Poster: Learning to Mine Parallel Natural Language/Source Code Corpora from Stack Overflow

Pengcheng Yin,* Bowen Deng,* Edgar Chen, Bogdan Vasilescu, Graham Neubig
Carnegie Mellon University, USA
{pcyin, bdeng1, edgarc, bogdanv, gneubig}@cs.cmu.edu

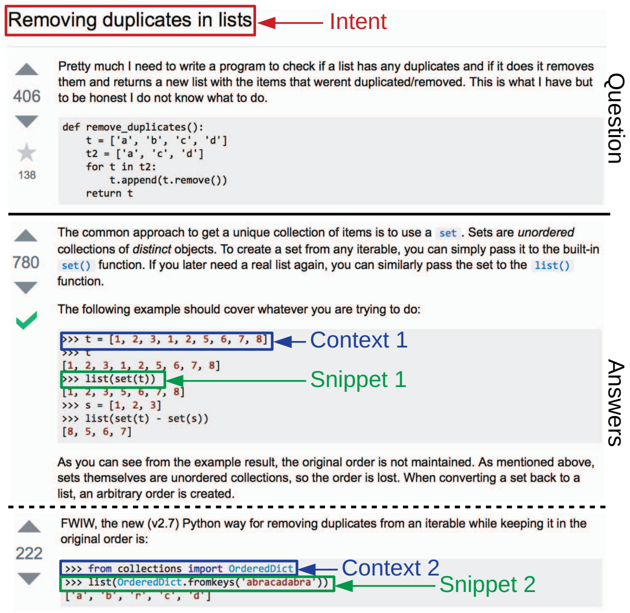
ABSTRACT

For tasks like code synthesis from natural language, code retrieval, and code summarization, data-driven models have shown great promise. However, creating these models requires parallel data between natural language (NL) and code with fine-grained alignments. Stack Overflow (SO) is a promising source to create such a data set but existing heuristic methods are limited both in their coverage and the correctness of the NL-code pairs obtained. In this paper, we propose a method to mine high-quality aligned data from SO by training a classifier using two sets of features: hand-crafted features considering the structure of the extracted snippets, and correspondence features obtained by training a neural network model to capture the correlation between NL and code. Experiments using Python and Java as test beds show that the proposed method greatly expands coverage and accuracy over existing mining methods, even when using only a small number of labeled examples.

1 INTRODUCTION

Recent years have witnessed a burgeoning new suite of developer assistance tools based on natural language processing (NLP) techniques, for code completion [4], source code summarization [1], automatic documentation of source code [10], code retrieval [2, 9] and even code synthesis from natural language [3, 6, 8, 11].

These applications, usually powered by statistical learning models, require training on parallel corpora of natural language (NL) and source code in high volume and high quality. While one can hope to mine such data from Big Code repositories like SO, straightforward mining approaches may also extract quite a bit of noise. We illustrate the challenges associated with mining aligned (parallel) pairs of NL and code from SO with the example of a Python question in Figure 1 demonstrates the main challenge of mining these pairs: it is not necessarily the case that the entirety of *every* code block accurately reflects the intent; some parts may simply describe the context, such as variable definitions (Context 1 in Figure 1) or import statements (Context 2), while other parts might be entirely irrelevant (*e.g.*, the latter part of the first code block in Figure 1). Given a NL intent and the goal of finding its matching source code snippets, prior work used either a straightforward approach that simply picks all code blocks that appear in the answers [2], or one



Removing duplicates in lists

Pretty much I need to write a program to check if a list has any duplicates and if it does it removes them and returns a new list with the items that werent duplicated/removed. This is what I have but to be honest I do not know what to do.

```
def remove_duplicates():
    t = ['a', 'b', 'c', 'd']
    t2 = ['a', 'c', 'd']
    for t in t2:
        t.append(t.remove())
    return t
```

The common approach to get a unique collection of items is to use a `set`. Sets are *unordered* collections of *distinct* objects. To create a set from any iterable, you can simply pass it to the built-in `set()` function. If you later need a real list again, you can similarly pass the set to the `list()` function.

The following example should cover whatever you are trying to do:

```
>>> t = [1, 2, 3, 1, 2, 5, 6, 7, 8]
>>> t
[1, 2, 3, 1, 2, 5, 6, 7, 8]
>>> list(set(t))
[1, 2, 3, 5, 6, 7, 8]
>>> s = [1, 2, 3]
>>> list(set(t) - set(s))
[8, 5, 6, 7]
```

As you can see from the example result, the original order is not maintained. As mentioned above, sets themselves are unordered collections, so the order is lost. When converting a set back to a list, an arbitrary order is created.

FWIW, the new (v2.7) Python way for removing duplicates from an iterable while keeping it in the original order is:

```
>>> from collections import OrderedDict
>>> list(OrderedDict.fromkeys('abracadabra'))
['a', 'b', 'r', 'c', 'd']
```

Figure 1: Excerpt from a SO post showing two answers, and the corresponding NL intent and code pairs.

that picks all code blocks from answers that are highly ranked or *accepted* [5, 10], which are clearly insufficient in these cases.

We propose a technique to extract aligned NL and code snippet pairs from Stack Overflow. Our key idea is to treat the problem as a classification problem: given an NL intent, *e.g.*, the question title for an SO post, and a set of candidate code fragments extracted from all answers of the post, we use a data-driven classifier to decide if a candidate code snippet aligns well with the NL intent. The classifier is trained on a small amount of labeled data, and is powered by both hand-crafted *structural* features, which capture the shape of valid code snippets, and neural network-based *correspondence* features, which measure the similarity between an NL intent and a code snippet. The structural features are designed to be language-agnostic and easy to implement, while the correspondence features are automatically learned on massive, noisy data without human intervention. Code for our mining algorithm and extracted corpora can be found at http://conala-corpus.github.io/.

2 MINING METHOD

Figure 2 illustrates our proposed mining method, which takes as input an SO post and outputs a ranked list of NL-code pairs. First, for every *how-to* SO question (*e.g.*,*"how to sort a list in descending order?"*), we consider its title as the questioner's intent, and extract

* PY and BD contributed equally to this work.

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-5663-3/18/05.
https://doi.org/10.1145/3183440.3195021

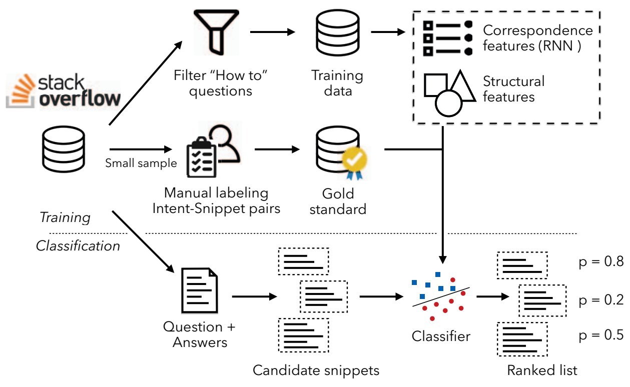


Figure 2: Overview of our approach.

all contiguous lines[1] from all code blocks in the question's answers as candidate snippets implementing this intent. Then, given each pair of intent and snippet, we use a logistic regression classifier to estimate the probability that the snippet actually implements the intent. The classifier uses two types of features:

Structural Features are language-independent features that are intended to distinguish whether we can reasonably expect that a candidate code snippet implements an intent. These features are designed to be both highly indicative and easy to implement. For example, FullBlock, StartOfBlock, EndOfBlock are a set of features that indicate if a candidate snippet is a full code block, at the start or end of the block, resp. We also design features that indicate the quality of the snippet, *e.g.*, the rank of its originating answer and whether the answer is marked as "accepted answer" by the asker. Our system employs a total of 13 structural features.

Correspondence Features are machine-learned features that directly measure the correspondence between an intent and a candidate snippet. We employ a state-of-the-art neural machine translation model [7] to measure the probability of the intent given the snippet and vice-versa. Intuitively, if this model assigns a high probability, the intent and snippet are likely to correspond to each-other. The neural model is trained on a massive, noisy corpus of code-NL pairs extracted using existing heuristic methods from SO posts in the target programming language.

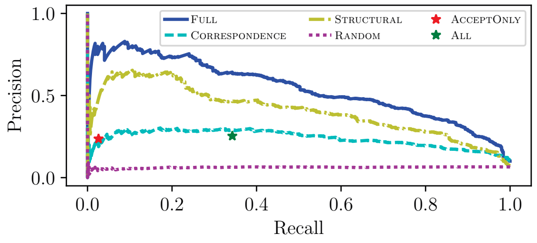
3 EXPERIMENTS

We conduct experiments on both Python and Java. For each language, we manually annotate a small number of intent-snippet pairs as the training/testing data of the classifier, and evaluate using five-fold cross validation. We compare our proposed mining approach (denoted as Full) with three baselines:

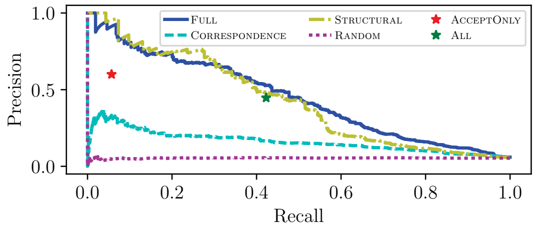
AcceptOnly selects the whole code snippet in the *accepted* answers containing exactly one code snippet [5, 10].
All selects all full code blocks in the top-3 answers in a post.
Random randomly selects from all possible candidate snippets.

Figure 3 depicts our main results. Our proposed method with the full feature set significantly outperforms all baselines: much better

[1] *e.g.*, in a 3-line block, we extract lines 1, 2, 3, 1-2, 2-3, and 1-3 as candidate code blocks



(a) Precision-Recall Curve on Python



(b) Precision-Recall Curve on Java

Figure 3: Evaluation Results on Mining Python and Java

recall (precision) at the same level of precision (recall) as the heuristic approaches. The increase in precision suggests the importance of intelligently selecting NL-code pairs using informative features, and the increase in recall suggests the importance of considering segments of code within code blocks, instead of simply selecting the full code block as in prior work. Comparing different types of features, we find that with structural features alone our model already significantly outperforms baseline approaches. Note that structural and correspondence features seem to be complementary, with the combination of the two feature sets further significantly improving performance, particularly on Python.

REFERENCES

[1] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A Convolutional Attention Network for Extreme Summarization of Source Code. *arXiv preprint arXiv:1602.03001* (2016).
[2] Miltiadis Allamanis, Daniel Tarlow, Andrew D Gordon, and Yi Wei. 2015. Bimodal Modelling of Source Code and Natural Language. In *ICML*.
[3] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Subhajit Roy, and others. 2016. Program Synthesis using Natural Language. In *ICSE*.
[4] Christine Franks, Zhaopeng Tu, Premkumar Devanbu, and Vincent Hellendoorn. 2015. CACHECA: A Cache Language Model based Code Suggestion Tool. In *ICSE*, Vol. 2.
[5] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing Source Code using a Neural Attention Model. In *ACL*.
[6] Nicholas Locascio, Karthik Narasimhan, Eduardo De Leon, Nate Kushman, and Regina Barzilay. 2016. Neural Generation of Regular Expressions from Natural Language with Minimal Domain Knowledge. In *EMNLP*.
[7] Graham Neubig. 2017. Neural Machine Translation and Sequence-to-Sequence Models: A Tutorial. *arXiv preprint arXiv:1703.01619* (2017).
[8] Chris Quirk, Raymond Mooney, and Michel Galley. 2015. Language to Code: Learning Semantic Parsers for If-This-Then-That Recipes. In *ACL*.
[9] Yi Wei, Nirupama Chandrasekaran, Sumit Gulwani, and Youssef Hamadi. 2015. *Building Bing Developer Assistant.* Technical Report. MSR-TR-2015-36, Microsoft Research.
[10] Edmund Wong, Jinqiu Yang, and Lin Tan. AutoComment: Mining question and answer sites for automatic comment generation. In *ASE*.
[11] Pengcheng Yin and Graham Neubig. 2017. A Syntactic Neural Model for General-Purpose Code Generation. In *ACL*.