PNet - ParseNet

Progress Report

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. Changes

We have changed the datasets used for our task of convertin natural language comments to source code. As mentioned in the Edinburgh paper [1], the Hearthstone and Django datasets, are not generalizable and would not be the right data to use for the task (Discussed more elaborately in Data). As a result, we decided to scrape our own data from open-source python repositories with guidance from [1].

II. Revised Project Plan

Due to the important task of gathering our own data for the task, the data creation and processing phase of our original plan has taken up significant time, resulting in the following new plan.

1) Literature Survey [Done]
2) Data Preprocessing [Done]
3) Baseline models [Done]
4) Model training [Future work]

III. Data

A. Problems with existing datasets

1) Django (DJ): The comments in the Django dataset are human annotated to explicitly describe what the following line of code is. Because of this, the BLEU scores reported by the research community on the Django dataset is high. However, this is not the case in general production environment and the dataset is less useful for our task of converting natural language comments to code.

[1] also critiques about the Django dataset on similar lines. Consider the following example from the Django dataset:

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

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

Example 1

As we can see, programmers are very unlikely to write such elaborate comments, and learning from such input pairs, while making it feasible to achieve high BLEU and accuracy scores, is not very useful for automatic code generation in production environments

2) Hearthstone (HS): The Hearthstone dataset contains a single template of code. The comment-code pairs used to train and test generate only a single template of code, which is far from the case in general production environment. Because of this, the BLEU score reported on the Hearthstone dataset is high. [1] also critiques about the HearthStone dataset on similar lines.

```
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()))])
```

Example 2

The dataset is full of such descriptions. Models trained on this dataset typically cannot work on any other input description. Hence clearly not suitable for converting production time comments to code.

3) Edinburgh: To overcome the shortcomings of the Django and Hearthstone datasets as described above, the [1] proposed a dataset of docstring-code pairs (here after referred to as "Edinburgh dataset") that they scraped from open source 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. Unfortunately, for generating source code we need information on implemention logic.

```
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))
```

Example 3

Given such a small description, there is no way an automatic code generation system will be able to generate the target code without additional information; Therefore, we conclude that code-docstring pairs are also not ideal for out task of generation source code from the comment/description

B. Data gathering

To overcome the shortcomings of the aforementioned datasets, we build a new dataset. We cloned open-source Python repositories as mentioned in [1], and parsed the .py files to get all the comment-code pairs. We were able to generate 200K code-comment pairs.

C. Data preprocessing

1) Enforcing Constraints: Consider only the comment-code pairs, where the code span is less than 5 lines. This will enforce that our data has only comment-code pairs where the comment is relevant to the corresponding source code.

2) POS Tagging: We parsed each comment in the dataset with NLTK's stanfordParser to obtain the POS tags of each token in the comment. We also find spans of phrases in the comment such as (NP PP VP ADJP etc.). We aspire to leverage this syntactic information to improve the model accuracy/BLEU scores. Please refer to the example below.

```
comment:
  add a and b and store the result into c

code:
  c = a + b
```

Example 4

Here "into c" is a Proportional phrase and the variable C appears in the left hand side of the statement. Given syntactic information, we believe the model will learn such associations.

D. Canonicalization

The raw comment-code pairs, have many project specific information (variable names, class names, function names etc.). We consider this information as noise as it would not contribute to learning the general structure of the program. However the types of all those variables is very important information. During canonicalization, we replace every reference to an identifier by the type of that identifier. Since Python is dynamically typed, we extract these type information using hand-written rules.

```
Before Canonicalization:
  comment:
    True if we are running on Python 3
  code:
    PY3 = sys.version_info[0] == 3

After Canonicalization:
  comment:
    True if we are running on Python NUM_1
  code:
    ANY_0 = (sys.GEN_0['NUM_0'] == 'NUM_1')
```

Example 5

Intuition:

- ANY_0 - During code generation, the LHS of the assignment can be of any reasonably named string. This will not affect the logic of the program
- GEN_0 - Can be any attribute of the sys module; (sys is not canonicalized as it is a built-in python module)
- NUM_0 - It can be any number
- NUM_1 - copied from the comment.

The idea is that canonicalization, represents the user intent in a project agnostic way. And we can substitute these canonicalized terms, using the context information based on where the comment appears during code generation

E. Comment-Code relevance

A comment is relevant to the code only if it talks about some variable in the code. Based on this hypothesis, we keep only the comment-code pairs, where the comment has a direct reference some variable/canonicalized-type in the code.

After all preprocessing, we end up with 12.5K comment-code pairs.

IV. Methods

The two methods used:

1) We fit our dataset to the method suggested by [2] (referred to as "Pengcheng's model", henceforth), which is also our basepaper. Pengcheng's method induces a grammar for the input source-code and establishes the problem as a "sequence of tokens (comments)" to "sequence of grammar rules". The method uses an Encoder-Decoder model with Bidirectional LSTM for the encoder and a RNN with soft attention, similar to [3].
2) Vanilla seq2seq: To further illustrate the problems with HS and DJ data, we train a vanilla seq2seq

model in TensorFlow. Google separately provides a high-level API for the same at google/seq2seq [4], under the Apache 2.0 license. We run the same model without modification.

V. Results

The sentence level BLEU scores, and the accuracy acheived using the Pengcheng's model is, 14.2 and 7.2 respectively. These scores are very less as compared to the scores reported for the Django and HearthStone datasets. Please refer to Table 1.

HearthStone		Django		Our DS	
Acc.	BLEU	Acc.	BLEU	Acc.	BLEU
16.2	75.8	71.6	84.5	7.2	14.2

Table 1

The unusually high BLEU scores indicate the issue with the Django and HearthStone datasets. However, The BLEU scores on our dataset, being similar to the baseline BLEU score mentioned in [1] indicates that our dataset expresses the toughness of the problem adequately. Since Pengcheng's model has preformed poorly, it is to no surprise that the vanilla seq2seq model also performs poorly with it's rather simplistic architecture.
We therefore consider both the above as baseline.

VI. Future Word

Our baseline results, suggest that converting Natural Language comment to code is a hard task. However, the research community in this area have largely ignored the context in which a comment appears. We tend to leverage that information for better code generation.

1) We will train a vanilla LSTM on the canonicalized code-comment pairs with the POS, and Phrase information. We hypothesize that this will be easier the model to learn, as we have a smaller target vocabulary, compared to the raw data.
2) Leverage Information Retrieval techniques to predict apt variable names during code generation, i.e. when we convert canonicalized code output from step 1 to actual compilable target code.

References

[1] 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.

[2] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation."

[3] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," arXiv preprint arXiv:1409.0473, 2014.

[4] D. Britz, A. Goldie, T. Luong, and Q. Le, "Massive exploration of neural machine translation architectures," arXiv preprint arXiv:1703.03906, 2017.