Learning to Mine Aligned Code and Natural Language Pairs 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 require 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: the questions are diverse and most of them have corresponding answers with high quality code snippets. However, existing heuristic methods (*e.g.*, pairing the title of a post with the code in the accepted answer) are limited both in their coverage and the correctness of the NL-code pairs obtained. In this paper, we propose a novel method to mine high-quality aligned data from SO using two sets of features: hand-crafted features considering the structure of the extracted snippets, and correspondence features obtained by training a probabilistic model to capture the correlation between NL and code using neural networks. These features are fed into a classifier that determines the quality of mined NL-code pairs. 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. Further, we find that reasonable results are achieved even when training the classifier on one language and testing on another, showing promise for scaling NL-code mining to a wide variety of programming languages beyond those for which we are able to annotate data.

1 INTRODUCTION

Recent years have witnessed the burgeoning of a new suite of developer assistance tools based on natural language processing (NLP) techniques, for code completion [9], source code summarization [2], automatic documentation of source code [44], deobfuscation [16, 34, 40], cross-language porting [27, 28], code retrieval [3, 42] and even code synthesis from natural language [7, 21, 32, 47].

Besides the creativity and diligence of the researchers involved, these recent success stories can be attributed to two properties of software source code. *First*, it is highly repetitive [8, 10], therefore predictable in a statistical sense. This statistical predictability enabled researchers to expand from models of source code and natural language (NL) created using hand-crafted rules, which have a long history [23], to data-driven models that have proven flexible, relatively easy-to-create, and often more effective than corresponding hand-crafted precursors [13, 27]. *Second*, source code is available in large amounts, thanks to the proliferation of open source software in general, and the popularity of open access, "Big Code" repositories like GitHub and Stack Overflow (SO); these platforms host tens of millions of code repositories and programming-related questions and answers, respectively, and are ripe with data that can, and is, being used to train such models [34].

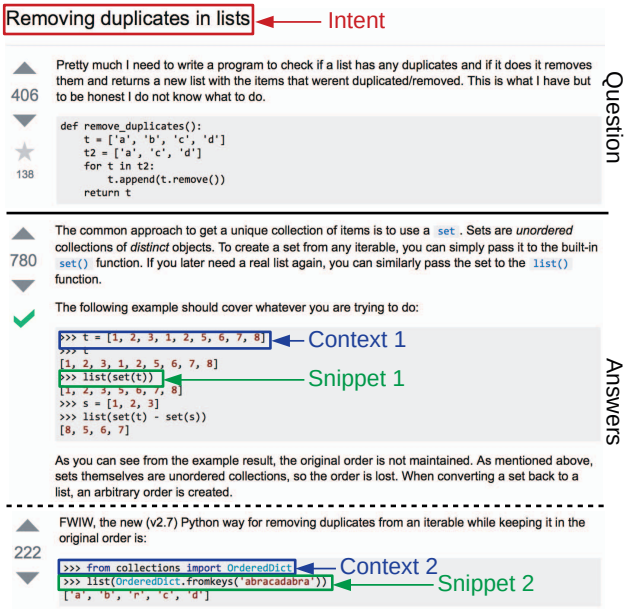


Figure 1: Excerpt from a SO post showing two answers, and the corresponding NL intent and code pairs.

However, the statistical models that power many such applications are only as useful as the data they are trained on, *i.e.*, garbage in, garbage out [35]. For a particular class of applications, such as source code retrieval given a NL query [42], source code summarization in NL [15], and source code synthesis from NL [33, 47], all of which involve correspondence between NL utterances and code, it is essential to have access to *high volume, high quality, parallel data*, in which NL and source code align closely to each other.

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

* PY and BD contributed equally to this work.

MSR '18, May 28–29, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5716-6/18/05…$15.00
https://doi.org/10.1145/3196398.3196408

mining aligned (parallel) pairs of NL and code from SO with the example of a Python question in Figure 1. Given a NL query (or intent), *e.g.*, "removing duplicates in lists", and the goal of finding its matching source code snippets among the different answers, prior work used either a straightforward mining approach that simply picks all code blocks that appear in the answers [3], or one that picks all code blocks from answers that are highly ranked or *accepted* [15, 44].[1] However, it is not necessarily the case that *every* code block accurately reflects the intent. Nor is it that the *entire* code block is answering the question; some parts may simply describe the context, such as variable definitions (Context 1) or import statements (Context 2), while other parts might be entirely irrelevant (*e.g.*, the latter part of the first code block).

There is an inherent trade-off here between scale and data quality. On the one hand, when mining pairs of NL and code from SO, one could devise filters using features of the SO questions, answers, and the specific programming language (*e.g.*, only consider accepted answers with a single code block or with high vote counts, or filtering out `print` statements in Python, much like one thrust of prior work [15, 44]); fine-tuning heuristics may achieve high pair quality, but this inherently reduces the size of the mined data set and it may also be very language-specific. On the other hand, extracting all available code blocks, much like the other thrust of prior work [3], scales better but adds noise (and still cannot handle cases where the "best" code snippets are smaller than a full code block). Ideally, a mining approach to extract parallel pairs would handle these tricky cases and would operate at scale, extracting *many high-quality pairs*. To date, none of the prior work approaches satisfies both requirements of high quality and large quantity.

In this paper, we propose a novel technique that fills this gap (see Figure 2 for an overview). Our key idea is to treat the problem as a classification problem: given an NL intent (*e.g.*, the SO question title) and *all* contiguous code fragments extracted from all answers of that question as candidate matches (for each answer code block, we consider all line-contiguous fragments as candidates, *e.g.*, for a 3-line code block 1-2-3, we consider fragments consisting of lines 1, 2, 3, 1-2, 2-3, and 1-2-3), we use a data-driven classifier to decide if a candidate aligns well with the NL intent. Our model uses two kinds of information to evaluate candidates: (1) *structural features*, which are hand-crafted but largely language-independent, and try to estimate whether a candidate code fragment is valid syntactically, and (2) *correspondence features*, automatically learned, which try to estimate whether the NL and code correspond to each other semantically. Specifically, for the latter we use a model inspired by recent developments in neural network models for machine translation [4], which can calculate bidirectional conditional probabilities of the code given the NL and vice-versa. We evaluate our method on two small labeled data sets of Python and Java code that we created from SO. We show that our approach can extract significantly more, and significantly more accurate code snippets in both languages than previous baseline approaches. We also demonstrate that the classifier is still effective even when trained on Python then used to extract snippets for Java, and vice-versa, which demonstrates potential for generalizability to other programming languages without laborious annotation of correct NL-code pairs.

[1] There is at most one accepted answer per question; see green check symbol in Fig 1.

Our approach strikes a good balance between training effort, scale, and accuracy: the correspondence features can be trained without human intervention on readily available data from SO; the structural features are simple and easy to apply to new programming languages; and the classifier requires minimal amounts of manually labeled data (we only used 152 Python and 102 Java manually-annotated SO question threads in total). Even so, compared to the heuristic techniques from prior work [3, 15, 44], our approach is able to extract up to an order of magnitude more aligned pairs with no loss in accuracy, or reduce errors by more than half while holding the number of extracted pairs constant.

Specifically, we make the following contributions:

- We propose a novel technique for extracting aligned NL-code pairs from SO posts, based on a classifier that combines snippet structural features, readily extractable, with bidirectional conditional probabilities, estimated using a state-of-the-art neural network model for machine translation.
- We propose a protocol and tooling infrastructure for generating labeled training data.
- We evaluate our technique on two data sets for Python and Java and discuss performance, potential for generalizability to other languages, and lessons learned.
- All annotated data, the code for the annotation interface and the mining algorithm are available at http://conala-corpus.github.io.

2 PROBLEM SETTING

Stack Overflow (SO) is the most popular Q&A site for programming related questions, home to millions of users. An example of the SO interface is shown in Figure 1, with a question (in the upper half) and a number of answers by different SO users. Questions can be about anything programming-related, including features of the programming language or best practices. Notably, many questions are of the *"how to"* variety, *i.e.*, questions that ask how to achieve a particular goal such as *"sorting a list"*, *"merging two dictionaries"*, or *"removing duplicates in lists"* (as shown in the example); for example, around 36% of the Python-tagged questions are in this category, as discussed later in Section 3.2. These *how-to* questions are the type that we focus on in this work, since they are likely to have corresponding snippets and they mimic NL-to-code (or vice versa) queries that users might naturally make in the applications we seek to enable, *e.g.*, code retrieval and synthesis.

Specifically, we focus on extracting triples of three specific elements of the content included in SO posts:

- **Intent:** A description in English of what the questioner wants to do; usually corresponds to some portion of the post title.
- **Context:** A piece of code that does not implement the intent, but is necessary setup, *e.g.*, import statements, variable definitions.
- **Snippet:** A piece of code that actually implements the intent.

An example of these three elements is shown in Figure 1. Several interesting points can be gleamed from this example. *First,* and most important, we can see that not all snippets in the post are implementing the original poster's intent: only two of four highlighted are actual examples of how to remove duplicates in lists, the other two highlighted are context, and others still are examples of interpreter output. If one is to train, *e.g.*, a data-driven system for code synthesis from NL, or code retrieval using NL, only the snippets, or

portions of snippets, that actually implement the user intent should be used. Thus, we need a mining approach that can distinguish which segments of code are actually legitimate implementations, and which can be ignored. *Second,* we can see that there are often several alternative implementations with different trade-offs (*e.g.*, the first example is simpler in that it doesn't require additional modules to be imported first). One would like to be able to extract all of these alternatives, *e.g.*, to present them to users in the case of code retrieval[2] or, in the case of code summarization, see if any occur in the code one is attempting to summarize.

These aspects are challenging even for human annotators, as we illustrate next.

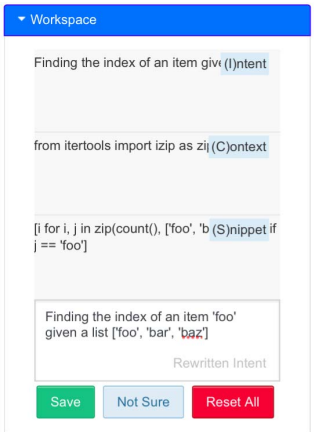
3 MANUAL ANNOTATION

To better understand the challenges with automatically mining aligned NL-code snippet pairs from SO posts, we manually annotated a set of labeled NL-code pairs. These also serve as the gold-standard data set for training and evaluation. Here we describe our annotation method and criteria, salient statistics about the data collected, and challenges faced during annotation.

For each target programming language, we first obtained all questions from the official SO data dump[3] dated March 2017 by filtering questions tagged with that language. We then generated the set of questions to annotate by: (1) including all top-100 questions ranked by view count; and (2) sampling 1,000 questions from the probability distribution generated by their view counts on SO; we choose this method assuming that more highly-viewed questions are more important to consider as we are more likely to come across them in actual applications. While each question may have any number of answers, we choose to only annotate the top-3 highest-scoring answers to prevent annotators from potentially spending a long time on a single question.

3.1 Annotation Protocol and Interface

Consistently annotating the intent, context, and snippet for a variety of posts is not an easy task, and in order to do so we developed and iteratively refined a web annotation interface and a protocol with detailed annotation criteria and instructions.



The annotation interface allows users to select and label parts of SO posts as (I)intent, (C)ontext, and (S)nippet using shortcut keys, as well as rewrite the intent to better match the code (*e.g.*, adding variable names from the snippet into the original intent), in consideration of potential future applications that may require more precisely aligned NL-code data; in the following experiments we solely consider the intent and snippet, and reserve examination of the context and re-written intent for future work. Multiple NL-code pairs that are part of the same post can be annotated this way. There is also a "not applicable" button that allows users to skip posts that are not of the "how to" variety, and a "not sure" button, which can be used when the annotator is uncertain.

The annotation criteria were developed by having all authors attempt to perform annotations of sample data, gradually adding notes of the difficult-to-annotate cases to a shared document. We completed several pilot annotations for a sample of Python questions, iteratively discussing among the research team the annotation criteria and the difficult-to-annotate cases after each, before finalizing the annotation protocol. We repeated the process for Java posts. Once we converged on the final annotation standards in both languages, we discarded all pilot annotations, and one of the authors (a graduate-level NLP researcher and experienced programmer) re-annotated the entire data set according to this protocol.

While we cannot reflect all difficult cases here for lack of space, below is a representative sample from the Python instructions:

- **Intents:** Annotate the command form when possible (*e.g.*, "how do I merge dictionaries" will be annotated as "merge dictionaries"). Extraneous words such as "in Python" can be ignored. Intents will almost always be in the title of the post, but intents expressed elsewhere that are different from the title can also be annotated.
- **Context:** Contexts are a set of statements that do not directly reflect the annotated intent, but may be necessary in order to get the code to run, and include import statements, variable definitions, and anything else that is necessary to make sure that the code executes. When no context exists in the post this field can be left blank.
- **Snippet:** Try to annotate full lines when possible. Some special tokens such as ">>>", "`print`", and "`In[...]`" that appear at the beginning of lines due to copy-pasting can be included. When the required code is encapsulated in a function, the function definition can be skipped.
- **Re-written intent:** Try to be accurate, but try to make the minimal number of changes to the original intent. Try to reflect all of the free variables in the snippet to be conducive to future automatic matching of these free variables to the corresponding position in code. When referencing string literals or numbers, try to write exactly as written in the code, and surround variables with a grave accent "`".

Table 1: Details of the labeled data set.

Lang.	#Annot.	#Ques.	#Answer Posts	#Code Blocks	Avg. Code Length	%Full Blocks	%Annot. with Context
Python	527	142	412	736	13.2	30.7%	36.8%
Java	330	100	297	434	30.6	53.6%	42.4%

3.2 Annotation Outcome

We annotated a total of 418 Python questions and 200 Java questions. Of those, 152 in Python and 102 in Java were judged as annotatable (*i.e.*, the *"how-to"* style questions described in Section 2), resulting in 577 Python and 354 Java annotations. We then removed the

[2]Ideally one would also like to present a description of the trade-offs, but mining this information is a challenge beyond the scope of this work.
[3]Available online at https://archive.org/details/stackexchange

annotations marked as "not sure" and all unparsable code snippets.[4] In the end we generated 527 Python and 330 Java annotations, respectively. Table 1 lists basic statistics of the annotations. Notably, compared to Python, Java code snippets are longer (13.2 vs. 30.6 tokens per snippet), and more likely to be full code blocks (30.7% vs. 53.6%). That is, in close to 70% of cases for Python, the code snippet best-aligned with the NL intent expressed in the question title was *not* a full code block (SO uses special HTML tags to highlight code blocks, recall the example in Figure 1) from one of the answers, but rather a subset of it; similarly, the best-aligned Java snippets were *not* full code blocks in almost half the cases. This confirms the importance of mining code snippets beyond the level of entire code blocks, a limitation of prior approaches.

Overall, we found the annotation process to be non-trivial, which raises several noteworthy threats to validity: (1) it can be difficult for annotators to distinguish between incorrect solutions and unusual or bad solutions that are nonetheless correct; (2) in cases where a single SO question elicits many correct answers with many implementations and code blocks, annotators may not always label all of them; (3) long and complex solutions may be mis-annotated; and (4) inline code blocks are harder to recognize than stand-alone code blocks, increasing the risk of annotators missing some. We made a best effort to minimize the impact of these threats by carefully designing and iteratively refining our annotation protocol.

4 MINING METHOD

In this section, we describe our mining method (see Figure 2 for an overview). As mentioned in Section 2, we frame the problem as a classification problem. First, for every "how to" SO question we consider its title as the intent and extract all contiguous lines from across all code blocks in the question's answers (including those we might manually annotate as context; inline code snippets are excluded) as candidate implementations of the intent, as long as we could parse the candidate snippets.[4] There are some cases where the title is not strictly equal to the intent, which go beyond the scope of this paper; for the purpose of learning the model we assume the title is representative. This step generates, for every SO question considered, a set of pairs (intent I, candidate snippet S). For example, the second answer in Figure 1, containing a three-line-long code block, would generate six line-contiguous candidate snippets, corresponding to lines 1, 2, 3, 1-2, 2-3, and 1-2-3. Our candidate snippet generation approach, though clearly not the only possible approach (1) is simple and language-independent, (2) is informed by our manual annotations, and (3) it gives good coverage of all possible candidate snippets.

Then, our task is, given a candidate pair (I, S), to assign a label y representing whether or not the snippet S reflects the intent I; we define y to equal 1 if the pair matches and -1 otherwise. Our general approach to making this binary decision is to use machine learning to train a classifier that predicts, for every pair (I, S), the probability that S accurately implements I, *i.e.*, $P(y = 1|I, S)$, based on a number of *features* (Sections 4.1 and 4.2). As is usual in supervised learning, our system first requires an offline *training* phase that learns the parameters (*i.e.*, feature weights) of the classifier, for which we use the annotated data described above (Section 3). This way, we can

[4]We use the built-in `ast` parser module for Python, and `JavaParser` for Java.

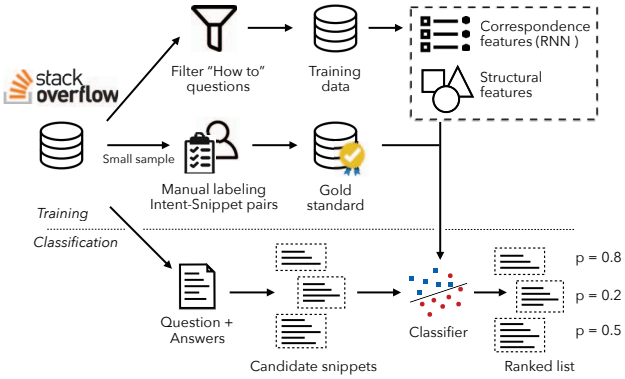


Figure 2: Overview of our approach.

apply our system to an SO page of interest, and compute $P(y = 1|I, S)$ for each possible intent/candidate snippet pair mined from the SO page. We choose logistic regression as our classifier, as implemented in the `scikit-learn` Python package.

As human annotation to generate training data is costly, our goal is to keep the amount of manually labeled training data to a minimum, such that scaling our approach to other programming languages in the future can be feasible. Therefore, to ease the burden on the classifier in the face of limited training data, we combined two types of features: hand-crafted *structural* features of the code snippets (Section 4.1) and machine learned *correspondence* features that predict whether intents and code snippets correspond to each-other semantically (Section 4.2). Our intuition, again informed by the manual annotation, was that "good" and "bad" pairs can often be distinguished based on simple hand-crafted features; these features could eventually be learned (as opposed to hand-crafted), but this would require more labeled training data, which is relatively expensive to create.

4.1 Hand-crafted Code Structure Features

The structural features are intended to distinguish whether we can reasonably expect that a particular piece of code implements an intent. We aimed for these features to be both informative and generally applicable to a wide range of programming languages. These features include the following:

- **FullBlock, StartOfBlock, EndOfBlock:** A code block may represent a single cohesive solution. By taking only a piece of a code block, we may risk acquiring only a partial solution, and thus we use a binary feature to inform the classifier of whether it is looking at a whole code block or not. On the other hand, as shown in Figure 1, many code blocks contain some amount of context before the snippet, or other extraneous information, *e.g.*, print statements. To consider these, we also add binary features indicating that a snippet is at the start or end of its code block.
- **ContainsImport, StartsWithAssignment, IsValue:** Additionally, some statements are highly indicative of a statement being context or extraneous. For example, import statements are highly indicative of a particular line being context instead of the snippet itself, and thus we add a binary feature indicating

whether an import statement is included. Similarly, variable assignments are often context, not the implementation itself, and thus we add another feature indicating whether the snippet starts with a variable assignment. Finally, we observed that in SO (particularly for Python), it was common to have single lines in the code block that consisted of only a variable or value, often as an attempt to print these values to the interactive terminal.

- **AcceptedAns, PostRank1, PostRank2, PostRank3:** The quality of the post itself is also indicative of whether the answer is likely to be valid or not. Thus, we add several features indicating whether the snippet appeared in a post that was the accepted answer or not, and also the rank of the post within the various answers for a particular question.
- **OnlyBlock:** Posts with only a single code block are more likely to have that snippet be a complete implementation of the intent, so we added another feature indicating when the extracted snippet is the only one in the post.
- **NumLinesX:** Snippets implementing the intent also tend to be concise, so we added features indicating the number of lines in the snippet, bucketed into X = 1, 2, 3, 4-5, 6-10, 11-15, >15.
- **Combination Features:** Some features can be logically combined to express more complex concepts. E.g., AcceptedAns + OnlyBlock + WholeBlock can express the strategy of selecting whole blocks from accepted answers with only one block, as used in previous work [15, 44]. We use this feature and two other combination features: specifically ¬StartWithAssign + EndOfBlock and ¬StartWithAssign + NumLines1.

4.2 Unsupervised Correspondence Features

While all of the features in the previous section help us determine which code snippets are likely to implement *some* intent, they say nothing about whether the code snippet actually implements *the particular* intent I that is currently under consideration. Of course considering this correspondence is crucial to accurately mining intent-snippet pairs, but how to evaluate this correspondence computationally is non-trivial, as there are very few hard and fast rules that indicate whether an intent and snippet are expressing similar meaning. Thus, in an attempt to capture this correspondence, we take an indirect approach that uses a potentially-noisy (*i.e.*, not manually validated) but easy-to-construct data set to train a probabilistic model to approximately capture these correspondences, then incorporate the predictions of this noisily trained model as features into our classifier.

Training data of correspondence features: Apart from our manually-annotated data set, we collected a relatively large set of intent-snippet pairs using simple heuristic rules for learning the correspondence features. The data set is created by pairing the question titles and code blocks from all SO posts, where (1) the code block comes from an SO answer that was accepted by the original poster, and (2) there is only one code block in this answer. Of course, many of these code blocks will be noisy in the sense that they contain extraneous information (such as extra import statements or variable definitions, *etc.*), or not directly implement the intent at all, but they will still be of use for learning which NL expressions in the intent tend to occur with which types of source code.

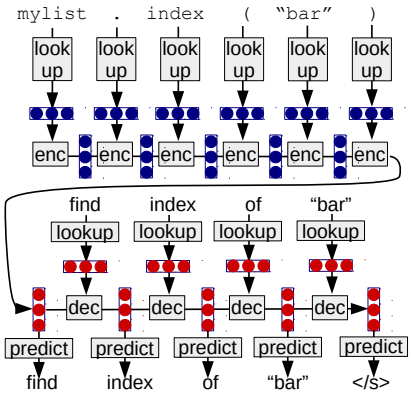


Figure 3: An example of neural MT encoder-decoder framework used in calculating correspondence scores.

Learning a model of correspondence: Given the training data above, we need to create a model of the correspondence between the intent I and snippet S. To this end, we build a statistical model of the bi-directional probability of the intent given the snippet $P(I \mid S)$, and the probability of the snippet given the intent $P(S \mid I)$.

Specifically, we follow previous work that has noted that models from *machine translation* (MT; [19]) are useful for learning the correspondences between natural language and code for the purposes of code summarization [15, 30], code synthesis from natural language [21], and code retrieval [3]. In particular, we use a model based on *neural MT* [4, 17], a method for MT based on neural networks that is well-suited for this task of learning correspondences for a variety of reasons, which we outline below after covering the basics. To take the example of using a neural MT model that attempts to generate an intent I given a snippet S, these models work by incrementally generating each word of the intent $i_1, i_2, \ldots, i_{|I|}$ one word at a time (the exact same process can be performed in the reverse direction to generate a snippet S given intent I). For example, if our intent is *"download and save an http file"*, the model would first predict and generate *"download"*, then *"and"*, then *"save"*, etc. This is done in a probabilistic way by calculating the probability of the first word given the snippet $P(i_1 \mid S)$ and outputting the word in the vocabulary that maximizes this probability, then calculating the probability of the second word given the first word and the snippet $P(i_2 \mid S, i_1)$ and similarly outputting the word with the highest probability, etc. Incidentally, if we already know a particular intent I and want to calculate its probability given a particular snippet S (for example to use as features in our classifier), we can also do so by calculating the probability of each word in the intent and multiplying them together as follows:

$$P(I \mid S) = P(i_1 \mid S)P(i_2 \mid S, i_1)P(i_3 \mid S, i_1, i_2)\ldots \quad (1)$$

So how do neural MT models calculate this probability? We will explain a basic outline of a basic model called the *encoder-decoder model* [39], and refer readers to references for details [4, 25, 39]. The encoder-decoder model, as shown in Figure 3, works in two stages: First, it *encodes* the input (in this case S) into a *hidden vector* of continuous numbers $\boldsymbol{h}$ using an encoding function

$$\boldsymbol{h}_{|S|} = \text{encode}(S). \quad (2)$$

This function generally works in two steps: looking up a vector of numbers representing each word (often called "word embeddings" or "word vectors"), then incrementally adding information about these embeddings one word at a time using a particular variety of network called a *recurrent neural network* (RNN). To take the specific example shown in the figure, at the first time step, we would look up a word embedding vector for the first word "`mylist`", $\boldsymbol{e}_1 = \boldsymbol{e}_{\texttt{mylist}}$ and then perform a calculation such as the one below to calculate the hidden vector for the first time step:

$$\boldsymbol{h}_1 = \tanh(W_{\text{enc,e}}\boldsymbol{e}_1 + b_{\text{enc}}), \tag{3}$$

where $W_{\text{enc,e}}$ and b_{enc} are a matrix and vector that are parameters of the model, and $tanh(\cdot)$ is the hyperbolic tangent function used to "squish" the values to be between -1 and 1. In the next time step, we would do the same for the symbol ".", using its embedding $\boldsymbol{e}_2 = \boldsymbol{e}_{.}$, and in the calculation from the second step onward we also use the result of the previous calculation (in this case $\boldsymbol{h}_2$):

$$\boldsymbol{h}_1 = \tanh(W_{\text{enc,h}}\boldsymbol{h}_1 + W_{\text{enc,e}}\boldsymbol{e}_2 + b_{\text{enc}}). \tag{4}$$

By using the hidden vector from the previous time step, the RNN is able to "remember" features of the previously occurring words within this vector, and by repeating this process until the end of the input sequence, it (theoretically) has the ability to remember the entire content of the input within this vector.

Once we have encoded the entire source input, we can use this encoded vector to predict the first word of the output. This is done by multiplying the vector $\boldsymbol{h}$ with another weight matrix to calculate a score $\boldsymbol{g}$ for each word in the output vocabulary:

$$\boldsymbol{g}_1 = W_{\text{pred}}\boldsymbol{h}_{|S|} + \boldsymbol{b}_{\text{pred}}. \tag{5}$$

We then predict the actual probability of the first word in the output sentence, for example *"find"*, by using the *softmax* function, which exponentiates all of the scores in the output vocabulary and then normalizes these scores so that they add up to one:

$$P(i_1 = \text{"find"}) = \frac{\exp(g_{\text{find}})}{\sum_{\tilde{g}} \exp(\tilde{g})}. \tag{6}$$

We use a neural MT model with this basic architecture, with the addition of a feature called *attention*, which, put simply, allows the model to "focus" on particular words in the source snippet S when generating the intent I. The details of attention are beyond the scope of this paper, but interested readers can reference [4, 22].

Why attentional neural MT models?: Attention-based neural MT models are well-suited to the task of learning correspondences between natural language intents and code snippets for a number of reasons. First, they are a purely probabilistic model capable of calculating $P(S \mid I)$ and $P(I \mid S)$, which allows them to easily be incorporated as features in our classifier, as described in the following paragraph. Second, they are powerful models that can learn correspondences on a variety of levels; from simple phenomena such as direct word-by-word matches [12], to soft paraphrases [36], to weak correspondences between keywords and large documents for information retrieval [14]. Finally, they have demonstrated success in a number of NL-code related tasks [2, 3, 21, 47], which indicates that they could be useful as part of our mining approach as well.

Incorporating correspondence probabilities as features: For each intent I and candidate snippet S, we calculate the probabilities $P(S \mid I)$ and $P(I \mid S)$, and add them as features to our classifier, as we did with the hand-crafted structural features in Section 4.1.

- **SGivenI, IGivenS:** Our first set of features are the logarithm of the probabilities mentioned above: $\log P(S \mid I)$ and $\log P(I \mid S)$.[5] Intuitively, these probabilities will be indicative of S and I being a good match because if they are not, the probabilities will be low. If the snippet and the intent are not a match at all, both features will have a low value. If the snippet and intent are partial matches, but either the snippet S or intent I contain extraneous information that cannot be predicted from the counterpart, then SGivenI and IGivenS will have low values respectively.
- **ProbMax, ProbMin:** We also represent the max and min of $\log P(S \mid I)$ and $\log P(I \mid S)$. In particular, the ProbMin feature is intuitively helpful because pairs where the probability in *either* direction is low are likely not good pairs, and this feature will be low in the case where either probability is low.
- **NormalizedSGivenI, NormalizedIGivenS:** In addition, intuitively we might want the *best* matching NL-code pairs within a particular SO page. In order to capture this intuition, we also normalize the scores over all posts within a particular page so that their mean is zero and standard deviation is one (often called the z-score). In this way, the pairs with the best scores within a page will get a score that is significantly higher than zero, while the less good scores will get a score close to or below zero.

5 EVALUATION

In this section we evaluate our proposed mining approach. We first describe the experimental setting in Section 5.1 before addressing the following research questions:

(1) How does our mining method compare with existing approaches across different programming languages? (Section 5.2)
(2) How do the structural and correspondence features impact the system's performance? (Section 5.2)
(3) Given that annotation of data for each language is laborious, is it possible to use a classifier learned on one programming language to perform mining on other languages? (Section 5.3)
(4) What are the qualitative features of the NL-code pairs that our method succeeds or fails at extracting? (Section 5.4)

We show that our method clearly outperforms existing approaches and shows potential for reuse *without retraining*, we uncover trade-offs between performance and training complexity, and we discuss limitations, which can inform future work.

5.1 Experimental Settings

We conduct experimental evaluation on two programming languages: Python and Java. These languages were chosen due to their large differences in syntax and verbosity, which have been shown to effect characteristics of code snippets on SO [45].

Learning unsupervised features: We start by filtering the SO questions in the Stack Exchange data dump[3] by tag (Python and Java), and we use an existing classifier [15] to identify the *how-to* style questions. The classifier is a support vector machine trained by

[5] We take the logarithm of the probabilities because the actual probability values tend to become very small for very long sequences (*e.g.*, 10^{-50} to 10^{-100}), while the logarithm is in a more manageable range (*e.g.*, -50 to -100).

Table 2: Details of the NL-code data used for learning unsupervised correspondence features.

Lang.	Training Data (NL/Code Pairs)	Validation Data	Intents		Code	
			Avg. Length	Vocabulary Size	Avg. Length	Vocabulary Size
Python	33,946	3,773	11.9	12,746	65.4	30,286
Java	37,882	4,208	11.6	13,775	65.7	29,526

bootstrapping from 100 labeled questions, and achieves over 75% accuracy as reported in [15]. We then extract intent/snippet pairs from all these questions as described in Section 4.2, collecting 33,946 pairs for Python and 37,882 for Java. Next we split the data set into training and validation sets with a ratio of 9:1, keeping the 90% for training. Statistics of the data set are listed in Table 2.[6]

We implement our neural correspondence model using the DyNet neural network toolkit [26]. The dimensionality of word embedding and RNN hidden states is 256 and 512. We use dropout [38], a standard method to prevent overfitting, on the input of the last softmax layer over target words ($p = 0.5$), and recurrent dropout [11] on RNNs ($p = 0.2$). We train the network using the widely used optimization method Adam [18]. To evaluate the neural network, we use the remaining 10% of pairs left aside for testing, retaining the model with the highest likelihood on the validation set.

Evaluating the mining model: For the logistic regression classifier, which uses the structural and correspondence features described above, the latter computed by the previous neural network, we use our annotated intent/snippet data (Section 3.2)[7] during **5-fold cross validation**. Recall, our code mining model takes as input a SO question (*i.e.*, intent reflected by the question title) with its answers, and outputs a ranked list of candidate intent/snippet pairs (with probability scores). For evaluation, we first rank all candidate intent/snippet pairs for all questions, and then compare the ranked list with gold-standard annotations. We present the results using standard precision-recall (PR) and Receiver Operating Characteristic (ROC) curves. In short, a PR curve shows the precision w.r.t. recall for the top-k predictions in the ranked list, with k from 1 to the number of candidates. A ROC curve plots the true positive rates w.r.t. false positive rates in similar fashion. We also compute the Area Under the Curve (AUC) scores for all ROC curves.

Baselines: As baselines for our model (denoted as Full), we implement three approaches reflecting prior work and sensible heuristics:

AcceptOnly is the state-of-the art from prior work [15, 44]; it selects the whole code snippet in the *accepted* answers containing exactly one code snippet.

All denotes the baseline method that exhaustively selects all full code blocks in the top-3 answers in a post.

Random is the baseline that randomly selects from all consecutive code segment candidates.

Similarly to our model, we enforce the constraint that all mined code snippets given by the baseline approaches should be parseable.

Additionally, to study the impact of hand-crafted **Structural** versus learned **Correspondence** features, we also trained versions of our model with either of the two types of features only.

5.2 Results and Discussion

Our main results are depicted in Figure 4. First, we can see that the precision of the random baseline is only 0.10 for Python and 0.06 for Java. This indicates that only one in 10-17 candidate code snippets is judged to validly correspond to the intent, reflecting the difficulty of the task. The AcceptOnly and All baselines perform significantly better, with precision of 0.5 or 0.6 at recall 0.05-0.1 and 0.3-0.4 respectively, indicating that previous heuristic methods using full code blocks are significantly better than random, but still have a long way to go to extract broad-coverage and accurate NL-code pairs (particularly in the case of Python).[8]

Next, turning to the full system, we can see that the method with the full feature set significantly outperforms all baselines (Figures 4b and 4d): much better 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 (Structural v.s. Correspondence), we find that with structural features alone our model already significantly outperforms baseline approaches; and these features are particularly effective for Java. On the other hand, interestingly the correspondence features alone provide less competitive results. Still, the structural and correspondence features seem to be complementary, with the combination of the two feature sets further significantly improving performance, particularly on Python. A closer examination of the results generated the following insights.

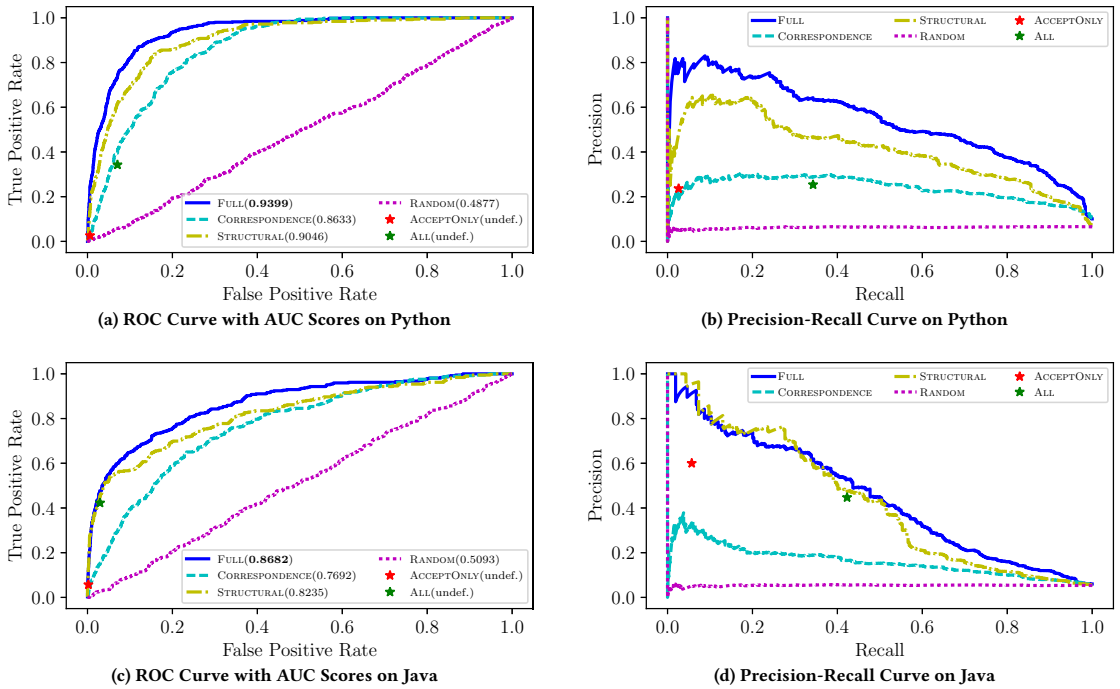
Why do correspondence features underperform? While these features effectively filter *totally* unrelated snippets, they still have a difficult time excluding related contextual statements, *e.g.*, imports, assignments. This is because (1) the snippets used for training correspondence features are full code blocks (as in §4.2), usually starting with `import` statements; and (2) the library names in `import` statements often have strong correspondence with the intents (*e.g.*, *"How to get current time in Python?" and* `import datetime`), yielding high correspondence probabilities.

What are the trends and error cases for structural features? Like the baseline methods, Structural tends to give priority to full code blocks; out of the top-100 ranked candidates for Structural, all were full code blocks (in contrast to only 21 for Correspondence). Because selecting code blocks is a reasonably strong baseline, and because the model has access to other strongly-indicative binary features that can be used to further prioritize its choices, it is able to achieve reasonable precision-recall scores only utilizing these features. However, unsurprisingly, it lacks fine granularity in terms

[6]Note that this data may contain some of the posts included in the cross-validation test set with which we evaluate our model later. However, even if it does, we are not using the annotations themselves in the training of the correspondence features, so this does not pose a problem with our experimental setting.

[7]Recall that our annotated data contains only how-to style questions, and therefore question filtering is not required. When applying our mining method to the full SO data, we could use the how-to question classifier in [15].

[8]Interestingly, AcceptOnly and All have similar precision, which might be due to two facts. First, we enforce all candidate snippets to be syntactically correct, which rules out erroneous candidates like input/output examples. Second, we use the top 3 answers for each question, which usually have relatively high quality.



(a) ROC Curve with AUC Scores on Python

(b) Precision-Recall Curve on Python

(c) ROC Curve with AUC Scores on Java

(d) Precision-Recall Curve on Java

Figure 4: Evaluation Results on Mining Python (a)(b) and Java (c)(d)

of pinpointing exact code segments that correspond to the intents; when it tries to select partial code segments, the results are likely to be irrelevant to the intent. As an example, we find that Structural tends to select the last line of code at each code block, since the learned weights for LineNum=1 and EndsCodeBlock features are high, even though these often consist of auxiliary `print` statement or even simply `pass` (for Python).

What is the effect of the combination? When combining Structural and Correspondence features together, the full model has the ability to use the knowledge of the Structural model extract full code blocks or ignore imports, leading to high performance in the beginning stages. Then, in the latter and more difficult cases, it is able to more effectively cherry-pick smaller snippets based on their correspondence properties, which is reflected in the increased accuracy on the right side of the ROC and precision-recall curves.

How do the trends differ between programming languages? Compared with the baseline approaches AcceptOnly and All, our full model performs significantly better on Python. We hypothesize that this is because learning correspondences between intent/snippet pairs for Java is more challenging. Empirically, Python code snippets are much shorter, and the average number of tokens for predicted code snippets on Python and Java is 11.6 and 42.4, respectively. Meanwhile, since Java code snippets are more verbose and contain significantly more boilerplate (*e.g.*, class/function definitions, type declaration, exception handling, *etc.*), estimating correspondence scores using neural networks is more challenging.

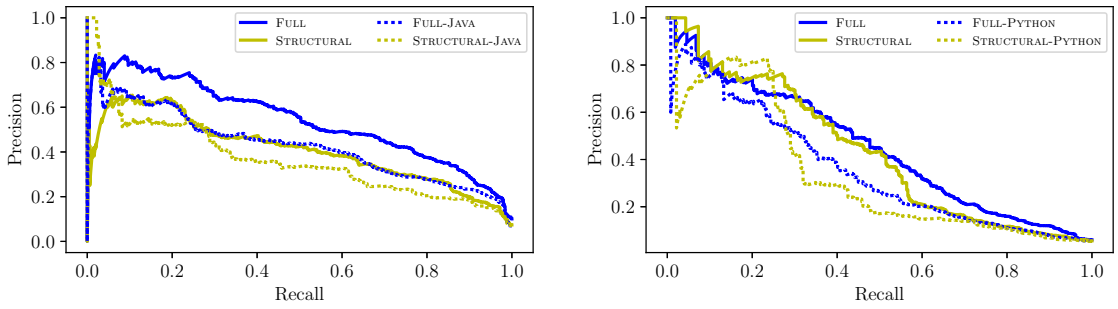
Also note that the Structural model performs much better on Java than on Python. This is due to the fact that Java annotations are more likely to be full code blocks (see Table 1), which can be easily captured by our designed features like FullBlock. Nevertheless, adding correspondence features is clearly helpful for the harder cases for both programming languages. For instance, from the ROC curve in Figure 4c, our full model achieves higher true positive rates compared with Structural, registering higher AUC scores.

5.3 Must We Annotate Each Language?

As discussed in §3, collecting high-quality intent/snippet annotations to train the code mining model for a programming language can be costly and time-consuming. An intriguing research question is how we could *transfer* the learned code mining model from one programming language and use it for mining intent/snippet data for another language. To test this, we train a code mining model using the annotated intent/snippet data on language A, and evaluate using the annotated data on language B.[9] This is feasible since almost all of the features used in our system is language-agnostic.[10] Also note values of a specific feature might have different ranges for different languages. As an example, the average value of SGivenI feature for Python and Java is -23.43 and -47.64, respectively. To mitigate this issue, we normalize all feature values to zero mean and unit variance before training the logistic regression classifier.

Figures 5a and 5b show the precision-recall curves for applying Java (Python) mining model on Python (Java) data. We report

[9]We still train the correspondence model using the target language unlabeled data.
[10]The only one that was not applicable to both languages was the SingleValue feature for Python, which helps rule out code that contains only a single value. We omit this feature in the cross-lingual experiments.



(a) Java ↦ Python (b) Python ↦ Java

Figure 5: Precision-Recall Curves for Transfer Learning on Java ↦ Python (a) and Python ↦ Java (b)

results for both the Structural model and our full model, and compare with the original models trained on the target programming language. Unsurprisingly, the original full model tuned on the target language still performs the best. Nevertheless, we observe that the performance gap between the original full model and the transferred one is surprisingly small. Notably, we find that overall the transferred full model (Full-Java) performs second best on Python, even outperforming the original Structural model. These results are encouraging, in that they suggest that it is likely feasible to train a single code mining classifier and then apply it to different programming languages, even those for which we do not have any annotated intent/snippet data.

5.4 Successful and Failed Examples

As illustration, we showcase successful and failed examples of our proposed approach, for Python in Table 3 and for Java in Table 4. Given a SO question (intent), we show the top-3 most probable code snippets. First, we find our model can correctly identify code snippets for various types of intents, even when the target snippets are not full code blocks. I_1 and I_6 demonstrate that our model can leave contextual information like variable definitions in the original SO posts and only retain the actual implementation of the intent.[11] I_2, I_3 and I_7 are more interesting: in the original SO post, there could be multiple possible solutions in the same code block (I_2 and I_7), or the gold-standard snippets are located inside larger code structures like a for loop (S_2 for I_3). Our model learns to "break down" the solutions in single code block into multiple snippets, and extract the actual implementation from large code chunks.

We also identify four sources of errors:

- *Incomplete code:* Some code snippets are incomplete, and the model fails to include intermediate statements (*e.g.*, definitions of custom variables or functions) that are part of the implementation. For instance, S_3 for I_3 misses the definition of the `keys_to_keep`, which is the set of keys excluding the key to remove.
- *Including auxiliary info:* Sometimes the model fails to exclude auxiliary code segments like the extra context definition (*e.g.*, S_1 for I_8) and `print` function. This is especially true for Java, where full code blocks are likely to be correct snippets, and the model tends to bias towards larger code chunks.
- *Spurious cases:* We identify two "spurious" cases where our correspondence feature often do not suffice. (1) *Counter examples*: the S_1 for I_4 is mentioned in the original post as a counter example, but the values of correspondence features are still high since `append()` is highly related to *"append it to another list"* in the intent. (2) *Related implementation*: I_5 shows an example where the model has difficulty distinguishing between the actual snippets and related implementations.
- *Annotation error:* We find cases where our annotation is incomplete. For instance, S_1 for I_9 should be correct. As discussed in Section 3, guaranteeing coverage in the annotation process is non-trivial, and we leave this as a challenge for future work.

[11] We refer readers to the original SO page for reference.

6 RELATED WORK

A number of previous works have proposed methods for mining intent-snippet pairs for purposes of code summarization, search, or synthesis. We can view these methods from several perspectives:

Data Sources: First, what data sources do they use to mine their data? Our work falls in the line of mining intent-snippet pairs from SO (*e.g.*, [15, 44, 46, 48]), while there has been research on mining from other data sources such as API documentation [5, 6, 24], code comments [43], specialized sites [32], parameter/method/class names [1, 37], and developer mailing lists [31]. It is likely that it could be adapted to work with other sources, requiring only changes in the definition of our structural features to incorporate insights into the data source at hand.

Methodologies: Second, what is the methodology used therein, and can it scale to our task of gathering large-scale data across a number of languages and domains? Several prior work approaches used heuristics to extract aligned intent-snippet pairs [6, 44, 48]). Our approach also contains an heuristic component. However, as evidenced by our experiments here, our method is more effective at extracting accurate intent-snippet pairs.

Some work on code search has been performed by retrieving candidate code snippets given an intent based on weighted keyword matches and other features [29, 42]. These methods similarly aim to learn correspondences between natural language queries and returned code, but they are tailored specifically for performing code search, apply a more rudimentary feature set (*e.g.*, they do not employ neural network-based correspondence features) than we

Table 3: Examples of Mined Python Code

I_1: Remove specific characters from a string in python
URL: https://stackoverflow.com/questions/3939361/
Top Predictions:
S_1 `string.replace('1', '')` ✓
S_2 `line = line.translate(None, '!@#$')` ✓
S_3 `line = re.sub('[!@#$]', '', line)` ✓

I_2: Get Last Day of the Month in Python
URL: https://stackoverflow.com/questions/42950/
Top Predictions:
S_1 `calendar.monthrange(year, month)[1]` ✓
S_2 `calendar.monthrange(2100, 2)` ✓
S_3 `(datetime.date(2000, 2, 1) - datetime.timedelta(days=1))` ✓

I_3: Delete a dictionary item if the key exists
URL: https://stackoverflow.com/questions/15411107/
Top Predictions:
S_1 `mydict.pop('key', None)` ✓
S_2 `del mydict[key]` ✓
S_3 `new_dict = {k: mydict[k] for k in keys_to_keep}` ✗

I_4: Python: take the content of a list and append it to another list
URL: https://stackoverflow.com/questions/8177079/
Top Predictions:
S_1 `list2.append(list1)` ✗
S_2 `list2.extend(list1)` ✓
S_3 `list1.extend(mylog)` ✓

I_5: Converting integer to string in Python?
URL: https://stackoverflow.com/questions/961632/
Top Predictions:
S_1 `int('10')` ✗
S_2 `str(10); int('10')` ✗
S_3 `a.__str__()` ✓

Table 4: Examples of Mined Java Code

I_6: How to convert List<Integer> to int[] in Java?
URL: https://stackoverflow.com/questions/960431/
Top Predictions:
S_1
```
int[] array = list.stream().mapToInt(i -> i).toArray(); ✓
```
S_2
```
int[] intArray2 = ArrayUtils.toPrimitive(myList.toArray(
    NO_INTS)); ✗
```
S_3
```
int[] intArray = ArrayUtils.toPrimitive(myList.toArray(new
    Integer[myList.size()])); ✓
```

I_7: How do I compare strings in Java?
URL: https://stackoverflow.com/questions/513832/
Top Predictions:
S_1 `new String("test").equals("test");` ✓
S_2 `Objects.equals(null, "test");` ✓
S_3 `nullString1.equals(nullString2);` ✓

I_8: How do I set the colour of a label (coloured text) in Java?
URL: https://stackoverflow.com/questions/2966334/
Top Predictions:
S_1
```
JLabel title = new JLabel("I love stackoverflow!", JLabel.
    CENTER);
title.setForeground(Color.white); ✗
```
S_2
```
frame.add(new JLabel("<html>Text color: <font color='red'>
    red</font></html>")); ✓
```
S_3 `label.setForeground(Color.red);` ✓

I_9: Generating a Random Number between 1 and 10 Java
URL: https://stackoverflow.com/questions/20389890/
Top Prediction: (only show one for space reason)
S_1
```
public static int randInt(int min, int max) {
    Random rand = new Random();
    int randomNum = rand.nextInt((max - min) + 1) + min;
    return randomNum; } ✗ (annotation error)
```

do, and will generally not handle sub-code-block sized contexts, which proved important in our work.

We note that concurrent to this work, [46] also explored the problem of mining intent/code pairs from SO, identifying candidate code blocks of an intent using information from both the contextual texts and the code in an SO answer. Our approach, however, considers more fine-grained, sub-code-block sized candidates, aiming to recover code solutions that *exactly* answer the intent.

Finally, some work has asked programmers to manually write NL descriptions for code [20, 30], or vice-versa [41]. This allows for the generation of high-quality data, but is time consuming and does not scale beyond limited domains.

7 THREATS TO VALIDITY

Besides threats related to the manual labeling (Section 3.2), we note the following overall threats to the validity of our approach:

Annotation Error: Our code mining approach is based on learning from a small amount of annotated data, and errors in annotation may impact the performance of the system (see Sections 3 and 5.4).

Data Set Volume: Our annotated data set contains mainly high-ranked SO questions, and is relatively small (with a few hundreds of examples for each language), which could penitentially hinder the generalization ability of the system on lower-ranked questions. Meanwhile, we used cross-validation for evaluation, while evaluating our mining method on full-scale SO data would be ideal but challenging.

8 CONCLUSIONS

In this paper, we described a novel method for extracting aligned code/natural language pairs from the Q&A website Stack Overflow. The method is based on learning from a small number of annotated examples, using highly informative features that capture structural aspects of the code snippet and the correspondence between it and the original natural language query. Experiments on Python and Java demonstrate that this approach allows for more accurate and more exhaustive extraction of NL-code pairs than prior work. We foresee the main impact of this paper lying in the resources it would provide when applied to the full Stack Overflow data: the NL-code pairs extracted would likely be of higher quality and larger scale. Given that high-quality parallel NL-code data sets are currently a significant bottleneck in the development of new data-driven software engineering tools, we hope that such a resource will move the field forward. In addition, while our method is relatively effective compared to previous work, there is still significant work to be done on improving mining algorithms to deal with current failure cases, such as those described in Section 5.4. Our annotated data set and evaluation tools, publicly available, may provide an impetus towards further research in this area.

REFERENCES

[1] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2015. Suggesting Accurate Method and Class Names. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 38–49.

[2] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A Convolutional Attention Network for Extreme Summarization of Source Code. *arXiv preprint arXiv:1602.03001* (2016).

[3] Miltiadis Allamanis, Daniel Tarlow, Andrew D Gordon, and Yi Wei. 2015. Bimodal Modelling of Source Code and Natural Language. In *International Conference on Machine Learning (ICML)*. 2123–2132.

[4] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural Machine Translation by Jointly Learning to Align and Translate. In *International Conference on Learning Representations (ICLR)*.

[5] Antonio Valerio Miceli Barone and Rico Sennrich. 2017. A Parallel Corpus of Python Functions and Documentation Strings for Automated Code Documentation and Code Generation. *arXiv preprint arXiv:1707.02275* (2017).

[6] Shaunak Chatterjee, Sudeep Juvekar, and Koushik Sen. 2009. Sniff: A Search Engine for Java Using Free-form Queries. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 385–400.

[7] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Subhajit Roy, and others. 2016. Program Synthesis using Natural Language. In *International Conference on Software Engineering (ICSE)*. ACM, 345–356.

[8] Premkumar Devanbu. 2015. New Initiative: the Naturalness of Software. In *International Conference on Software Engineering (ICSE)*, Vol. 2. IEEE, 543–546.

[9] Christine Franks, Zhaopeng Tu, Premkumar Devanbu, and Vincent Hellendoorn. 2015. CACHECA: A Cache Language Model based Code Suggestion Tool. In *International Conference on Software Engineering (ICSE)*, Vol. 2. IEEE, 705–708.

[10] Mark Gabel and Zhendong Su. 2010. A Study of the Uniqueness of Source Code. In *International Symposium on Foundations of Software Engineering (FSE)*. ACM, 147–156.

[11] Yarin Gal and Zoubin Ghahramani. 2016. A Theoretically Grounded Application of Dropout in Recurrent Neural Networks. In *Annual Conference on Neural Information Processing Systems (NIPS)*. 1019–1027.

[12] Jiatao Gu, Zhengdong Lu, Hang Li, and Victor O.K. Li. 2016. Incorporating Copying Mechanism in Sequence-to-Sequence Learning. In *Annual Meeting of the Association for Computational Linguistics (ACL)*. ACL, 1631–1640.

[13] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. *Commun. ACM* 59, 5 (2016), 122–131.

[14] Po-Sen Huang, Xiaodong He, Jianfeng Gao, Li Deng, Alex Acero, and Larry Heck. 2013. Learning Deep Structured Semantic Models for Web Search using Clickthrough Data. In *International Conference on Information and Knowledge Management (CIKM)*. ACM, 2333–2338.

[15] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing Source Code using a Neural Attention Model. In *Annual Meeting of the Association for Computational Linguistics (ACL)*. ACL, 2073–2083.

[16] Alan Jaffe, Jeremy Lacomis, Edward J. Schwartz, Claire Le Goues, and Bogdan Vasilescu. 2018. Meaningful Variable Names for Decompiled Code: A Machine Translation Approach. In *International Conference on Program Comprehension (ICPC)*. ACM.

[17] Nal Kalchbrenner and Phil Blunsom. 2013. Recurrent Continuous Translation Models. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*. ACL, 1700–1709.

[18] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *CoRR* abs/1412.6980 (2014). http://arxiv.org/abs/1412.6980

[19] Philipp Koehn. 2010. *Statistical Machine Translation*. Cambridge Press.

[20] Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D Ernst. 2018. NL2Bash: A Corpus and Semantic Parser for Natural Language Interface to the Linux Operating System. *International Conference on Language Resources and Evaluation (LREC)* (2018).

[21] 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 *Conference on Empirical Methods in Natural Language Processing (EMNLP)*. ACL, 1918–1923.

[22] Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective Approaches to Attention-based Neural Machine Translation. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*. ACL, 1412–1421.

[23] Lance A Miller. 1981. Natural Language Programming: Styles, Strategies, and Contrasts. *IBM Systems Journal* 20, 2 (1981), 184–215.

[24] Dana Movshovitz-Attias and William W Cohen. 2013. Natural Language Models for Predicting Programming Comments. In *Annual Meeting of the Association for Computational Linguistics (ACL)*. ACL, 35–40.

[25] Graham Neubig. 2017. Neural Machine Translation and Sequence-to-Sequence Models: A Tutorial. *arXiv preprint arXiv:1703.01619* (2017).

[26] Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, Kevin Duh, Manaal Faruqui, Cynthia Gan, Dan Garrette, Yangfeng Ji, Lingpeng Kong, Adhiguna Kuncoro, Gaurav Kumar, Chaitanya Malaviya, Paul Michel, Yusuke Oda, Matthew Richardson, Naomi Saphra, Swabha Swayamdipta, and Pengcheng Yin. 2017. DyNet: The Dynamic Neural Network Toolkit. *arXiv preprint arXiv:1701.03980* (2017).

[27] Anh Tuan Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2014. Statistical Learning Approach for Mining API Usage Mappings for Code Migration. In *International Conference on Automated Software Engineering (ASE)*. ACM, 457–468.

[28] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2013. Lexical Statistical Machine Translation for Language Migration. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 651–654.

[29] Haoran Niu, Iman Keivanloo, and Ying Zou. 2016. Learning to Rank Code Examples for Code Search Engines. *Empirical Software Engineering* (2016), 1–33.

[30] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to Generate Pseudo-Code from Source Code Using Statistical Machine Translation. In *International Conference on Automated Software Engineering (ASE)*. IEEE, 574–584.

[31] Sebastiano Panichella, Jairo Aponte, Massimiliano Di Penta, Andrian Marcus, and Gerardo Canfora. 2012. Mining Source Code Descriptions from Developer Communications. In *International Conference on Program Comprehension (ICPC)*. IEEE, 63–72.

[32] Chris Quirk, Raymond Mooney, and Michel Galley. 2015. Language to Code: Learning Semantic Parsers for If-This-Then-That Recipes. In *Annual Meeting of the Association for Computational Linguistics (ACL)*. 878–888.

[33] Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract Syntax Networks for Code Generation and Semantic Parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Vancouver, Canada, 1139–1149. http://aclweb.org/anthology/P17-1105

[34] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Code". In *ACM Symposium on Principles of Programming Languages (POPL)*. ACM, 111–124.

[35] Victor S Sheng, Foster Provost, and Panagiotis G Ipeirotis. 2008. Get Another Label? Improving Data Quality and Data Mining using Multiple, Noisy Labelers. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 614–622.

[36] Richard Socher, Eric H Huang, Jeffrey Pennin, Christopher D Manning, and Andrew Y Ng. 2011. Dynamic Pooling and Unfolding Recursive Autoencoders for Paraphrase Detection. In *Advances in Neural Information Processing Systems (NIPS)*. 801–809.

[37] Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. 2011. Generating Parameter Comments and Integrating with Method Summaries. In *International Conference on Program Comprehension (ICPC)*. IEEE, 71–80.

[38] Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research* 15, 1 (2014), 1929–1958.

[39] Ilya Sutskever, Oriol Vinyals, and Quoc VV Le. 2014. Sequence to Sequence Learning with Neural Networks. In *Advances in Neural Information Processing Systems (NIPS)*. 3104–3112.

[40] Bogdan Vasilescu, Casey Casalnuovo, and Premkumar Devanbu. 2017. Recovering Clear, Natural Identifiers from Obfuscated JavaScript Names. In *Joint Meeting on the Foundations of Software Engineering (ESEC/FSE)*. ACM. to appear.

[41] Yushi Wang, Jonathan Berant, and Percy Liang. 2015. Building a Semantic Parser Overnight. In *Annual Meeting of the Association for Computational Linguistics (ACL)*. ACL, 1332–1342.

[42] Yi Wei, Nirupama Chandrasekaran, Sumit Gulwani, and Youssef Hamadi. 2015. *Building Bing Developer Assistant*. Technical Report. MSR-TR-2015-36, Microsoft Research.

[43] Edmund Wong, Taiyue Liu, and Lin Tan. 2015. CloCom: Mining Existing Source Code for Automatic Comment Generation. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 380–389.

[44] Edmund Wong, Jinqiu Yang, and Lin Tan. 2013. AutoComment: Mining question and answer sites for automatic comment generation. In *International Conference on Automated Software Engineering (ASE)*. IEEE, 562–567.

[45] Di Yang, Aftab Hussain, and Cristina Videira Lopes. 2016. From query to usable code: an analysis of Stack Overflow code snippets. In *Working Conference on Mining Software Repositories (MSR)*. ACM, 391–402.

[46] Ziyu Yao, Daniel S. Weld, Wei-Peng Chen, and Huan Sun. 2018. StaQC: A Systematically Mined Question-Code Dataset from Stack Overflow. In *WWW 2018: The 2018 Web Conference*.

[47] Pengcheng Yin and Graham Neubig. 2017. A Syntactic Neural Model for General-Purpose Code Generation. In *Meeting of the Association for Computational Linguistics (ACL)*.

[48] Alexey Zagalsky, Ohad Barzilay, and Amiram Yehudai. 2012. Example overflow: Using Social Media for Code Recommendation. In *International Workshop on Recommendation Systems for Software Engineering (RSSE)*. IEEE Press, 38–42.