# NL2code

Sai Krishna Karanam karanam.s@husky.neu.edu
Nischal Mahaveer Chand mahaveerchand.n@husky.neu.edu
Varun Sundar Rabindranath rabindranath.v@husky.neu.edu

## I. Introduction

### A. Importance of NL2code

Recent Integrated Development Environments (IDEs) provide a feature called "context-aware code completion" which helps programmers speed up the coding process. This is very helpful for programmers who are already aware of the programming language syntax and the design paradigms. But, for beginners, this can be time consuming.

The first step, for beginners, is to search how to implement the code in natural language. Due to the vast amount of resources available online, many unreliable or outdated, having a NL to code generator will make programming more enjoyable for beginners and more productive for veteran programmers. With the introduction of a new language almost everyday, it has become more and more important to establish an interface between NL and code.

We aim to create neural based models to generate code from NL. Below we describe several attempts of implemening various encoder-decoder models for the same. It should be noted that the project does not aim at providing a framework to build industry standard deployable code, but is rather a proof-of-concept. The scope of this project is limited to generate an accurate single line program statement given user intent in the form of a comment or description.

Examples

| | |
|---|---|
| input | call the function sorted with argument x |
| output | sorted(x) |
| input | for every k in keys |
| output | for k in keys: |

## II. Related Word

Techniques to generate regular expressions [1], input parsers [2], UML diagrams, object oriented class layout, and general purpose code from Natural Language (NL) specification have been researched for a long time.

Recent approaches in converting NL to executable source code predominantly use neural network techniques. These approaches follow two basic patterns. One, is to directly convert NL to source code by modeling the task as a Sequence-to-Sequence conversion task, using Recurrent Neural Networks (RNN) (Lili Mou et al., 2015, Ling et al., 2016) [3], [4] to that end. The other is to generate Abstract Syntax Trees (ASTs) from NL input, then deterministically convert the produced AST to code (Pengcheng Yin et al., 2017) [5].

However, research [5] shows that the second approach is better than the first and emphasizes the use of syntax information of the target language, in NL to code tasks. We consider the recent research paper by Pengcheng Yin et. al. [5] , on syntactic neural model for code generation as the primary motivation for this project. [5] proposes an approach where they translate NL to an Abstract Syntax Tree (AST) first, using the grammar of the target language as a prior knowledge. They report a 10% absolute improvement in accuracy compared to the previous state-of-the-art on standard datasets. Since, converting from AST to code can be done deterministically, their system always produces syntactically correct, executable code. This aspect is lacking in the Sequence-to-Sequence models, where the correctness of the generated code is not guaranteed. They use a Bidirectional LSTM (BiLSTM) network for encoding each word of the NL input as a context specific embedding, and an RNN as a decoder to generate an output AST.

Maxim Robinovich et. al. [6], propose an approach similar to [5] for converting NL to AST. They confirm the experimental results of [5], thus again emphasizing the use of target language syntax. However Maxim Robinovich et. al., have evaluated their system on a wider range of datasets achieving near state-of-the-art results on average.

Pengcheg Yin et.al [5], report that 25% of the errors incurred by their system, on one of the datasets, is due to the generated code only partially implementing the required functionality. We hypothesize that this could be due to the standard datasets used for this task being noisy. We propose a denoising procedure, explained in the Data Preprocessing section to clean the dataset, thereby making the datasets more generic.

Barone et. al, 2017 [7], hereafter referred to as the "Edinburgh paper", explores the difficulty of the code generation task and critiques the currently available datasets for this task. They also comment that the effectiveness reported by [5] should not be taken at face value.

In the Edinburgh paper, Barone et al. scraped online open source Python repositories to extract "docstring-code" pairs. They propose that such data accurately represents the query/code pairs found in production environment. They used a Neural Machine Translation approach to translate docstring to code and reported a baseline BLEU score of 10.24, While [5] reports a BLEU score of 84.5 on the Django dataset.

We wanted to check how [5] performs on a similar dataset. We scraped query-code pairs from the same online open source Python repositories as [7]. [5], on our scraped

dataset reports a BLEU score of only 14.2 which reaffirms the [7]'s claim that the reported effectiveness of various models could not be taken at face value and that the task of code generation is more difficult than what is conceived by the community. Much of this misrepresentation could be attributed to the fact that the datasets currently available for this task are very limited and noisy.

## III. Dataset

### A. Standard Datasets

There are 3 datasets that have been used by the research community for the task of code-generation from the comments.

1) Django: Django dataset is a collection of 18000 lines of python code from the open source web framework, Django. The 18000 code statements have been manually annotated with a Natural language description. As a result of manual annotations, all the annotations are artificially descriptive of the corresponding code. This is the main drawback of the dataset, as the manual annotations rarely reflects the comments that programmers write in production environment. This can be seen in the following example.

```
comment:
  raise and exception InvalidCacheBackendError
  with string "Could not find config for '\%s'
  in settings.CACHES" as argument , replace '
  \%s ' with alias .

code:
  raise InvalidCacheBackendError ( "Could not
  find config for '\%s' in settings.CACHES" \%
  alias )
```

2) HearthStone: HearthStone dataset is a collection of 600 Python class - description pairs that implement cards for the online multiplayer card game HearthStone. The class - description pairs used to train and test are very similar, in that any model trained on this dataset would be able to generate only a single template of code, like the one presented in the example. We could not use this dataset as the scope of the dataset is very narrow and far from the case in general production environment.

```
card details:
  name: ['D', 'i', 'r', 'e', ' ',
         'W', 'o', 'l', 'f', ' ',
         'A', 'l', 'p', 'h', 'a']
  cost: ['2']
  type: ['Minion']
  rarity: ['Common']
  race: ['Beast']
  class: ['Neutral']
  description: ['Adjacent', 'minions', 'have',
        '+', '1', 'Attack', '.']
  health: ['2']
  attack: ['2']
  durability: ['-1']

correspoding code:
  class DireWolfAlpha(MinionCard):
    def __init__(self):
      super().__init__( "Dire Wolf Alpha", 2,
        CHARACTER_CLASS.ALL, CARD_RARITY.COMMON,
        minion_type=MINION_TYPE.BEAST)
    def create_minion(self, player):
      return Minion(2, 2, auras=[Aura(ChangeAttack(1),
                          MinionSelector(Adjacent()))])
```

3) Edinburgh: The Edinburgh dataset is a collection of 200k docstring-code pairs scraped from Opensource python repositories. The docstring-code pairs actually reflect the comments written in production environment. However, the docstring reflects what a particular entity (method, class, function etc.) represents rather than how it is implemented. For the task of generating source code, we need information on implemention logic thus making the dataset unusable for our task.

```
comment:
  Process elementNode by svgReader.

code:
  def processSVGElementtext(elementNode, svgReader):
    if svgReader.yAxisPointingUpward:
      return
    fontFamily = getStyleValue('Gentium Basic Regular',
        elementNode, 'font-family')
    fontSize = getRightStripAlphabetPercent(getStyleValue
        ('12.0', elementNode, 'font-size'))
    matrixSVG = getChainMatrixSVGIfNecessary(elementNode,
        svgReader.yAxisPointingUpward)
    loopLayer = svgReader.getLoopLayer()
    translate = euclidean.getComplexDefaultByDictionaryKeys(
        complex(), elementNode.attributes, 'x', 'y')
    for textComplexLoop in getTextComplexLoops(fontFamily,
      fontSize, elementNode.getTextContent(),
      svgReader.yAxisPointingUpward):
      translatedLoop = []
      for textComplexPoint in textComplexLoop:
        translatedLoop.append(textComplexPoint + translate)
      loopLayer.loops.append(matrixSVG.
        getTransformedPath(translatedLoop))
```

### B. Our dataset

1) Scrapped Dataset: We scraped several of the Opensource python repositories from github and collected 12.5k pairs of code-comments after preprocessing the data. We found that the code is very project specific and the descriptions are context specific. This dataset also had the same issue as the Edinburgh dataset as in the descriptions mainly focused on why a code is written rather than how it's written.

2) Processed Django: Among all the datasets explored, the Django dataset represents the problem of converting NL to source code more accurately. But, as mentioned above the Django dataset is very noisy, hence we denoise it (techniques mentioned in data preprocessing) before using it.

| Dataset Statistics | |
|---|---|
| Maximum comment length | 133 |
| Maximum code length | 701 |
| Average comment length | 16.0 |
| Average code length | 8.7 |
| Vocabulary size - Query | 2205 |
| Vocabulary size - Code | 814 |
| Vocabulary size - POS | 44 |
| Vocabulary size - Phrase | 14 |

## IV. Date Preprocessing

In order to add syntax information to the query and make the existing Django dataset as generic as possible, we carried out three preprocessing steps, described below.

### A. POS tagging

The motivation behind POS tagging the queries/comments is that we believe it would provide invaluable information for the code generator.

| Noun/Symbol in Query - Entity in Code | |
|---|---|
| Query | Sum a and b // a and b are Nouns |
| Code | a + b |
| Verbs in Query - Strong indicator of user intent | |
| Query | Sum a and b // Sum is a verb |
| Code | a + b // Indicates add, binary operator |
| Prepositions in Query - Mirrors some keywords | |
| Query | if "a" is present in the Alphabets list |
| Code | if "a" in Alphabets: |

In all the above mentioned examples there is a strong correlation between the POS information in the query and the code generated. Perhaps the most interesting is the correlation between the prepositions and the python language itself, where, as mentioned in the above, the prepositions 'if' and 'in' are actually keywords in Python. We use NLTK with Stanford parser to extract POS information from the input query.

### B. Phrase Identification

We propose that using phrase chunk information would also be invaluable for the code generator.

| | |
|---|---|
| Query | "Add a and b, store results in c" // "in c" is a prepositional phrase |
| Code | c = a + b // "in c" indicates a target of the assignment |
| Query | "Call the function get_data" // Verb phrase |
| Code | get_data() // entity in verb phrase; could help pointer networks |

The above examples explain that the code generator model can leverage the information that a prepositional phrase may contain entities that are the target of an assignment statement and a verb phrase may contain entities that are function calls.

### C. Type Simulation

Python is dynamically typed. That is the runtime determines the types of variables dynamically when the program is executed. The Django dataset in its base form contains a lot of project specific references, which makes the dataset very noisy. We hypothesize that replacing the types of each variable/entity referenced in the comment by its corresponding type would make the dataset very generic.

| Query | |
|---|---|
| Before | ___all___ is a list containing 4 elements : 'get_cache' , 'cache' , 'DEFAULT_CACHE_ALIAS' , 'InvalidCacheBackendError' |
| After | ANY_0 is an list containing 4 elements : STR_1 , STR_2 , STR_0 , STR_4 |
| Code | |
| Before | ___all___ = ['get_cache', 'cache', 'DEFAULT_CACHE_ALIAS', 'InvalidCacheBackendError'] |
| After | ANY_0 = [ 'STR_1', 'STR_2', 'STR_0', 'STR_4' ] |

From the above example, one could immediately see that, all query/code pairs following the "create a list of 4 elements" paradigm will be converted to a generic form as suggested in the example.

To simulate the types of variables/entities, we used the widely-used python module "ast". Using this module we converted the code, in the input query/code pairs to Abstract Syntax Tree (AST) and used a bunch of hand-written rules to identify the types of variables/entities. A few hand-written rules that we used are elaborated in the following figure.

- Variable that is indexed by a string, is mostly a DICT - d['key'] = 'value'
- Variable that is the "name" of a "Call" AST node is a FUNCTION
- Variable that is the "name" of a "Attribute" AST node is a CLASS
- Variable that is the "attr" of a "Attribute" AST node is either a CLASS/FUNCTION/NUM/STR
- Anything that appears within quotes in the AST/-Query is a STR
- ...

## V. Models

We describe four models, each is built upon Pengcheng's model, but encodes the input in a different method. First we start by briefly describing Pengcheng's model.

### A. Pengcheng's model

Pengcheng recognized that adding syntax information to the model would give better result. Their model follows an encoder-decoder architecture, and takes the raw comment as input and generates an AST of the corresponding code as output.

The encoder comprises of an embedding layer and a BiLSTM layer. It takes a comment as input, embeds each word in the comment to give token embeddings $TE_t$, for each word $t$ in the comment. Each $TE_t$ is sequentially fed into the BiLSTM layer, to produces a Query Embedding (QE) of 128 dimensions. QE is passed to the decoder module.

The decoder uses an Recurrent Neural Network (RNN) to sequentially generate each node of the AST. Each node

maps to a timestep in the RNN decoding process and thus, generating the AST can be interpreted as unrolling the RNN. The RNN also maintains an internal state to track the generation process at each timestep.

Pengcheng tackles this problem with a probabilistic grammar model of generating an AST $y$ given NL description $x$: $p(y|x)$. The best possible AST $\hat{y}$ is given by:

$$\hat{y} = \arg\max_{y} p(y|x) \tag{1}$$

$\hat{y}$ is then deterministically converted to the corresponding Python code.

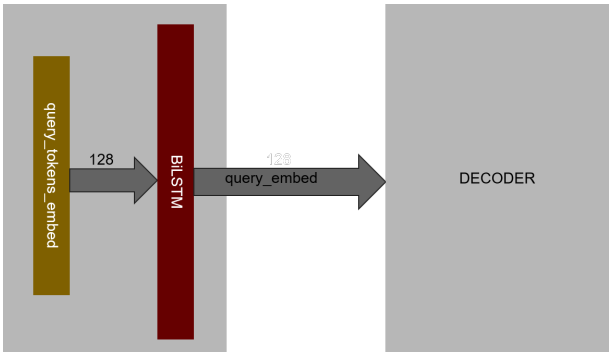


Figure 1: Architecture of Pengcheng's model. The encoder embeds each token of the NL input to give $TE_t$, which is then passed into the BiLSTM layer. The encoder produces a 128 dimension QE which is fed into the decoder.

## B. Our models

As described, the decoder is already state-of-the-art, and needs no further modifications. All models described here-on use the same decoder architecture as Pengcheng's model with modifications only to the encoder and the input data. All models are trained and tested using the dataset described in Processed Django.

1) Basic Concat (BC): For our first attempt to incorporate syntax information into the encoder, we decided to concatenate (shown as ":") the POS ID and phrase ID of each token to the corresponding token embedding, giving the Augmented Token Embedding (ATE). The ATE is then fed into a modified BiLSTM layer that takes 130 dimension embeddings, rather than the default 128 dimensions.

$TE_t$ dimension: 128
$POS\_ID_t$ and $Phrase\_ID_t$ dimension: 1 each; total 2
$ATE_t$ dimension = 130

$ATE_t = [TE_t : POS\_ID_t : Phrase\_ID_t]$
QE = BiLSTM(ATE)

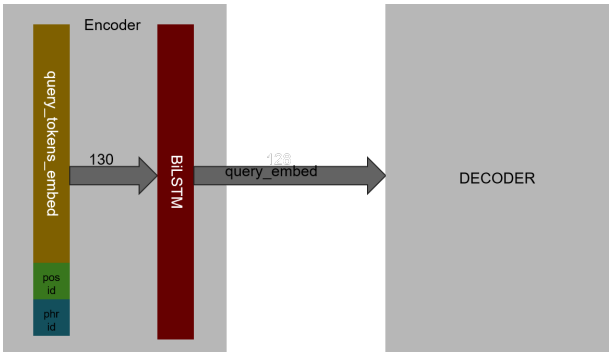


Figure 2: Architecture of Basic Concat model. Notice that the POS ID and phrase ID are directly appended to the corresponding TE. The resulting 130 dimension embedding is passed to the BiLSTM.

2) Linear Projection (LP): To add syntactic information over a sequence of tokens, we used an embedding layer for POS and phrase tags. The resulting TE, POS embedding (POSE), and Phrase embedding (PhE) are then concatenated to produce the ATE; which is a $(128 * 3)$ dimension vector. We then apply a linear projection (using a dense layer with the linear activation function) to give ATE Projected (ATEP) which is passed to BiLSTM.

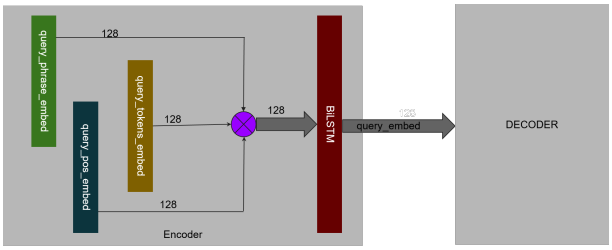


Figure 3: Architecture of Linear Projection model. Each embedding - TE, POSE, and PhE are concatenated and projected (shown as purple circles). The resulting ATEP is passed to the BiLSTM.

$TE_t$ dimension: 128
$POSE_t$ dimension: 128
$PhE_t$ dimension: 128
Dense = (128 * 3) input nodes, 128 output nodes

$ATE_t = [TE_t : POSE_t : PhE_t]$
$ATEP_t = Dense(ATE_t)$
QE = BiLSTM(ATEP)

3) Linear Projection Reduced Dimension (LPrd): Subsequently, we noticed that the POS and Phrase vocabulary sizes were relatively smaller than token vocabulary size. Thus, we reduce the embedding dimensions of POSE and PhE to 8 and 32 respectively. The process described in LP is then repeated.

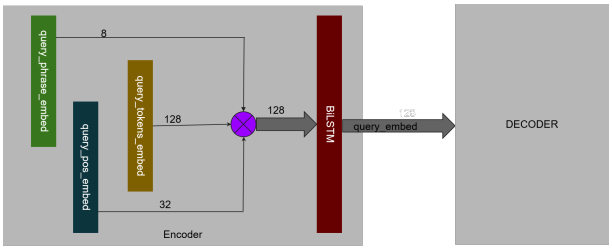


Figure 4: Architecture of Linear Projection Reduced Dimension model. Same as LP, but with reduced dimensions for POSE and PhE.

$TE_t$ dimension: 128
$POSE_t$ dimension: 8
$PhE_t$ dimension: 32
Dense = (128 + 8 + 32) input nodes, 128 output nodes

$ATE_t = [TE_t : POSE_t : PhE_t]$
$ATEP_t = \text{Dense}(ATE_t)$
QE = BiLSTM(ATEP)

4) Raw Query Independent Preprojection (AdvLP): Rather than applying one linear projection on ATE, we apply two here, where the first is independent of the input query. POSE and PhE are concatenated and projected to create Augmentation Embedding (AE), which is then concatenated with TE and projected to produce QE. We hypothesize that this would preserve most of the information from the input query during projection.

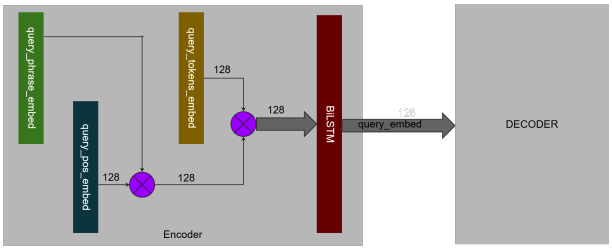


Figure 5: Architecture of Raw Query Independent Preprojection model. POSE and PhE are concatenated and projected independent of the input query. The resulting is concatenated with TE and projected to produce QE.

$TE_t$ dimension: 128
$POSE_t$ dimension: 128
$PhE_t$ dimension: 128
Dense = (128 * 2) input nodes, 128 output nodes

$AE_t = \text{Dense}([POSE_t : PhE_t])$
$ATE_t = [TE_t : AE_t]$
$ATEP_t = \text{Dense}(ATE_t)$
QE = BiLSTM(ATEP)

## VI. Experiments

All decoder dimensions and configurations are left untouched and are thus the same as in Pengcheng's model. For each of the above models, the embedding sizes are as described in Models. Each model is run for a maximum of 50 epochs, with early stopping if the validation metrics do not change for 10 epochs.

## VII. Results

We evaluate our model on two metrics: BLEU and Accuracy.

1) BLEU: NLTK's bleu_score function is used with NIST geometric sequence smoothing.
2) Accuracy: We use exact match for accuracy. +1 if the token matches the reference exactly, 0 otherwise. The total is normalized by the number of the prediction.

| Models | | Metrics | |
|---|---|---|---|
| | | BLEU | Accu. |
| Pengcheng | | 84.5 | 71.6 |
| Base | | 73.2 | 67.9 |
| NL2code | BC | 73.6 | 69.4 |
| | LP | <u>74.3</u> | <u>69.7</u> |
| | LPrd | 73.6 | 69.0 |
| | AdvLP | 73.7 | 69.1 |

## VIII. Conclusion

On top of the existing state-of-the-art model on code generation, we incorporated POS and phrase information of the input query into model, and tested if these additional features improve the effectiveness of the model. We found that the POS and phrase information does improve the BLEU scores by 1% and accuracy by 2%. We also did a critical analysis of the datasets available for code-generation and proposed a denoising approach to make one of the datasets (Django dataset) more generic.

Though, Neural Network approaches give state-of-the-art results, it fails badly on some easy cases as well. Therefore, we believe more inclusion of semantic information and additional features is important and wish to explore it in the future.

## References

[1] J. Berant, A. Chou, R. Frostig, and P. Liang, "Semantic parsing on freebase from question-answer pairs," in Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing, 2013, pp. 1533–1544.
[2] T. Lei, F. Long, R. Barzilay, and M. Rinard, "From natural language specifications to program input parsers," in Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), vol. 1, 2013, pp. 1294–1303.
[3] L. Mou, R. Men, G. Li, L. Zhang, and Z. Jin, "On end-to-end program generation from user intention by deep neural networks," arXiv preprint arXiv:1510.07211, 2015.
[4] W. Ling, E. Grefenstette, K. M. Hermann, T. Kočiskỳ, A. Senior, F. Wang, and P. Blunsom, "Latent predictor networks for code generation," arXiv preprint arXiv:1603.06744, 2016.

[5] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," arXiv preprint arXiv:1704.01696, 2017.
[6] M. Rabinovich, M. Stern, and D. Klein, "Abstract syntax networks for code generation and semantic parsing," arXiv preprint arXiv:1704.07535, 2017.
[7] A. V. M. Barone and R. Sennrich, "A parallel corpus of python functions and documentation strings for automated code documentation and code generation," arXiv preprint arXiv:1707.02275, 2017.