amount of previous words is not fixed. RNN can be used as a replacement for conditional distribution $P(\vec{X}|\vec{Y})$ for SEQ2SEQ tasks, e.g. tagging (in this case $\vec{X}$ is the tag sequence, and $\vec{Y}$ is the feature sequence. Features can either be manually designed (which is called Conditional Random Field (CRF) in the most general form), or reuse a pre-trained word embedding. RNN is expected to replace the manual feature design.

There are several variants of RNN used similarly/differently in SE tasks:

- RNNME-p is a faster variant of RNN with a hidden layer size of p that combines RNN-p with a class-based maximum entropy model (Mikolov et al., 2011). RNNME-40 (i.e. **40 hidden layers**) is used in SLANG tool (Raychev et al., 2014), as an alternative LM parallel with 3-gram model, for computing conditional probability of the next word, given previous words. Training on all data had a reported runtime of 5h using Core i7 GPU.

- Long-Short-Term-Memory (LSTM) is a concrete example of RNN. In Code-NN (Iyer et al., 2016), the LSTM with 400 hidden units is used in evaluation (word embedding size is also 400). Training could take several hours (Table 2). In many other works (Gu et al., 2018) (200 hidden units, 100 embedding size, training cost: 50 hours on Tesla K40) (Tufano et al., 2018) (256 hidden units, 512 embedding size, training for 50K epochs, 1 layer encoder + 2 layer decoder) (Iyer et al., 2018) (512 hidden units per direction, 2 layers, 512 embedding size, 30 epochs of 20 mini-batch size), the bidirectional version of it is used for capturing different direction of dependencies.

2.4 Attention mechanism

Attention mechanism is essentially adding another layer of weights whenever you summarize something into something simpler. For example, when

used with a SEQ2SEQ RNN model, attention can summarize the sequence into a single vector. Then you get a SEQ2ONE model.

In NLP, attention mechanism is used to alleviate the problem of hidden state bottleneck [Cheng et al., 2016] by retrieve and make use of relevant previous hidden states.

Useful visualization techniques include heatmaps of attention weight.

In (Li et al., 2017) and (Vasic et al., 2019), an advanced neural network based on attention mechanism, called "Pointer Networks", is applied. Pointer network is a variation of SEQ2SEQ model with attention: instead of weighting the input elements, it points at them probabilistically. In effect, you get a permutation of inputs.

3 Taxonomy

First, we summarize the *key attributes* of the work in this domain (define this domain) (Table 1), and use these to guide our further exploration.

- **Time** of publication
- **NLP Model Techniques**, e.g. LSTM
- Downstream **Task** Category
- **Dataset** Scale and Source
- Generality: What source programming languages are the datasets composed of? (**PLs**)
- Domain-specific engineering (**DSE**) in data pre-processing, feature design and extraction etc.

3.1 Downstream Task Categories

Code search is the task of finding relevant code snippet given some informal description. It is like StackOverflow or GitHub search. In neural code search paper (Sachdev et al., 2018), the authors find that "while a basic word–embedding based search procedure works acceptably, better results can be obtained by adding a layer of supervision, as well as by a customized ranking strategy". They used "a combination of word embedding (Sachdev et al., 2018), TF–IDF weighting, and efficient higher–dimensional vector similarity search". Their evaluation dataset is "a benchmark of 100 Android–specific queries from Stack Overflow". Their model is trained on "the corpus of 1,000 Android projects from GitHub". The model takes SO question as input, and output relevant code in Github corpus. The ground-truth is the code snippet in the original SO answer. The supervision layer learns a translation from query words to code words, since there might be some synonyms. The ranking strategy is basically resorting the results according to some manual rules. The manual eval result is 68/100, the auto eval result is 176/518.

Specification generation is the task of generating formal specification of code semantics from informal documentation about the code. Jdoctor (Blasi et al., 2018) combines "pattern, lexical, and semantic matching to translate Javadoc comments into executable procedure specifications written as Java expressions". In their empirical evaluation, Jdoctor achieved "precision of 92% and recall of 83% in translating Javadoc into procedure specifications". In details, Jdoctor uses a natural language parser to find propositions. Extracting $\langle$ subject,predicate $\rangle$ pairs from natural language sentences is referred to as Open Information Extraction (OIE). Jdoctor completes the POS tagging process by means of the Stanford Parser, which produces a semantic graph (an enriched parse tree) representing the input sentence. The semantic similarity matching is based on GloVe word embeddings. To compare multiple words at once, Jdoctor uses the Word Mover's Distance (WMD) algorithm. The dataset is a collection of Java programs, in which 563 methods are analyzed. The ground-truth, i.e. the executable specification, is manually constructed. For example, the amount of normal pre-conditions is 243.

Code completion is the task of automatically synthesizing code completions for programs being written. The SLANG tool (Raychev et al., 2014) extracts sequences of method calls from a large codebase, and index these into a statistical language model. The model is used to predict the rest of the API sequence. The models include N-gram and Recurrent Neural Networks. Their best model is a combined language model between a 3-gram (Witten-Bell smoothing) and a RNNME-40 language model. The training data is 3,090,194 Android methods. The evaluation data are manually selected, compromising of 20 tasks, 14 code snippets, and 50 methods. SLANG returns the correct completion as a first result in 58 out of 84 test cases.

Patch Synthesis: NMT paper (Tufano et al.,

Name	Task(3.1)	PLs	NLP Model	DSE(3.2)	Dataset(3.3)
SLANG 2014(Raychev et al., 2014)	COMPL	Android	ngram, RNNME-p	P, SA	3M (GH)
CodeNN 2016(Iyer et al., 2016)	SUM	SQL, C#	LSTM, Att	P, TC, ID, TY	30K+60K (SO)
SQLNet 2017(Xu et al., 2017)	GEN	SQL	LSTM, Att	SK	80K (WK)
DeepCS 2018(Gu et al., 2018)	SEARCH	Java	biLSTM, MLP	P, TC	18M (GH)
Jdoctor 2018(Blasi et al., 2018)	GEN	Java	SemPar	P, IR	1K (OSS)
Sachdev 2018(Sachdev et al., 2018)	SEARCH	Android	TF-IDF	P, TC	700K (GH)
Tufano 2018(Tufano et al., 2018)	BUGFIX	Java	biLSTM, Att	P, TC, ID, DIFF, IDIOM	58K (GH)
CONCODE 2018(Iyer et al., 2018)	GEN	Java	biLSTM, Att	P, TC, ID	100K (GH)
Dong 2016(Dong and Lapata, 2016)	GEN	Prolog, λ, IFTTT	LSTM, Att	None	500+1K+ 5K+77K (PUB)
Lin 2018(Lin et al., 2018)	SENTI	*	RNN	None	1.5K (SO)+ 7K(PUB)
Li 2017(Li et al., 2017)	COMPL	JavaScript, Python	RNN, Att, PointerNet	AST	150K$\times$2 (PUB)
Vasic 2018(Vasic et al., 2019)	BUGLOC, BUGFIX	C#, Python	RNN, Att, mtPointerNet	None	150K+10K(PUB)
Gelman 2018(Gelman et al., 2018)	SEARCH, SUM	*	PCA, CNN	None	4M (SO)
Allamanis 2013 (Allamanis and Sutton, 2013)	TOPIC	*	LDA	P	300K (SO)
Santos 2018 (Santos et al., 2018)	BUGLOC	Java	ngram, LSTM	P, ID	2M (GH)
DeepTyper 2018 (Hellendoorn et al., 2018)	SEM	TypeScript	GRU	P	50K (GH)

Table 1: Key attributes summary for a selected subset of work. The negative sign '-' means that author gets negative results from their study

2018) uses Neural Machine Translation techniques (RNN Encoder- Decoder architecture) for learning bug-fixing patches for real defects. The whole dataset, called BFPsmall, has 58,350 bug-fixes. The accuracy of predicting the right patch is 538 out of 5,835 cases.

Summarization: Summarization is the task of summarizing programs using short natural language utterance. It can help people understand large, unfamiliar, poorly-documented software in a shorter time. For example, Code-NN (Iyer et al., 2016) tries to use neural networks for this task (See section 4).

3.2 Domain-specific Engineering

- **P**: parsing token sequence into abstract syntax tree (AST)
- **TC**: AST cleaning, i.e. removing terminals that match specific rules
- **ID**: identifier anonymization/renaming, e.g. turn `int listElement` into `int var1`. Semantic scoping rules might be used to ensure a consistent renaming
- **TY**: type annotation, i.e. annotating extra type information on AST by type inference
- **SK**: syntax-guided sketching, i.e. leaving

slots to be filled in a pre-determined syntactical skeleton

- **IR**: intermediate representation and its translation
- **DIFF**: create AST difference using tools like GumTree Spoon AST Diff
- **IDIOM**: flag a manually collected important identifiers that won't be removed from corpus in **ID** step
- **SA**: static analysis, e.g. used to transform program corpus into an API invocation sequence
- **AST**: standardized format of AST consisting of value/type pairs

3.3 Dataset

- **SO**: StackOverflow
- **GH**: GitHub
- **WK**: Wikipedia
- **OSS**: Open Source Software
- **PUB**: dataset from previous publications

4 Case Study: Code-NN

4.1 Introduction

Code-NN (Iyer et al., 2016) is a neural network model that can generate natural language summary of C# code snippets and SQL queries. Generating such a summary is often challenging because the text can include complex, non-local aspects of the code. Code-NN uses LSTM networks (for surface realization) and attention mechanism (for content selection) to produce the output sentences. The results are reproduced successfully using the original artifact.

4.2 Dataset

Data collection from StackOverflow

- C#: 934,464 posts
- SQL: 977,623 posts

C#	**# Statements**		**# Functions**	
	≥ 3	23,611 (44.7%)	≥ 3	26,541 (51.0%)
	≥ 4	17,822 (33.7%)	≥ 4	20,221 (38.2%)
	# Loops		**# Conditionals**	
	≥ 1	10,676 (20.0%)	≥ 1	11,819 (22.3%)
SQL	**# Subqueries**		**# Tables**	
	≥ 1	11,418 (35%)	≥ 3	14,695 (44%)
	≥ 2	3,625 (11%)	≥ 4	10,377 (31%)
	# Columns		**# Functions**	
	≥ 5	12,366 (37%)	≥ 3	6,290 (19%)
	≥ 6	9,050 (27%)	≥ 4	3,973 (12%)

Table 1: Statistics for code snippets in our dataset.

C#	**Avg. code length**	38 tokens	**# tokens**	91,156
	Avg. title length	12 words	**# words**	24,857
SQL	**Avg. query length**	46 tokens	**# tokens**	1,287
	Avg. title length	9 words	**# words**	10,086

Table 2: Average code and title lengths together with vocabulary sizes for C# and SQL after post-processing.

Figure 1: Dataset Statistics

Data Cleaning

- QA pairs with non-code answer. resulting in a total of 145,841 pairs for C# and 41,340 pairs for SQL.
- Cleaning meaningless titles For the final dataset, authors retained 66,015 C# (title, query) pairs and 32,337 SQL pairs.

Transforming Snippet to AST and AST Generalization

- best-effort parse
- strip out all comments
- replace literals with tokens denoting their types.
- for SQL, replace table and column names with numbered placeholder tokens.

Dataset Splitting 80% training, 10% validation and 10% testing.

Testset Extension Ask human annotators to provide two additional titles. See Figure 1 for final dataset.

4.3 Model

4.4 Experimental Setup

To reproduce this work, I forked the repo from the original author's artifact on GitHub. My fork is the

one for conducting the following reproducibility study.

According to the instructions in `README.md`, I produced the results by adding an extra configurable `decay` environmental variable and use the `run.sh` provided by the original author.

The training and inference is on GeForce GTX 1080 Ti (12 GB GRAM).

4.5 Evaluation Results

My reproduction produced the following results: Table 2 (its rendering: Figure 3b and Figure 3a). Compared to the original results (Figure 2), BLEU-4 score (sql, csharp) has a **the difference less than 7% of the reported score**.

	Model	METEOR	BLEU-4
C#	IR	7.9 (6.1)	13.7 (12.6)
	MOSES	9.1 (9.7)	11.6 (11.5)
	SUM-NN	10.6 (10.3)	19.3 (18.2)
	CODE-NN	**12.3 (13.4)**	**20.5 (20.4)**
SQL	IR	6.3 (8.0)	13.5 (13.0)
	MOSES	8.3 (9.7)	15.4 (15.9)
	SUM-NN	6.4 (8.7)	13.3 (14.2)
	CODE-NN	**10.9 (14.0)**	**18.4 (17.0)**

Table 3: Performance on EVAL for the GEN task. Performance on DEV is indicated in parentheses.

	Model	Naturalness	Informativeness
C#	IR	3.42	2.25
	MOSES	1.41	2.42
	SUM-NN	**4.61***	1.99
	CODE-NN	**4.48**	**2.83**
SQL	IR	3.21	2.58
	MOSES	2.80	2.54
	SUM-NN	4.44	2.75
	CODE-NN	**4.54**	**3.12**

Table 4: Naturalness and Informativeness measures of model outputs. Stat. sig. between CODE-NN and others is computed with a 2-tailed Student's t-test; $p < 0.05$ except for *.

Figure 2: Original Evaluation Results

4.6 Discussion

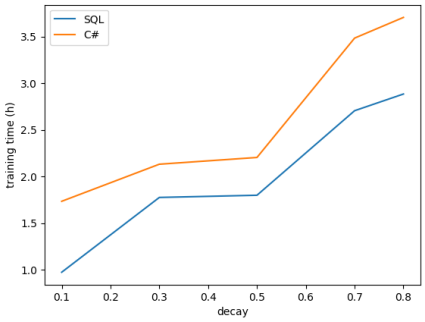
1. BLEU-4 score is reproducible

2. Choosing different learning rate decays has significant effect on training time, but not on the performance score

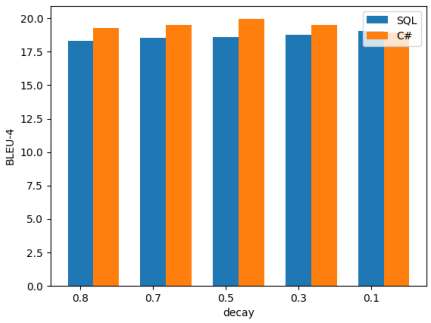
5 Case Study: SQLNet

5.1 Introduction

This work aims at generating better SQL queries from NL description, which is formalized in the



(a) Reproduction Decay - Runtime Analysis



(b) Reproduction Decay - BLEU-4 Analysis

WikiSQL dataset. There are many prior work, but the performance on that dataset is far from satisfying. This work combines sketch-based synthesis technique to avoid the ordering problem in SEQ2SEQ model. Its own neural network, called SQLNet, uses column-attention and SEQ2SET to fill in these slots. The result outperformed the prior art by 9% to 13%, and is reproduced successfully by us using the original artifact.

5.2 Dataset

WikiSQL (Zhong et al., 2017) is a dataset of 80654 hand-annotated examples of questions and SQL queries distributed across 24241 tables from Wikipedia.

Figure 2: An example in WikiSQL. The inputs consist of a table and a question. The outputs consist of a ground truth SQL query and the corresponding result from execution.

5.3 Model

Compared to the SEQ2SEQ model, which has a uniform but limited dependency relationships, the sketch-based model essentially encoded de-

Language	BLEU-4	Time spent (seconds)	Decay
sql	18.3023	10384.9	0.8
csharp	19.2872	13339.2	0.8
sql	18.5313	9741.38	0.7
csharp	19.5126	12541.4	0.7
sql	18.586	6480.81	0.5
csharp	19.9454	7938.36	0.5
sql	18.7667	6396.58	0.3
csharp	19.5126	7678.27	0.3
sql	19.0755	3512.49	0.1
csharp	18.9647	6247.93	0.1

Table 2: CodeNN Reproduction results

Model	Acc_{lf} (dev/test)	Acc_{ex} (dev/test)	Uses execution
X-SQL (He et al., 2019)	86.2 / 86.0	92.3 / 91.8	Inference
SQLNet (Xu et al., 2017)	-	69.8 / 68.0	
Seq2SQL (Zhong et al., 2017)	49.5 / 48.3	60.8 / 59.4	Training
Baseline(Zhong et al., 2017)	23.3 / 23.4	37.0 / 35.9	

Table 3: Part of the WikiSQL leaderboard showing the SOTA as of Mar 21 2019, the reproduced work, and the original paper. Source: https://github.com/salesforce/WikiSQL

pendencies based on the *language syntax tree* in model (Figure 4).

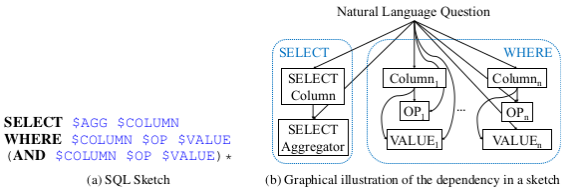


Figure 2: Sketch syntax and the dependency in a sketch

Figure 4: SQLNet sketch-based dependencies

For each slot, a SEQ2SET prediction model with column attention is used. The SEQ2SET is that, instead of generating a sequence of column names, we can simply predict which column names appear in this subset of interest as a distribution of column name *col* conditioned on the question Q: $P_{wherecol}(col|Q)$. The column-attention captures that fact that different word in the question has different degree of relevance to some column. For example, the token "player" is more relevant to predicting the `player` column in the `SELECT` clause.

5.4 Experimental Setup

To reproduce this work, I forked the repo from the original author's artifact on GitHub. My fork is the one for conducting the following reproducibility study.

According to the instructions in `README.md`, I produced the results using the following three configurations.

- baseline: `python2 train.py --baseline --datas`
- CA: `python2 train.py --ca`
- TE,CA: `python2 train.py --ca --train_emb`

Training and inference is run on a GeForce GTX 1080 Ti (12 GB GRAM).

5.5 Evaluation Results

My reproduction produced the following results: Table 4 (its rendering: Figure 5). Compared to the original results (Figure 6), for each type of test accuracy ($Acc_{qm,ex,agg,sel,where}$), **the difference is less than 1%**.

Step	Runtime (s)	Mode	Acc_{qm}	Acc_{ex}	Acc_{agg}	Acc_{sel}	Acc_{where}
train	44818	CA	N/A	N/A	N/A	N/A	N/A
test	221.589	CA	0.597 (0.609)	0.667 (0.677)	0.901 (0.898)	0.897 (0.907)	0.706 (0.720)
train	44282.4	TE,CA	N/A	N/A	N/A	N/A	N/A
test	219.307	TE,CA	0.608 (0.622)	0.675 (0.688)	0.901 (0.898)	0.900 (0.912)	0.719 (0.734)
train	24865.2	baseline	N/A	N/A	N/A	N/A	N/A
test	307.245	baseline	0.531 (0.536)	0.602 (0.610)	0.921 (0.924)	0.897 (0.903)	0.616 (0.618)

Table 4: Reproduction result of SQLNet model. Accuracy format: test (dev)

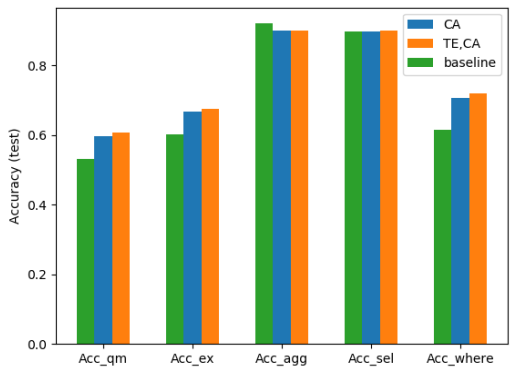


Figure 5: SQLNet accuracy (test) reproduction

	dev			test		
	Acc_{lf}	Acc_{qm}	Acc_{ex}	Acc_{lf}	Acc_{qm}	Acc_{ex}
Seq2SQL (Zhong et al. (2017))	49.5%	-	60.8%	48.3%	-	59.4%
Seq2SQL (ours)	52.5%	53.5%	62.1%	50.8%	51.6%	60.4%
SQLNet	-	**63.2%**	**69.8%**	-	**61.3%**	**68.0%**

Table 1: Overall result on the WikiSQL task. Acc_{lf}, Acc_{qm}, and Acc_{ex} indicate the logical form, query-match and the execution accuracy respectively.

	dev			test		
	Acc_{agg}	Acc_{sel}	Acc_{where}	Acc_{agg}	Acc_{sel}	Acc_{where}
Seq2SQL (ours)	90.0%	89.6%	62.1%	90.1%	88.9%	60.2%
Seq2SQL (ours, C-order)	-	-	63.3%	-	-	61.2%
SQLNet (Seq2set)	-	-	69.1%	-	-	67.1%
SQLNet (Seq2set+CA)	**90.1%**	91.1%	72.1%	**90.3%**	90.4%	70.0%
SQLNet (Seq2set+CA+WE)	**90.1%**	**91.5%**	**74.1%**	**90.3%**	**90.9%**	**71.9%**

Table 2: Break down result on the WikiSQL dataset. Seq2SQL (C-order) indicates that after Seq2SQL generates the `WHERE` clause, we convert both the prediction and the ground truth into a canonical order when being compared. Seq2set indicates that the sequence-to-set technique is employed. +CA indicates that column attention is used. +WE indicates that the word embedding is allowed to be trained. Acc_{agg} and Acc_{sel} indicate the accuracy on the aggregator and column prediction accuracy on the `SELECT` clause, and Acc_{where} indicates the accuracy to generate the `WHERE` clause.

Figure 6: SQLNet original results

5.6 Discussion

1. Accuracy score is reproducible

2. Column-attention with embedding training is the best configuration, but not significantly better than others.

6 Case Study: Extending Jdoctor with Paraphrasing-based Semantic Parsing for Testcase Generation

6.1 Background

- *JDoctor* JDoctor (Blasi et al., 2018) generates program specifications from semi-structured JavaDoc comments. Program specifications express intended program behavior, and for Java member methods program specifications can be expressed by a boolean Java expression. They are hardly directly available in practice, but can be obtained from semi-structured JavaDoc comments written in natural language.

  JavaDoc comments for each method usually contains a series of directives, such as `@param` (for parameters), `@returns` (for return values) and `@throws` (for exceptions). Some of these directives contain descriptions about restrictions for parameters, return values or exceptions. These descriptions are called guard descriptions, from which JDoctor generates guard conditions which are boolean Java expressions semantically equivalent to guard descriptions (See Figure 7). JDoctor utilize several hand-crafted rules and feature extraction processes to generate guard conditions from guard descriptions. The authors of JDoctor extracted guard conditions from several libraries and generated corresponding guard descriptions, which will be used as the baseline to evaluate our model.

- *Semantic Parsing via Paraphrasing* Semantic Parsing via Paraphrasing (Berant and Liang, 2014) deals with semantic parsing problems in question answering tasks. One common problem of question answering is that the

same question can be expressed in a myriad of equivalent ways. The authors of this paper trains a paraphrasing model that converts question utterance into its canonical form, and then parse it into logical expressions so that the question can be executed.

```
/**
 * @param funnel the funnel of T's that the constructed {@code
 *     BloomFilter<T>} will use
 * @param expectedInsertions the number of expected insertions to the
 *     constructed {@code BloomFilter<T>}; must be positive
 * @param fpp the desired false positive probability (must be positive
 *     and less than 1.0)
 */
public static <T> BloomFilter<T> create(
  Funnel<? super T> funnel,
  int expectedInsertions,
  double fpp) { ... }
```

expectedInsertions > 0
fpp > 0 && fpp < 1.0

Figure 7: An example of guard samples. "Must be positive" and "must be positive and less than 1.0" are guard descriptions extracted from JavaDoc directives, and guard conditions `expectedInsertions > 0` and `fpp > 0 && fpp < 1.0` are extracted from guard desciptions by JDoctor.

6.2 Tasks

The tasks of this case study is to train a paraphrasing-based model (hereinafter referred to as paraphrasing model) that first paraphrases guard description into its canonical form and then parse it into Java expressions to generate corresponding conditions. To evaluate the relative performance of the model compared to JDoctor, we compare the guard conditions generated by the paraphrasing model with original guard conditions from the JDoctor datasets.

6.3 Model

To train or apply our model, we will need a dataset with guard samples. A guard sample contains a description and a corresponding condition (expressed in the form of Java expressions), which states the restrictions on method parameters, return values or exceptions. A `@param`, `@returns` or `@throws` directive may or may not contain a guard sample. Throughout the training or application process, we will make use of identifier tables, which are bidirectional mappings between values (such as literal values or identifier names) with reference tokens (e.g. `#I1`, `#I2` etc).

When training the paraphrasing model, we perform the following steps:

Description Transformation The description of each guard sample undergoes several kinds of transformations:

1. *Tokenization* Tokenization of the description is done with NLTK.

2. *Operator Transform* Operators in the description are replaced by their canonical descriptions (see below for details). For example, occurance of operator '¡' will be replaced by its textual description ["less", "than"].

3. *Simple Identifier Matching* Each identical occurance of identifier will be replaced by a corresponding identifier reference token.

4. *Number Transform* Each occurance of numerical constants (Integers and floating point numbers) will be replaced by a corresponding number reference token.

Hint Sequence Generation For each guard description, we generate a few hint sequences of the same length, which contain extra context information for the description. For words that do not generate a particular kind of hint, the correponding hint token is a pad token.

1. *Number Hints* To generate number hints sequence, we first scan for occurance of consecutive number words. Then we parse these number words into numbers and replace the numbers with corresponding number reference tokens.

2. *Identifier Hints* To generate identifier hints sequence, we first gather all identifier names related to current method, then split these names into a group of words by camel case or snake case convention. For each

group of words we then look for their vector representations. In other words, we have built a sequence of vectors for each identifier. Then, for each N-gram in the description, we also built a sequence of vectors for each N-gram, and then pick the identifier whose sequence of words has the smallest Word Mover's Distance to the current N-gram. The identifier is then converted to corresponding identifier reference and added to the identifier hints sequence.

Canonical Description Generation The condition of each guard sample is first parsed into an Abstract Syntax Tree (AST), which is then used to generate canonical guard descriptions. The canonical description for operators, field access, array access and method invocations are assembled from the canonical descriptions of their operands. For example, expression "[LHS] + [RHS]" will generate canonical guard description "(LHS Description) add (RHS Description)". Like the description of guard samples, all occurances of identifiers and number literals are replaced by corresponding reference tokens.

Train Paraphrasing Model We train a seq2seq model to generate canonical guard description from original description and its hint sequences. We use the original description and hint sequences generated in previous steps as source data and the canonical description as target data. The seq2seq model treats original description and hint sequences as multiple input channels. Each channel first goes through embedding look-up and a bidirectional encoder GRU network, which produces multiple output sequences. These channel outputs are then token-wise concatenated and conbined into one channel and further applied to attention mechanism and a bidirectional decoder GRU network, which will finally generate target sequences.

When predicting guard conditions from guard descriptions, we perform the following steps:

Description Transformation and **Hint Sequence Generation** Same as the training process.

Apply Paraphrasing Model The transformed description as well as its hint sequences are applied to the Seq2seq paraphrasing model, which will generate an equivalent canonical guard description. All reference tokens in the output sequence will then be dereferenced back into numbers and identifier names.

Canonical Description Parsing The canonical guard description is then parsed into guard conditions (Java expressions) using a AntLR4-based parser.

The training and application process is shown in Figure 8. Green arrow denotes training steps, blue arrow denotes application steps, while black arrow denotes steps applied in both training and application.

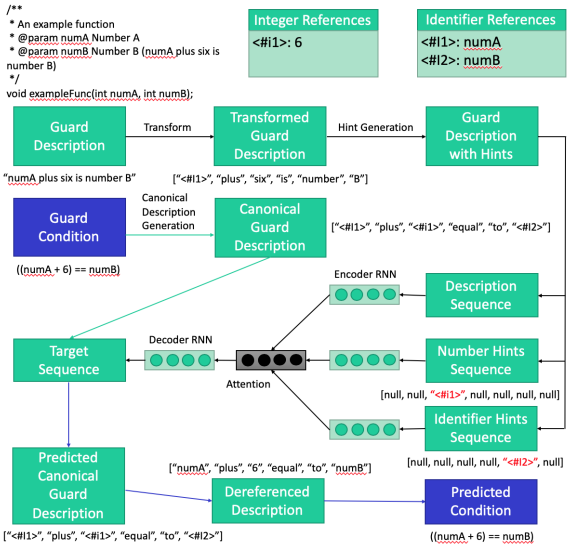


Figure 8: Diagram for the paraphrasing model

6.4 Experiments

We trained the paraphrasing-based semantic parsing model using the guard samples generated by JDoctor. During training, we chose a dataset as the source dataset and split it into training, validation and testing set (3:1:1). We used the training set to train the paraphrasing model and used validation set to monitor its performance over time. Then we test the performance of the model on the test portion of the source dataset. After training process, we apply our model to another dataset, called transfer set, to see if our model can adapt to guard descriptions with different writing styles.

We used Java standard library, Apache commons collections and Google Guava as the source dataset and Apache commons math as the target dataset. Table Table 5 shows the result of compar-

ing predicted guard conditions to original guard conditions. For each guard sample, GE (Generation Error) indicates that the canonical description parsing step in the applying process fails, DNC (Does Not Compile) indicates that generated condition does not compile due to type mismatch or other reasons, DO (Different Outputs) indicates that our model produce conditions different from original conditions, and SO (Same Outputs) indicates that predictions are identical to original conditions.

The experiment result shows that like many other deep learning models, trained models perform better when more training data is provided. Additionally, models trained on a particular source dataset do continue to work on another transfer set, but the accuracy of the model can drop significantly. This might be attributed to different writing style or more complex statement of guard descriptions in the transfer set.

6.5 Error Analysis

Correct Outputs Most of the simple binary operator expressions are predicted correctly. For example, guard description "s is positive" will be parsed as `s > 0`. Besides, description that involves few identifiers, expresses simple logical relationship and does not require type deduction also works. One of the example is guard description "The group is shutdown", which is correctly parsed as `(((group == null) == false) && group.isShutdown())`.

Failing Outputs There are mainly three kinds of failing outputs:

1. *Canonical Description Generation Error* Sometimes, the canonical description parsing step in the applying process can fail and the model produces no prediction. This might be attributed to using a deterministic semantic parser (generated from AntLR4) with too strict rules.

2. *Missing Part of the Condition* The condition predicted by our model misses constraints from the original description. For example, our model generates condition `(fractions == null)` for description "fractions array is null or cycleMethod is null", while the actual condition should be `fractions==null || cycleMethod==null`. The reason for this error may be that seq2seq network ignores parts of the input, or the canonical description parsing process fails halfway and thus only part of the condition is generated.

3. *Misuse Variable Names as Method Names* Because our paraphrasing model does not take variable and method type information into consideration, the model sometimes generates conditions that misuse variable names as method names and do not compile. For example, our model generates condition `(((columnData == null) == false) && columnData.length())` from description "columnData is empty", but the actual condition is `columnData.length == 0` since length here is a member variable instead of a method.

6.6 Conclusion

Our experiments indicate that seq2seq-based paraphrasing model does learn simple variations of JavaDoc statements, but fails to generate correct canonical JavaDoc (and condition) when more class context information (such as type information and visibility of variables and methods) is needed. We feel that more context information, such as variable and method types, should be provided to the paraphrasing model. Besides, either more preprocessing steps need to be applied to the guard description, or the neural network needs to be modified to be aware of the context information. We prefer the latter approach since it requires fewer manual labor and would like to make it a future research focus.

7 Discussion

7.1 Recipes of NLP4SE

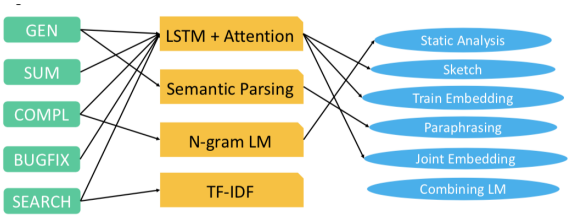


Figure 9: Summary of a NLP4SE Recipe

Based on our taxonomy and exploratory case studies, we produced a preliminary "recipe" for this line of research in Figure 9.

Model	GE	DNC	DO	SO	Total
Source Dataset					
java-std	44 (7%)	13 (2%)	89 (14%)	487 (77%)	633
apache-commons-collections	14 (5%)	6 (2%)	24 (9%)	232 (84%)	276
guava	6 (7%)	8 (9%)	12 (14%)	60 (70%)	86
Transfer Dataset (apache-commons-math)					
java-std	48 (4%)	51 (4%)	342 (29%)	751 (63%)	1162
apache-commons-colletions	72 (6%)	155 (13%)	524 (44%)	441 (37%)	1162
guava	224 (19%)	247 (21%)	327 (27%)	394 (33%)	1162

Table 5: Comparison of guard conditions generated by paraphrasing model with original guard conditions from JDoctor datasets.

Each line between the first column (task) and second column (model) represents a "applying" relationship. Each line between the second column (model) and third column (optimization) represents "with" relationship. "optimization" column represents the special techniques that are proven useful when applied to this intersection of research.

7.2 Future Directions

Except for asking and improving over the existing line in Figure 9, we can ask more questions:

1. This work's scope while being large, is not going to be complete. What are the other items in each column that are not covered by our study?

2. What about the missing links, even with the existing items in that diagram?

3. Some thicker lines mean more work on this direction. Why is it so? What is common in this sub-direction?

8 Conclusion

Our study showed that there exist a lot of prior exploration of applying NLP techniques, especially deep learning based ones, to SE tasks. However, many of them, while claiming to be language-agnostic, were not evaluated in a setting with more than two different languages. Compared to the overwhelming success of applications domain like machine translation of human languages, it still remains unclear whether NLP techniques can be applied in scale with proven benefits over existing tools.

However, despite we are yet to see the huge success, reproducibility and extensibility of existing artifacts are proven to be sufficient to support future work. Also, there are more and more standardized datasets in this domain, fostering more work with less overhead.

References

Miltiadis Allamanis and Charles Sutton. 2013. Why, when, and what: analyzing stack overflow questions by topic, type, and code. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 53–56. IEEE.

Yoav Artzi and Luke Zettlemoyer. 2011. Bootstrapping semantic parsers from conversations. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '11, pages 421–432, Stroudsburg, PA, USA. Association for Computational Linguistics.

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

Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. 2013. Semantic parsing on freebase from question-answer pairs. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1533–1544.

Jonathan Berant and Percy Liang. 2014. Semantic parsing via paraphrasing. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 1415–1425.

Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. 2018. Translating code comments to procedure specifications. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 242–253, New York, NY, USA. ACM.

Qingqing Cai and Alexander Yates. 2013. Large-scale semantic parsing via schema matching and lexicon extension. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 423–433.

Kedar Dhamdhere, Kevin S. McCurley, Ralfi Nahmias, Mukund Sundararajan, and Qiqi Yan. 2017. Analyza: Exploring data with conversation. In *Proceedings of the 22Nd International Conference on Intelligent User Interfaces*, IUI '17, pages 493–504, New York, NY, USA. ACM.

Li Dong and Mirella Lapata. 2016. Language to logical form with neural attention. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 33–43. Association for Computational Linguistics.

Michael D Ernst. 2017. Natural language is a programming language: Applying natural language processing to software development. In *2nd Summit on Advances in Programming Languages (SNAPL 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Ben Gelman, Bryan Hoyle, Jessica Moore, Joshua Saxe, and David Slater. 2018. A language-agnostic model for semantic source code labeling. In *Proceedings of the 1st International Workshop on Machine Learning and Software Engineering in Symbiosis*, MASES 2018, pages 36–44, New York, NY, USA. ACM.

Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 933–944, New York, NY, USA. ACM.

Pengcheng He, Yi Mao, Kaushik Chakrabarti, and Weizhu Chen. 2019. X-sql: reinforce context into schema representation. Technical report, Microsoft Dynamics 365 AI.

Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. 2018. Deep learning type inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 152–162, New York, NY, USA. ACM.

Wonseok Hwang, Jinyeong Yim, Seunghyun Park, and Minjoon Seo. 2019. Achieving 90% accuracy in wikisql.

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

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1643–1652. Association for Computational Linguistics.

Yoon Kim. 2014. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*.

Jian Li, Yue Wang, Michael R Lyu, and Irwin King. 2017. Code completion with neural attention and pointer networks. *arXiv preprint arXiv:1711.09573*.

Bin Lin, Fiorella Zampetti, Gabriele Bavota, Massimiliano Di Penta, Michele Lanza, and Rocco Oliveto. 2018. Sentiment analysis for software engineering: How far can we go? In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 94–104. IEEE.

Tomáš Mikolov, Anoop Deoras, Daniel Povey, Lukáš Burget, and Jan Černockỳ. 2011. Strategies for training large scale neural network language models. In *2011 IEEE Workshop on Automatic Speech Recognition & Understanding*, pages 196–201. IEEE.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119.

Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543.

Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 419–428, New York, NY, USA. ACM.

Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. 2018. Retrieval on source code: A neural code search. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2018, pages 31–41, New York, NY, USA. ACM.

Eddie Antonio Santos, Joshua Charles Campbell, Dhvani Patel, Abram Hindle, and José Nelson Amaral. 2018. Syntax and sensibility: Using language models to detect and correct syntax errors. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 311–322. IEEE.

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

Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 832–837, New York, NY, USA. ACM.

Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh singh. 2019. Neural program repair by jointly learning to localize and repair. In *International Conference on Learning Representations*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008.

Marilyn A. Walker, Alex Rudnicky, Rashmi Prasad, John Aberdeen, Elizabeth Owen Bratt, John Garofolo, Helen Hastie, Audrey Le, Bryan Pellom, Alex Potamianos, Rebecca Passonneau, Salim Roukos, Greg S, Stephanie Seneff, and Dave Stallard. 2002. Darpa communicator: Cross-system results for the 2001 evaluation. In *In ICSLP 2002*, pages 269–272.

Xiaojun Xu, Chang Liu, and Dawn Song. 2017. Sqlnet: Generating structured queries from natural language without reinforcement learning. *arXiv preprint arXiv:1711.04436*.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*.