Mapping Language to Code in Programmatic Context

Srinivasan Iyer*, **Ioannis Konstas**†, **Alvin Cheung*** and **Luke Zettlemoyer***
*Paul G. Allen School of Computer Science and Engineering, Univ. of Washington, Seattle, WA
{sviyer, akcheung, lsz}@cs.washington.edu
†Heriot-Watt University, Edinburgh, UK
i.konstas@hw.ac.uk

Abstract

Source code is rarely written in isolation. It depends significantly on the programmatic context, such as the class that the code would reside in. To study this phenomenon, we introduce the task of generating class member functions given English documentation and the programmatic context provided by the rest of the class. This task is challenging because the desired code can vary greatly depending on the functionality the class provides (e.g., a sort function may or may not be available when we are asked to "return the smallest element" in a particular member variable list). We introduce CONCODE, a new large dataset with over 100,000 examples consisting of Java classes from online code repositories, and develop a new encoder-decoder architecture that models the interaction between the method documentation and the class environment. We also present a detailed error analysis suggesting that there is significant room for future work on this task.

1 Introduction

Natural language can be used to define complex computations that reuse the functionality of rich, existing code bases. However, existing approaches for automatically mapping natural language (NL) to executable code have considered limited language or code environments. They either assume fixed code templates (i.e., generate only parts of a method with a predefined structure; Quirk et al., 2015), a fixed context (i.e., generate the body of the same method within a single fixed class; Ling et al., 2016), or no context at all (i.e., generate code tokens from the text alone; Oda et al., 2015). In this paper, we introduce new data and methods for learning to map language to source code within the context of a real-world programming environment, with application to generating member functions from documentation for automatically collected Java class environments.

```
public class SimpleVector implements Serializable {
  double[] vecElements;
  double[] weights;

  NL Query: Adds a scalar to this vector in place.
  Code to be generated automatically:
  public void add(final double arg0) {
    for (int i = 0; i < vecElements.length; i++){
      vecElements[i] += arg0;
    }
  }

  NL Query: Increment this vector
  Code to be generated automatically:
  public void inc() {
    this.add(1);
  }
}
```

Figure 1: Code generation based on the class environment and method documentation. The figure shows a class where the programmer wants to automatically generate the `add` method from documentation, assuming the rest of the class is already written. The system needs to understand that `vecElements` is the vector to be augmented, and that the method must take in a scalar parameter as the element to be added. The model also needs to disambiguate between the member variables `vecElements` and `weights`.

The presence of rich context provided by an existing code environment better approximates the way programmers capitalize on code re-use, and also introduces new language understanding challenges. Models must (a) map the NL to environment variables, library API calls and user-defined methods found elsewhere in the class based on their names, types and signatures, and (b) decide on the structure of the resulting code. For example, in Figure 1, to generate the method `inc()` from the corresponding NL, *Increment this vector*, it is crucial to know of the existence of class method `add()`. This helps us decide if it should directly call `add()` or generate the method from scratch by iterating through the `vecElements` array and incrementing each element. Similarly, for generating the `add()` method, the code needs to use the class variable

Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, pages 1643–1652
Brussels, Belgium, October 31 - November 4, 2018. ©2018 Association for Computational Linguistics

`vecElements` correctly. Overall, the code environment provides rich information relating to the intent of the developer, and can be used to considerably reduce ambiguity in the NL documentation.

To learn such a code generator, we use a specialized neural encoder-decoder model that (a) encodes the NL together with representations based on subword units for environment identifiers (member variables, methods) and data types, and (b) decodes the resulting code using an attention mechanism with multiple steps, by first attending to the NL, and then to the variables and methods, thus also learning to copy variables and methods. This two-step attention helps the model to match words in the NL with representations of the identifiers in the environment. Rather than directly generating the output source code tokens (Dong and Lapata, 2016; Iyer et al., 2017), the decoder generates production rules from the grammar of the target programming language similar to Rabinovich et al. (2017), Yin and Neubig (2017), and Krishnamurthy et al. (2017) and therefore, guarantees the syntactic well-formedness of the output.

To train our model, we collect and release CONCODE, a new dataset comprising over 100,000 (class environment, NL, code) tuples by gathering Java files containing method documentation from public Github repositories. This is an order of magnitude larger than existing datasets that map NL to source code for a general purpose language (MTG from Ling et al. (2016) has 13k examples), contains a larger variety of output code templates than existing datasets built for a specific domain, and is the first to condition on the environment of the output code. Also, by design, it contains examples from several domains, thus introducing open-domain challenges of new identifiers during test time (some e.g. class environments are LookupCommand, ColumnFileReader and ImageSequenceWriter). Our model achieves an exact match accuracy of 8.6% and a BLEU score (a metric for partial credit; Papineni et al., 2002) of 22.11, outperforming retrieval and recent neural methods. We also provide an extensive ablative analysis, quantifying the contributions that come from the context and the model, and suggesting that our work opens up various areas for future investigation.

NL query:
Adds a scalar to this vector in place

Variables: [Type, Name]
double[] vecElements
double[] weights

Environment

Methods: [Return Type, Name, ParameterList]
void inc ()
float dotProduct (SimpleVector other)
float multiply(float scalar)

Source code:

```
public void add(final double arg0) {
   for (int i = 0; i < vecElements.length(); i++){
      vecElements[i] += arg0;
   }
}
```

AST Derivation:

```
MemberDeclaration-->MethodDeclaration
MethodDeclaration-->
   TypeTypeOrVoid IdentifierNT FormalParameters
    MethodBody
TypeTypeOrVoid-->void
IdentifierNT-->add
FormalParameters-->( FormalParameterList )
FormalParameterList-->FormalParameter
  ...
Primary-->IdentifierNT
IdentifierNT-->i
Nt_68-->+=
Expression-->Primary
Primary-->IdentifierNT
IdentifierNT-->arg0
```

Figure 2: Our task involves generating the derivation of the source code of a method based on the NL documentation, class member variables (names and data types), and other class member methods (method names and return types), which form the code environment.

2 Task Definition

We introduce the task of generating source code from NL documentation, conditioned on the class environment the code resides in. The environment comprises two lists of entities: (1) class member variable names with their data types (for example, `double[] vecElements` as seen in Figure 2), and (2) member function names together with their return types (for example, `void inc()`).[1] Formally, let $q^{(i)}$, $a^{(i)}$ denote the NL and source code respectively for the ith training example, where $a^{(i)}$ is a sequence of production rules that forms the derivation of its underlying source code. The environment comprises a list of variables names ${v^{(i)}}_{1..|v^{(i)}|}$ and their corresponding types ${t^{(i)}}_{1..|t^{(i)}|}$, as well as method names ${m^{(i)}}_{1..|m^{(i)}|}$ and their return types

[1]The method parameters and body can be used as well but we leave this to future work.

$r^{(i)}{}_{1..|r^{(i)}|}$. Our goal is to generate the derivation of $a^{(i)}$ given $q^{(i)}$ and the environment (see Figure 2).

3 Models

We evaluate a number of encoder-decoder models that generate source code derivations from NL and the class environment. Our best model encodes all environment components broken down into sub-word units (Sennrich et al., 2016) separately, using Bi-LSTMs and decodes these contextual representations to produce a sequence of valid production rules that derive syntactically valid source code. The decoder also uses a two-step attention mechanism to match words in the NL with environment components, and then uses a supervised copy mechanism (Gu et al., 2016a) to incorporate environment elements in the resulting code. We describe this architecture below.

3.1 Encoder

The encoder computes contextual representations of the NL and each component in the environment. Each word of the NL, q_i, is embedded into a high dimensional space using Identifier matrix I (denoted as $\mathbf{q_i}$) followed by the application of a n-layer bidirectional LSTM (Hochreiter and Schmidhuber, 1997). The hidden states of the last layer $(h_1, \cdots, h_z)$ are passed on to the attention layer, while the hidden states at the last token are used to initialize the decoder.

$$h_1, \cdots, h_z = \text{BiLSTM}(\mathbf{q_1}, \ldots, \mathbf{q_z})$$

To encode variables and methods, the variable types (t_i) and method return types (r_i) are embedded using a type matrix T (denoted as $\mathbf{t_i}$ and $\mathbf{r_i}$). To encode the variable and method names (v_i, m_i), they are first split based on camel-casing, and each component is embedded using I, represented as $\mathbf{v_{i1}}, \ldots, \mathbf{v_{ij}}$ and $\mathbf{m_{i1}}, \ldots, \mathbf{m_{ik}}$. The encoded representation of the variable and method names is the final hidden state of the last layer of a Bi-LSTM over these embeddings ($\mathbf{v_i}$ and $\mathbf{m_i}$). Finally, a 2-step Bi-LSTM is executed on the concatenation of the variable type embedding and the variable name encoding. The corresponding hidden states form the final representations of the variable type and the variable name $(\hat{t_i}, \hat{v_i})$ and are passed on to the attention mechanism. Method return types and names are processed identically using the same Bi-LSTMs and embedding matrices $(\hat{r_i}, \hat{m_i})$.

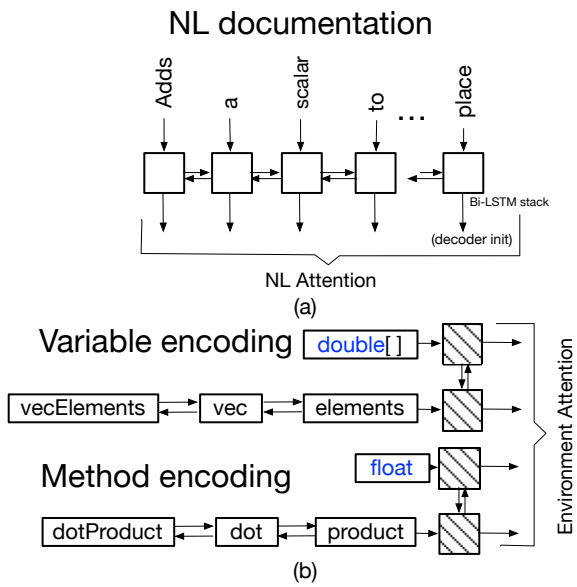

Figure 3: The encoder creates contextual representations of the NL (a), the variables and the methods (b). Variable (method) names are split based on camel-casing and encoded using a BiLSTM. The variable (method) type and name are further contextualized using another BiLSTM.

$$\begin{aligned}
\mathbf{t_i} &= t_i T;\ \mathbf{v_{ij}} = v_{ij} I \\
\mathbf{r_i} &= r_i T;\ \mathbf{m_{ik}} = m_{ik} I \\
\mathbf{v_i} &= \text{BiLSTM}(\mathbf{v_{i1}}, \ldots, \mathbf{v_{ij}}) \\
\mathbf{m_i} &= \text{BiLSTM}(\mathbf{m_{i1}}, \ldots, \mathbf{m_{ik}}) \\
\hat{t_i}, \hat{v_i} &= \text{BiLSTM}(\mathbf{t_i}, \mathbf{v_i}) \\
\hat{r_i}, \hat{m_i} &= \text{BiLSTM}(\mathbf{r_i}, \mathbf{m_i})
\end{aligned}$$

Figure 3 shows an example of the encoder.

3.2 Decoder

We represent the source code to be produced as a sequence of production rules (a_t at step t), with a non-terminal n_t on the left hand side and a combination of terminal and non-terminal symbols on the right hand side (see Figure 2). The first non-terminal is *MemberDeclaration*. Subsequently, every non-terminal is expanded in a depth first left to right fashion, similar to Yin and Neubig (2017). The probability of a source code snippet is decomposed as a product of the conditional probability of generating each step in the sequence of rules conditioned on the previously generated rules. Our decoder is an LSTM-based RNN that produces a context vector c_t at each time step, which is used to compute a distribution over next actions.

$$p(a_t|a_{<t}) \propto exp(W^{n_t} c_t) \tag{1}$$

Here, W^{n_t} is a $|n_t| \times H$ matrix, where $|n_t|$ is the total number of unique production rules that n_t can be expanded to. The context vector c_t is computed using the hidden state s_t of an n-layer decoder LSTM cell and attention vectors over the NL and the context (z_t and e_t), as described below.

Decoder LSTM The decoder uses an n-layer LSTM whose hidden state s_t is computed based on the current non-terminal n_t to be expanded, the previous production rule a_{t-1}, the parent production rule par(n_t) that produced n_t, the previous decoder LSTM state s_{t-1}, and the decoder state of the LSTM cell that produced n_t, denoted as s_{n_t}.

$$s_t = \text{LSTM}(n_t, a_{t-1}, \text{par}(n_t), s_{t-1}, s_{n_t}) \quad (2)$$

We use an embedding matrix N to embed n_t and matrix A to embed a_{t-1} and par(n_t). If a_{t-1} is a rule that generates a terminal node that represents an identifier or literal, it is represented using a special rule *IdentifierOrLiteral* to collapse all these rules into a single previous rule.

Two-step Attention At time step t, the decoder first attends to every token in the NL representation, h_i, using the current decoder state, s_t, to compute a set of attention weights α_t, which are used to combine h_i into an NL context vector z_t. We use a general attention mechanism (Luong et al., 2015), extended to perform multiple steps.

$$\alpha_{t,i} = \frac{\exp(s_t^{\mathrm{T}} \mathbf{F} h_i)}{\sum_i \exp(s_t^{\mathrm{T}} \mathbf{F} h_i)}$$
$$z_t = \sum_i \alpha_{t,i} h_i$$

The context vector z_t is used to attend over every type (return type) and variable (method) name in the environment, to produce attention weights β_t that are used to combine the entire context $x = [t : v : r : m]$ into an environment context vector e_t.[2]

$$\beta_{t,j} = \frac{\exp(z_t^{\mathrm{T}} \mathbf{G} x_j)}{\sum_j \exp(z_t^{\mathrm{T}} \mathbf{G} x_j)}$$
$$e_t = \sum_j \beta_{t,j} x_j$$

Finally, c_t is computed using the decoder state and both context vectors z_t and e_t:

$$c_t = tanh(\hat{W}[s_t : z_t : e_t])$$

[2]":" denotes concatenation.

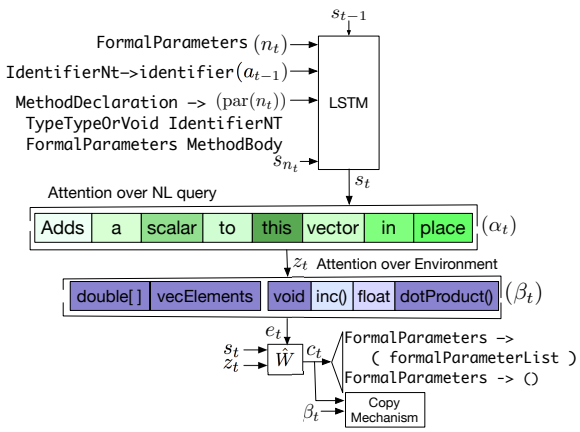

Figure 4: The hidden state s_t of our decoder is a function of the previous hidden state, current non-terminal, previous production rule, parent rule, and the parent hidden state. s_t is used to attend on the NL and compress it into z_t, which is then used to attend over the environment variables and methods to generate e_t. The decoder uses all these context vectors to produce a distribution over valid right hand side values of the current non-terminal, and also learns to copy from the environment.

Supervised Copy Mechanism Since the class environment at test time can belong to previously unseen new domains, our model needs to learn to copy variables and methods into the output. We use the copying technique of Gu et al. (2016a) to compute a copy probability at every time step t using vector b of dimensionality H.

$$\text{copy}(t) = \sigma(b^T c_t)$$

Since we only require named identifiers or user defined types to be copied, both of which are produced by production rules with n_t as *IdentifierNT*, we make use of this copy mechanism only in this case. Identifiers can be generated by directly generating derivation rules (see equation 1), or by copying from the environment. The probability of copying an environment token x_j, is set to be the attention weights $\beta_{t,j}$ computed earlier, which attend exactly on the environment types and names which we wish to be able to copy. The copying process is supervised by preprocessing the production rules to recognize identifiers that can be copied from the environment, and both the generation and the copy probabilities are weighted by $1 - \text{copy}(t)$ and copy(t) respectively. The LSTM decoder with attention mechanism is illustrated in Figure 4.

3.3 Baseline Models

Retrieval We evaluate a retrieval baseline, where the output source code for a test example is the training example source code whose NL is closest

in terms of cosine similarity to the test NL using a tf-idf representation. We then replace all occurrences of environment variables and methods in the chosen training source code with similarly typed variables and methods from the environment of the test example, and we break ties randomly.

Seq2seq We evaluate a Seq2Seq baseline by representing the NL and context as a sequence formed by the concatenation of the NL, the variables and the methods with separators between them. The variables (methods) are represented with the type (return type) followed by the name, with a different separator between them. The encoder is an n-layer LSTM which initializes an LSTM-based decoder using its final hidden states. The decoder uses an attention mechanism (Luong et al., 2015) over the encoder states to produce a conditional distribution over the next source code token (not production rule) given all the previous tokens. We replace UNK tokens in the output with source tokens having the most attention weight. We also attempted to evaluate the Seq2Tree model of Dong and Lapata (2016) but the redundancy in the model resulted in extremely long output sequences which did not scale. Experiments on a smaller dataset gave comparable results to Seq2seq.

Seq2prod This baseline corresponds to the action sequence model by Yin and Neubig (2017), with a BiLSTM over a sequence representation of the NL and context (same as Seq2seq), and a decoder that learns to generate a derivation of the AST of the underlying source code, similar to our model. The decoder uses the same attention mechanism as the Seq2seq, however, it uses supervised copying from the entire input sequence to handle unknown words encountered during testing.

4 CONCODE

We built a new dataset (CONCODE) from public Java projects on Github that contains environment information together with NL (Javadoc-style method comments) and code tuples. We gather Java files from ∼33,000 repositories, which are then split into train, development, and test sets based on repository, rather than purely randomly. Dividing based on the repository keeps the domains in the test set separate from the training set, therefore providing near zero-shot conditions that should truly test the ability of models to generalize to associate unseen NL tokens with previously unseen environment variables and methods. We also remove all examples from the development and test sets where the NL is exactly present in the training set. We further eliminate all Java classes that inherit from parent classes, since the resulting code may use variables and methods inherited from parent classes that reside in separate source files. While this is an important and interesting feature, we leave it for future work.

	Count
Train	100,000
Valid/Test	2,000/2,000
Average NL Length	13.73
Average Code Characters	119
Average Code Tokens	26.3
Average # Environment Variables	4.89
Average # Environment Methods	10.95
Average AST Nodes	79.6
# Node Types	153
# Production Rules	342*
% Getters	16.74%
% Setters	3.39%
% using Class Variables	68%
% using Class Methods	16.2%
% Unknown Types	7.65%

Table 1: Statistics of our dataset of (NL, context and code) tuples collected from Github. *Does not include rules that generate identifiers.

Every method that contains a Javadoc comment is treated as a training example. The Javadoc comment forms the NL, the method body is the target code to be generated, and all member variables, as well as other member method signatures are extracted and stored as part of the context. The Javadoc is preprocessed by stripping special symbols such as `@link`, `@code`, `@inheritDoc` and special fields such as `@params`. Methods that do not parse are eliminated and the rest are pre-processed by renaming locally defined variables canonically, beginning at *loc0* and similarly for arguments, starting with *arg0*. We also replace all method names with the word *function* since it doesn't affect the semantics of the resulting program. Generating informative method names has been studied by Allamanis et al. (2015a). We replace all string literals in the code with constants as they are often debug messages. Finally, we use an ANTLR java gram-

mar[3] that is post-processed by adding additional non-terminals and rules to eliminate wildcard symbols in the grammar, in order to convert the source code into a sequence of production rules. The resulting dataset contains 100,000 examples for training, and 2000 examples each for development and testing, respectively. Table 1 summarizes the various statistics. We observe that on average, an environment contains ∼5 variables and ∼11 methods. Around 68% of the target code uses class member variables, and 16% of them use member methods, from the environment. Based on a frequency cutoff of 2 on the training set, we find that 7% of the types in the development set code are unknown, hence they need to be copied from the environment. Since CONCODE is extracted from a diverse set of online repositories, there is a high variety of code templates in the dataset compared to existing datasets. For example, a random baseline on the Hearthstone card game dataset (Ling et al., 2016) gives a BLEU score of 40.3, but only a score of 10.2 on CONCODE. We plan to release all code and data from this work.[4]

5 Experimental Setup

We restrict all models to examples where the length of the combination of the NL and the context is at most 200 tokens and the length of the output source code is at most 150 tokens. Source NL tokens are lower-cased, camel-cased identifiers are split and lower-cased, and are used together with the original version. The vocabulary for identifiers uses a frequency threshold of 7, resulting in a vocabulary of $32,600$ tokens. The types vocabulary uses a threshold of 2 resulting in $22,324$ types. We include all 153 non-terminals and 342 previous rules. We use a threshold of 2 for output production rules to filter out the long tail of rules creating identifiers and literals, resulting in $18,135$ output rules. Remaining values are replaced with the UNK symbol.

Hyperparameters We use an embedding size H of 1024 for identifiers and types. All LSTM cells use 2-layers and a hidden dimensionality of 1024 (512 on each direction for BiLSTMs). We use an embedding size of 512 for encoding non-terminals and rules in the decoder. We use dropout with $p = 0.5$ in between LSTM layers and at the output of the decoder over c_t. We train our model for 30 epochs using mini-batch gradient descent with a batch size of 20, and we use Adam (Kingma and Ba, 2015) with an initial learning rate of 0.001 for optimization. We decay our learning rate by 80% based on performance on the development set after every epoch.

[3] https://github.com/antlr/grammars-v4

[4] https://github.com/sriniiyer/concode

Model	Exact	BLEU
Retrieval	2.25 (1.65)	20.27 (18.15)
Seq2Seq	3.2 (2.9)	23.51 (21.0)
Seq2Prod	6.65 (5.55)	21.29 (20.55)
Ours	8.6 (7.05)	22.11 (21.28)

Table 2: Exact match accuracy and BLEU score (for partial credit) on the test (development) set, comprising 2000 examples from previously unseen repositories.

Model	Exact	BLEU
Ours	7.05	21.28
-Variables	1.6	20.78
-Methods	6.2	21.74
-Two step attention	5.75	17.2
-Camel-case encoding	5.7	21.83

Table 3: Ablation of model features on the development set.

Inference and Metrics Inference is done by first encoding the NL and context of the test example. We maintain a stack of symbols starting with the non-terminal, *MemberDeclaration*, and at each step, a non-terminal (terminals are added to the output) is popped off the stack to run a decoding step to generate the next set of symbols to push onto the stack. The set of terminals generated along the way forms the output source code. We use beam search and maintain a ranked list of partial derivations (or code tokens for Seq2seq) at every step to explore alternate high-probability derivations. We use a beam size of 3 for all neural models. We copy over source tokens whenever preferred by the model output. We restrict the output to 150 tokens or 500 production rules.

To evaluate the quality of the output, we use Exact match accuracy between the reference and generated code. As a measure of partial credit, we also compute the BLEU score (Papineni et al., 2002), following recent work on code generation (Ling et al., 2016; Yin and Neubig, 2017). BLEU is an n-gram precision-based metric that will be higher when more subparts of the predicted code match the provided reference.

NL: Gets the value of the tags property. This accessor method returns a reference to the live list, not a snapshot.

Variables:
```
String validationPattern;
List<String> tags;
```
Methods:
```
String getValidationPattern
void setValidationPattern
```
Code:
```
List <String> function() {
  if ( tags == null ) {
    tags = new ArrayList <String>();}
  return this.tags; }
```
(a)

NL: Convert mixed case to underscores.

Variables:
```
NamingStrategy INSTANCE;
```
Methods:
```
String classToTableName
String collectionTableName
String tableName
String columnName
String addUnderscores
```
Code:
```
String function (String arg0) {
   return addUnderscores (arg0);
}
```
(b)

NL: Skips the next char

Variables:
```
String str;
int pos
```
Methods:
```
char ch()
int pos()
int length()
int gatherInt()
boolean hasNext()
```
Code:
```
void function () {
  pos ++ ;
}
```
(c)

NL: Returns the execution data store with data for all loaded classes.

Variables:
```
SessionInfoStore sessionInfos
ExecutionDataStore executionData
```
Methods:
```
void load
SessionInfoStore getSessionInfoStore
void save
```
Code:
```
ExecutionDataStore function () {
  return executionData ;
}
```
(d)

NL: Return a library that loads the core from code.jquery.com

Prediction:
```
JQueryLibrary function
   (String arg0) {
  return new JQueryLibrary ( arg0 ) ;
}
```
Reference:
```
JQueryLibrary function(String arg0) {
  return new JQueryLibrary (
      BASE_RESOURCE_URL + "string" +
        arg0 + "string" ) ;
}
```
(e)

NL: Empty the violations list

Variables:
```
long numPptEntries
List<Violation> violations
```
Prediction:
```
void function() {
  violations.clear() ;
}
```
Reference:
```
void function () {
 violations =
  new ArrayList<Violation>();
}
```
(f)

NL: Clear the update -timestamps data.

Variables:
```
boolean DEBUG_ENABLED
String REGION_NAME
TimestampsRegion region
```
Prediction:
```
void function() {
  region.clear() ;
}
```
Reference:
```
void function() {
  region.evictAll();
}
```
(h)

NL: Returns true if this registry maps one or more keys to the specified value.

Variables:
```
Map _values
Map _register
```
Methods:
```
IWidgetIdentifier get
WidgetLocator add
```
Prediction:
```
boolean function
  (IWidgetIdentifier arg0) {
  return _values.contains(arg0);
}
```
Reference:
```
boolean function
  (IWidgetIdentifier arg0) {
  return _register.containsValue(arg0);
}
```
(i)

Prediction:
```
List < ? > function ( ) {
  return iTransformers ; }
```
Reference:
```
Transformer [ ] function ( ) {
  return iTransformers ; }
```
(g)

Figure 5: Analysis of our model output on development set examples. Some environment variables and methods are omitted for space. (a)-(d) represent cases where the model exactly produced the reference output. (e)-(g) are cases where the model output is very reasonable for a practical setting. In (f), the model produces a better solution than the reference. In (h), the context lacks information to produce the reference code. The model chooses the wrong element in (i) and could be improved by better encoder representations.

6 Results

We present results for the context based code generation task on the test and dev sets in Table 2. Our model outperforms all baselines and achieves a gain of 1.95% exact match accuracy and 0.82 BLEU points, with respect to the next best models. The combination of independently encoding subword units and applying a two-step attention mechanism helps the model learn to better associate the correct variables/methods from the context and the language in the NL. Figure 5 (a) shows an example output of our model, which produces code structure intermixed with member variables (tags). In (b) our model learns to call method `addUnderscores` (an UNK in the vocabulary) with its correct return type (`String`). Similarly, in (d) our model also successfully learns to use a previously unseen type (`ExecutionDataStore`) when making use of the corresponding variable. (c) is an example of where the NL does not directly refer to the variable to be used. The mismatch between dev and test results is because we ensure that the dev and test examples come from non-overlapping Github repositories, resulting in different distributions.

Using a constrained decoder that generates syntax tree rules instead of tokens leads to significant gains in terms of exact match score (6.65 for Seq2prod vs 3.2 for Seq2seq), and shows that this is an important component of code generation systems. Seq2prod, however, fails on examples (b)-(d), since it is harder to learn to match the NL tokens with environment elements. Finally, all neural models outperform the retrieval baseline with member substitution.

To understand which components of the data and the model contribute most to the output, we perform an ablation study on the development set (Table 3). Removing the variables leads to a significant hit in exact match accuracy since 68% of examples (Table 1) refers to class variables; a similar but smaller reduction is incurred by removing methods. The presence of these components makes this task more challenging and also more aligned with programming scenarios in practice. Removing the two-step attention mechanism leads to a 1.3% drop in accuracy since the attention on the NL is unable to interact with the attention on the environment variables/methods. Removing camelcase encoding leads to a small drop mainly because many variable (method) names are single words.

Category	Fraction
Totally Wrong	62%
Marginally Correct	9%
Mostly Correct	16%
Exact Match	11%
Semantically Equivalent	2%

Table 4: Qualitative distribution of errors on the development set.

7 Error Analysis

Subfigures 5(e)-(i) show cases where our model output did not exactly match the reference. In (e)-(g), the model output is semantically equivalent to the reference and is a very reasonable prediction in a practical setting. For example, in (e) the only difference between the prediction and the reference is the string concatenations to the url. Interestingly, in example (f) the prediction is a cleaner way to achieve the same effect as the reference, and this is a great example of the application of these models for suggesting standard and efficient code. Unfortunately, our model is penalized by the exact match metric here. Similarly, in (g), the model uses a generic list (`List<?>`) in place of the specific type (`Transformer[]`). Example (h) demonstrates a case where the model is unaware of methods that can be called on class members (specifically that `evictAll` is a member of the `TimestampsRegion` class), and requires augmenting the environment with additional member type documentation, which we believe will be an important area for future work. Example (i) calls for richer encoder representations, since our model incorrectly uses the `_values` variable instead of `_register`, as it is unable to associate the word "registry" with the right elements.

We further perform a qualitative analysis of 100 predictions on our development set (Table 4) and find that there is significant room for improvement with 71% of the predictions differing significantly from their references. 16% of the predictions are very close to their references with the difference being 1-2 tokens, while 11% are exactly correct. 2% of the predictions were semantically equivalent but not exactly equal to their references.

8 Related Work

There is significant existing research on mapping NL directly to executable programs in the form of logical forms (Zettlemoyer and Collins, 2005), λ-DCS (Liang et al., 2013), regular expressions (Kushman and Barzilay, 2013; Locascio et al., 2016), database queries (Iyer et al., 2017; Zhong et al., 2017) and general purpose programs (Balog et al., 2016; Allamanis et al., 2015b). Ling et al. (2016) generate Java and Python source code from NL for card games, conditioned on categorical card attributes. Manshadi et al. (2013) generate code based on input/output examples for applications such as database querying. Gu et al. (2016b) use neural models to map NL queries to a sequence of API calls, and Neelakantan et al. (2015) augment neural models with a small set of basic arithmetic and logic operations to generate more meaningful programs. In this work, we introduce a new task of generating programs from NL based on the environment in which the generated code resides, following the frequently occurring pattern in large repositories where the code depends on the types and availability of variables and methods in the environment.

Neural encoder-decoder models have proved effective in mapping NL to logical forms and also for directly producing general purpose programs. Ling et al. (2016) use a sequence-to-sequence model with attention and a copy mechanism to generate source code. Instead of directly generating a sequence of code tokens, recent methods focus on constrained decoding mechanisms to generate syntactically correct output using a decoder that is either grammar-aware or has a dynamically-determined modular structure paralleling the structure of the abstract syntax tree (AST) of the code (Dong and Lapata, 2016; Rabinovich et al., 2017; Krishnamurthy et al., 2017; Yin and Neubig, 2017). Our model also uses a grammar-aware decoder similar to Yin and Neubig (2017) to generate syntactically valid parse trees, augmented with a two-step attention mechanism (Chen et al., 2016), followed by a supervised copying mechanism (Gu et al., 2016a) over the class environment.

Recent models for mapping NL to code have been evaluated on datasets containing highly templated code for card games (Hearthstone & MTG; Ling et al., 2016), or manually labeled per-line comments (DJANGO; Oda et al., 2015). These datasets contain $\sim$20,000 programs with short textual descriptions possibly paired with categorical data, whose values need to be copied onto the resulting code from a single domain. In this work, we collect a new dataset of over 100,000 NL and code

pairs, together with the corresponding class environment. Each environment and NL describe a specific domain and the dataset comprises thousands of different domains, that poses additional challenges. Having an order of magnitude more data than existing datasets makes training deep neural models very effective, as we saw in the experimental evaluation. While massive amounts of Github code have been used before for creating datasets on source code only (Allamanis and Sutton, 2013, 2014; Allamanis et al., 2016), we instead extract from Github a dataset of NL and code with an emphasis on context, in order to learn to map NL to code within a class.

9 Conclusion

In this paper, we introduce new data and methods for learning to generate source code from language within the context of a real-world code base. To train models for this task, we collect and release CONCODE, a large new dataset of NL, code, and context tuples from online repositories, featuring code from a variety of domains. We also introduced a new encoder decoder model with a specialized context encoder which outperforms strong neural baselines by 1.95% exact match accuracy. Finally, analysis suggests that even richer models of programmatic context could further improve these results.

Acknowledgements

The research was supported in part by DARPA (FA8750-16-2-0032), the ARO (ARO-W911NF-16-1-0121), the NSF (IIS-1252835, IIS-1562364, IIS1546083, IIS-1651489, OAC-1739419), the DOE (DE-SC0016260), the Intel-NSF CAPA center, and gifts from Adobe, Amazon, Google, and Huawei. The authors thank the anonymous reviewers for their helpful comments.

References

Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2015a. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 38–49.

Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning*, pages 2091–2100.

Miltiadis Allamanis and Charles Sutton. 2013. Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 207–216. IEEE Press.

Miltiadis Allamanis and Charles Sutton. 2014. Mining idioms from source code. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 472–483. ACM.

Miltiadis Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. 2015b. Bimodal modelling of source code and natural language. In *Proceedings of The 32nd International Conference on Machine Learning*, pages 2123–2132.

Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*.

Danqi Chen, Jason Bolton, and Christopher D. Manning. 2016. A thorough examination of the CNN/-Daily Mail reading comprehension task. In *Association for Computational Linguistics (ACL)*.

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.

Jiatao Gu, Zhengdong Lu, Hang Li, and Victor O.K. Li. 2016a. Incorporating copying mechanism in sequence-to-sequence learning. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1631–1640, Berlin, Germany. Association for Computational Linguistics.

Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016b. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 631–642, New York, NY, USA. ACM.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation*, 9(8):1735–1780.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. 2017. Learning a neural semantic parser from user feedback. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 963–973, Vancouver, Canada. Association for Computational Linguistics.

Diederik Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *ICLR*.

Jayant Krishnamurthy, Pradeep Dasigi, and Matt Gardner. 2017. Neural semantic parsing with type constraints for semi-structured tables. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1516–1526, Copenhagen, Denmark. Association for Computational Linguistics.

Nate Kushman and Regina Barzilay. 2013. Using semantic unification to generate regular expressions from natural language. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 826–836. Association for Computational Linguistics.

Percy Liang, I. Michael Jordan, and Dan Klein. 2013. Learning dependency-based compositional semantics. *Computational Linguistics*, 39(2).

Wang Ling, Phil Blunsom, Edward Grefenstette, Moritz Karl Hermann, Tomáš Kočiský, Fumin Wang, and Andrew Senior. 2016. Latent predictor networks for code generation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 599–609. Association for Computational Linguistics.

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 *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1918–1923, Austin, Texas. Association for Computational Linguistics.

Thang Luong, Hieu Pham, and D. Christopher Manning. 2015. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1412–1421. Association for Computational Linguistics.

Mehdi Hafezi Manshadi, Daniel Gildea, and James F Allen. 2013. Integrating programming by example and natural language programming. In *AAAI*.

Arvind Neelakantan, Quoc V Le, and Ilya Sutskever. 2015. Neural programmer: Inducing latent programs with gradient descent. *arXiv preprint arXiv:1511.04834*.

Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 574–584. IEEE.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318.

Chris Quirk, Raymond Mooney, and Michel Galley. 2015. Language to code: Learning semantic parsers for if-this-then-that recipes. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 878–888, Beijing, China. Association for Computational Linguistics.

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)*, pages 1139–1149, Vancouver, Canada. Association for Computational Linguistics.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany. Association for Computational Linguistics.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 440–450, Vancouver, Canada. Association for Computational Linguistics.

Luke S. Zettlemoyer and Michael Collins. 2005. Learning to map sentences to logical form: structured classification with probabilistic categorial grammars. In *UAI '05, Proceedings of the 21st Conference in Uncertainty in Artificial Intelligence*.

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