Semi-Supervised Code Generation by Editing Intents

Edgar Chen
advised by Graham Neubig, Bogdan Vasilescu
Carnegie Mellon University
`edgarc@cs.cmu.edu`

```
Intent:  count the occurrences of
a list item
Code:  l .  count ( 'b' )
```

Figure 1: A datapoint from StackOverflow, representing a Question (intent) and its Answer (snippet)

Abstract

In this work, we explore the semantic parsing task of generating code in a general-purpose programming language (e.g. Python) from a natural language description. One significant issue with the current state of code generation from natural language is the lack of data in a ready-to-translate form, ie written as a command with variable names specified. We propose a deep generative model to rewrite and augment utterances such as StackOverflow questions into an acceptable form by adding variable names and other important information. We treat the rewritten natural language representation as a latent variable, and sample an edited natural language sentence which we condition the code generation process on. We train the models separately in a supervised setting and together in an unsupervised setting.

1 Introduction

Programmers learning a new programming language like Java or Python can often struggle with implementing a simple command into code, due to unfamiliarity with syntax or other issues. Even experienced programmers can forget the specific details of certain library functions. Often, programmers stuck on such details will query search engines with natural language descriptions looking for reference materials or contents on question and answer sites such as StackOverflow (SO). However, this means that the programmer must search for the right question, then the right answer, then understand the code in the answer well enough to implement his own version of it to attain the results that he is interested in, which can be tough for a novice.

Semantic parsing is the category of tasks in NLP which involve transforming some unstructured natural language text into a structured logical representation. These representations can range from simple grammars for querying knowledge bases (Berant et al., 2013) to smartphone voice commands which can be executed by virtual agents such as Siri and Alexa (Kumar et al., 2017). NLP researchers have shown significant progress on learning these representations using data. However, these representations are often limited in their expressivity due to limitations in the expressive power of the syntax and the limited domain of the task. The ability to generate code in general-purpose programming languages is desirable as they are both sufficiently expressive for programmable tasks and can easily be understood and integrated by programmers into existing systems and codebases.

One necessity for learning a strong code generation system is data which includes a concise natural language *intent* for which the corresponding code *snippet* captures all of the instructions and detail in the intent, while not including any extraneous code. Previous works have focused on cases where code is annotated, line-by-line, with detailed natural language descriptions that have variable names (Oda et al., 2015), or where the code can only interact with a fixed number of variables (i.e. pieces in a game) (Ling et al., 2016). We would like to our models to capture the complexities of all questions that programmers can ask, which can be achieved by mining data from a question-and-answer site like StackOverflow which allows users the freedom to ask any code-related question. However, using the question text from an SO post is often insufficient information to actually generate code. In a usual code generation task, an example input would be a fully-formed command "How to get maximal element in list 'my_list'", and its corresponding output would be a line of code: `max(my_list)`. However, on StackOverflow a question posted by users would take the form "How to get maximal element in a list?", expressing only an abstract intent, and the answer to the post would still be `max(my_list)`. Despite the fact that the intent and snippet agree, the intent does not contain enough information to generate the entire code snippet, as it is missing the variable name of the list.

One approach to fixing this issue would be manual annotation, but the process is slow and takes a skilled programmer in the target programming language, and thus is not easily scaled. We take a two-step approach to utilize this data: First, we construct a model which rewrites the abstract natural language intents into concrete commands by augmenting them with variables from the code and other important information such as variable types. Then, we construct a model which generates code snippets from these *rewritten intents*. Because data with rewritten intents is sparse, we treat the rewritten intents as underlying latent variables which we infer to perform the final code generation step, and train our models in tandem, semi-supervised. This approach surpasses our baseline approach of ignoring the rewriting step and just generating code directly from the intent, as well as learning rewritten intent generation using only the small supervised dataset.

2 Related Work

Gradient-based optimization of generative latent-variable models was first popularized with the Variational Auto-Encoder (VAE) (Kingma and Welling, 2013). However, Kingma and Welling focused on sampling and training continuous latent variables. Miao and Blunsom (2016) repurpose the ideas from the VAE to train models structured, discrete latent variables, and demonstrate that the latent variables can be used as the summaries in sentence compression to good effect.

```
Intent:  count the occurrences of
a list item
Code:  l . count ( 'b' )
Rewritten Intent:  count the
occurrences of item 'b' in list l
```

Figure 2: A datapoint separated into 3 parts: intent in blue, code in green, and generable tokens in red.

Xu et al. (Lin et al., 2017) have worked on learning to generate Bash shell script commands from natural language. Their approach relies on synthesizing a template for a Bash one-liner, then filling in the arguments. They show good results in both accuracy and human evaluation, and show that their model is able to capture relatively complex intents as commands and arguments. However, they limit the scope of the task to running Unix command-line utilities with arguments, which means that many of the powerful elements of Bash syntax are lost, such as control flow, I/O redirection, and variables, as most command-line use of Bash does not include these features. Our work is a step closer to generating code in a general-purpose language which allows these more complex syntactic features.

3 Methods

3.1 Problem Description

A sample of fully annotated data is shown in Figure 2. We would like to infer high-quality rewritten intents from the intent (English) and code (Python). The quality of a rewritten intent is measured by its faithfulness to the original intent and its completeness in providing a fully-formed natural language command allowing humans and machines to write code with no additional information. To facilitate the generation of these rewritten intents, we require that they are composed of mostly copies of words from the intent and variable names and literals (strings, integers, etc.) from the code. This means that the rewritten intent will mostly contain content from the original intent, with some variable names possibly added to allow for a complete generation. In Figure 2, for example, without the variable names inserted into the rewritten intent, it is impossible to write the code; even if we know that we want to call the `count` function, there's no indication as to which list it should be called from or what object it should be called on. The rewritten intent tells us that the list is `l` and that `count` should be called on `'b'`. One last element of the rewritten intent is the generable tokens: these are common words such as prepositions (in, of, on) and types (list, file) which are sometimes not found in the intents due to their brevity. We allow the model to generate these for higher fidelity and clearer match with the manually annotated rewritten intents.

3.2 Model

The model can be separated into two parts: there is a rewritten intent generator and a code generator.

3.2.1 Rewritten Intent Generator

The base model is an encoder-decoder (Cho et al., 2014) model. We use fine-tuned embeddings $e(w)$ to convert each token in the snippet and intent into a vectors, then encode each sequence separately with bidirectional LSTMs (Hochreiter and Schmidhuber, 1997). The decoder is an LSTM which allows for copies using pointer networks as described in Gu et al. (Gu et al., 2016) We would like to generate probabilities for the rewritten intent $z = [z_1, z_2, ..., z_{|z|}]$. The intent $x = [x_1, x_2, ..., x_{|x|}]$ and snippet $y = [y_1, y_2, ..., y_{|y|}]$ can be encoded and concatenated into a single list of vectors:

$$\begin{aligned} r = [\,&\text{BiLSTM}(e(x_1), ..., e(x_{|x|})), \\ &\text{BiLSTM}(e(y_1), ..., e(y_{|y|}))] \end{aligned} \qquad (1)$$

Let h_t be the current hidden state of the LSTM. Then we calculate the copy scores α_t:

$$\alpha_{t,i} = v_a^\top \tanh(W_a[h_t; r_i]) \qquad (2)$$

Where v_a, W_a are trainable parameters. We explicitly disallow copies of code tokens which are not variable names or literals, e.g. `"foo"` or `3`. Then we compute scores β_t for generable tokens. These are computed directly from the hidden state:

$$\beta_t = W_g h_t \qquad (3)$$

Once we have computed these two, to get the final generation probabilities we compute

$$q_t = \text{softmax}([\alpha_t; \beta_t]) \qquad (4)$$

The probability of generating a copy or token with index i as the next word is $\Pr(z_t = i) = q_{t,i}$.

Finally, we calculate h_{t+1}. If we decided to copy a token, we would like to use information about the copied token, so we augment the input to the LSTM f, the embedding of the copied word $e(z_t)$, with the encoding of the copied token $r[z_t]$. If z_t was not copied and instead generated, then we let $r[z_t] = 0$.

$$h_{t+1} = f(h_t, [e(w_t); r[z_t]]) \qquad (5)$$

3.2.2 Code Generator

The code generator is also an encoder-decoder LSTM model. We use a bidirectional LSTM to encode the rewritten intent, to obtain a list of encodings $s = \text{BiLSTM}(z_1, z_2, ..., z_{|z|})$. Then, we calculate scores for copying and generating code tokens $\hat{y} = [\hat{y}_1, ..., \hat{y}_{|\hat{y}|}]$. For copies, we employ the same strategy as before: letting the current hidden state be h_t, we calculate only for i such that w_i is a code token:

$$\gamma_{t,i} = v_b^\top \tanh(W_b[h_t; s[i]]) \qquad (6)$$

Letting v_b, W_b be trainable parameters. To calculate the probabilities of the code tokens, we calculate

$$\delta_t = U_c((\text{softmax}(v_c^\top \tanh(W_c[h_t; s[i]])))^\top r) \qquad (7)$$

Letting U_c, v_c, W_c be trainable parameters. To get the final token generation probabilities $\Pr(\hat{y}_t = i) = p_{t,i}$ we take

$$p_t = \text{softmax}([\gamma_t; \delta_t]) \qquad (8)$$

Since the purpose of this is not primarily to copy, we do not use the encoding of copied tokens when computing the next hidden state

$$h_{t+1} = f(h_t, e(\hat{y}_t)) \qquad (9)$$

3.2.3 Training

The training process roughly follows the setup in Miao and Blunsom (Miao and Blunsom, 2016), which also has a discrete VAE architecture. Supervised data is trained in the usual

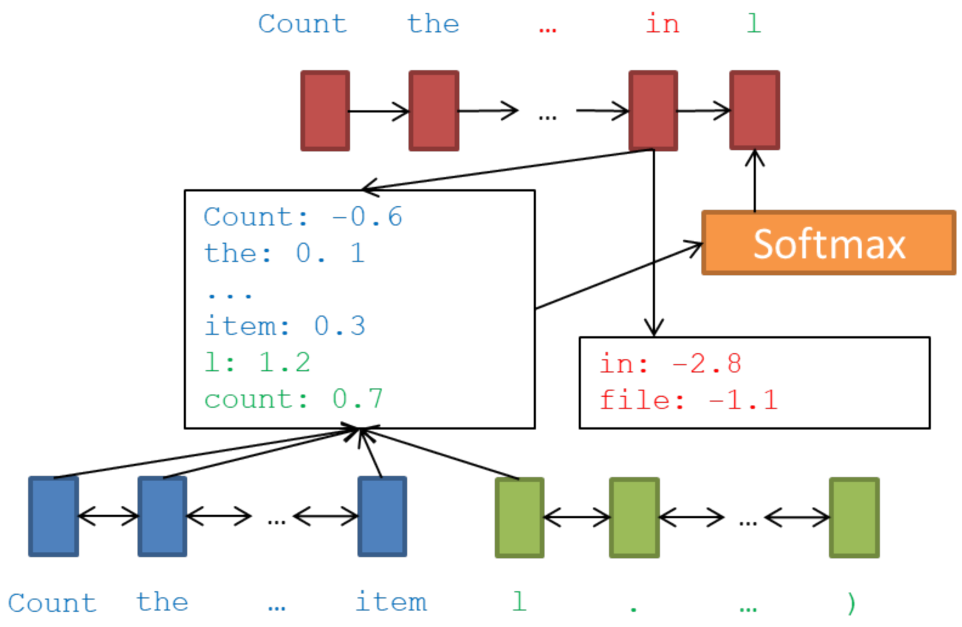

Figure 3: Diagram of the Rewritten Intent Generator.

sequence-to-sequence method of maximizing the log-likelihoods $\sum_t \log q_t, \sum_t \log p_t$. Let ϕ, θ be the parameters of the rewritten intent and code generation models respectively. Unsupervised data is trained by sampling from the latent variable distribution $q_\phi(z|x,y)$, then computing the probability distribution $p_\theta(y|z)$ in the code generation step. We want to optimize the variational lower bound of p_θ, which is

$$\mathbb{E}_{q_\phi(z|x,y)}[\log p_\theta(y|z)] - D_{KL}[q_\phi(z|x,y)||p'(w)] \tag{10}$$

$$\leq \log \int \frac{q_\phi(z|x,y)}{q_\phi(z|x,y)} p_\theta(y|z)p'(w)dw = \log p(y) \tag{11}$$

Here, p' is the prior language model on the rewritten intent distribution we are sampling from. Intuitively, this means that we want to balance out the two objectives of drawing samples the code generation model to generate code well and drawing them from the prior language model to ensure that they still look like words.

Instead of directly optimizing L, we control the importance of these objectives by introducing a coefficient λ and instead optimizing

$$\mathbb{E}_{q_\phi(z|x,y)}[\log p_\theta(y|z)] - \lambda D_{KL}[q_\phi(z|x,y)||p'(w)] \tag{12}$$

If λ is small, this means that the model will focus more on samples which perform well on code generation as opposed to samples which are likely to appear in the language model. Our ultimate goal is not to enforce the language model constraints on the rewritten intent tightly, but to create a good code generation model and rewritten intent model which supports it well.

We optimize the parameters ϕ and θ differently. For the parameters θ, we want to maximize the log-likelihood of generating the correct code snippet given z.

$$\frac{\partial L}{\partial \theta} = \mathbb{E}_{q_\phi(z|x,y)}[\log p_\theta(y|z)] \tag{13}$$

$$\approx \frac{1}{M}\sum_m \log p_\theta(y|z) \tag{14}$$

$$\approx \min_m \log p_\theta(y|z) \tag{15}$$

Equations 14 and 15 suggest two different strategies to update the paramters. In both cases, we sample M samples from $q_\phi(z|x,y)$, the rewritten intent generator. In Equation 14, we take the mean of the log-likelihood and optimize based on all the samples; in Equation 15 we optimize only on the sample which has the minimum loss. The first is the standard method of approximating the expectation, the second avoids optimizing the code generation process based on bad samples from the rewritten intent generator. The effectiveness of these strategies is explored in the experiments section.

We update the parameters ϕ in the rewritten intent generator by letting the learning signal

$$l(z, x, y) = \log p_\theta(y|z) - \lambda(\log q_\phi(z|x, y) - \log p'(w))$$

Then we take the gradient:

$$\frac{\partial L}{\partial \phi} = \mathbb{E}_{q_\phi(z|x,y)}[l(z, x, y)\frac{\partial \log q_\phi(z|x, y)}{\partial \phi}] \quad (16)$$

$$\approx \frac{1}{M}\sum_m [l(z^{(m)}, x, y)\frac{\partial \log q_\phi(z^{(m)}|x, y)}{\partial \phi}] \quad (17)$$

Taking M samples as we did before. However, this estimator has high variance because of the variability in quality of samples from $q_\phi(z^{(m)}|x, y)$. Thus, we use REINFORCE (Williams, 1992) to reduce the variance of the training process, and introduce a baseline $B(x, y)$. For the baseline to be effective, we minimize the following expectation during training:

$$\mathbb{E}_{q_\phi(z|x,y)}[l(z, x, y) - B(x, y)]^2 \quad (18)$$

We use a MLP which takes as input the bag of words for x, y and the lengths of the sentences as our baseline B. Our final gradient estimator is then

$$\frac{\partial L}{\partial \phi} \approx \frac{1}{M}\sum_m \Big[(l(z^{(m)}, x, y) - B(x, y))\frac{\partial \log q_\phi(z^{(m)}|x, y)}{\partial \phi}\Big] \quad (19)$$

4 Experiments

4.1 Dataset and Preparation

The data consists of three parallel English-Python corpora mined from Stack Overflow in a collaborative effort between Pengcheng Yin, Bowen Deng, myself, and the two advisors on this project, Graham Neubig and Bogdan Vasilescu. Our mining method is described in Yin et al. (Yin et al., 2018); we calculate alignment scores between the questions asked on SO and snippets of code posted in the answers. Specifically, we calculate the alignment scores by training a logistic regression classifier on a list of hand-tuned features such as vote count and snippet length, as well as machine-learned scores which measure the ability of a model to translate the natural language into code and vice versa. Although the initial annotation and mining process worked well on the top 1000 questions, extending to the top 10000 questions on SO proved to be difficult. While the top 1000 questions are mostly about common Python language constructs and usage of popular functions in popular libraries, the top 10000 questions present a much more varied set of challenges, often diving deeply into a specific library or asking about how to fix a bug in a specific program. We decided to annotate the model outputs with the highest alignment scores so that we could teach the model which datapoints it misclassified, particularly types of datapoints which did not appear in the top 1000. We retrained the model on the new data and were able to make significant improvements, to the point where ¿60% of data with the highest-ranking alignment scores was correctly aligned, as opposed to 25-30%.

- Supervised: 490 question and answer pairs mined from the top 1000 Python questions on StackOverflow and manually annotated. This annotation includes the original English question (intent), the Python code snippet which reflects a correct answer (snippet), and a natural language sentence in the form of a command with all the variable names specific to the snippet which reflects the original intent of the question (rewritten intent).

- Unsupervised Verified: 390 question and answer pairs mined from the top 10000 Python questions on StackOverflow and manually verified. Only English (intent) and Python (snippet) data are provided.

- Unsupervised: 2242 question and answer pairs mined from the top 10000 Python questions on StackOverflow, with no manual verification. Only English (intent) and Python (snippet) data are provided.

4.2 Implementation

The model is implemented in Dynet (Neubig et al., 2017), with 2-layer LSTMs with hidden layer and input size 128. We use the Adam optimizer (Kingma and Ba, 2014) and Dropout (Zaremba et al., 2014) to improve the optimization process.

For the following experiments, we pre-trained with Django data, which consists of line-by-line annotations of the Python Django library (Oda et al., 2015), and supervised data for 20 epochs, then trained with a random sample of 10% of the supervised data points combined with the all of unsupervised datapoints. We use a 60/40 train/test split and measure the losses.

4.3 Results

When unsupervised data is added to the training in epoch 21, both Figure 4 and Figure 5 reflect increased ability to generate code. The model seems to learn how to better utilize the rewritten intents it creates to generate code, even if it doesn't learn how to generate rewritten intents closer to our annotations. This could potentially be due to the fact that rewritten intents can be written in many ways, and the model only selects one style for code generation. Figure 6 contains outputs of the program. Notably, the code generation tends to fall into similar patterns, suggesting that there isn't enough of a learning signal to learn rewritten intents well from the unsupervised data in our current model.

5 Conclusion and Future Work

There is still a long way to go on learning from the StackOverflow dataset. This work itself is far from complete; there are still areas in which the model could be optimized for higher performance, as it is not currently generating good outputs. To this end, we plan to improve the data in quantity and quality of annotations, optimization process, and model architecture. However, even if this work were complete, it would just be a single step towards generating the correct code from StackOverflow. There are still many issues to be covered, from referencing the right variables and arguments to using the appropriate syntax, and unlike tasks like Machine Translation, a mostly-correct code generation will not generate the appropriate output when run, but a mostly-correct translated sentence is still comprehensible by speakers of the target language. We hope that this work, when complete, will encourage the development of more robust code generation models.

References

Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. 2013. Semantic parsing on freebase from question-answer pairs. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1533–1544. Association for Computational Linguistics.

Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using rnn encoder-decoder for statistical machine translation.

Jiatao Gu, Zhengdong Lu, Hang Li, and Victor O. K. Li. 2016. Incorporating copying mechanism in sequence-to-sequence learning.

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

Diederik P. Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980.

Diederik P Kingma and Max Welling. 2013. Auto-encoding variational bayes.

Anjishnu Kumar, Arpit Gupta, Julian Chan, Sam Tucker, Björn Hoffmeister, and Markus Dreyer. 2017. Just ASK: building an architecture for extensible self-service spoken language understanding. *CoRR*, abs/1711.00549.

Xi Victoria Lin, Chenglong Wang, Deric Pang, Kevin Vu, Luke Zelemoyer, and Michael D. Ernst. 2017. Program synthesis from natural language using recurrent neural networks.

Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tom Koisk, Andrew Senior, Fumin Wang, and Phil Blunsom. 2016. Latent predictor networks for code generation.

Yishu Miao and Phil Blunsom. 2016. Language as a latent variable: Discrete generative models for sentence compression.

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.

Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 574–584.

Ronald J. Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3):229–256.

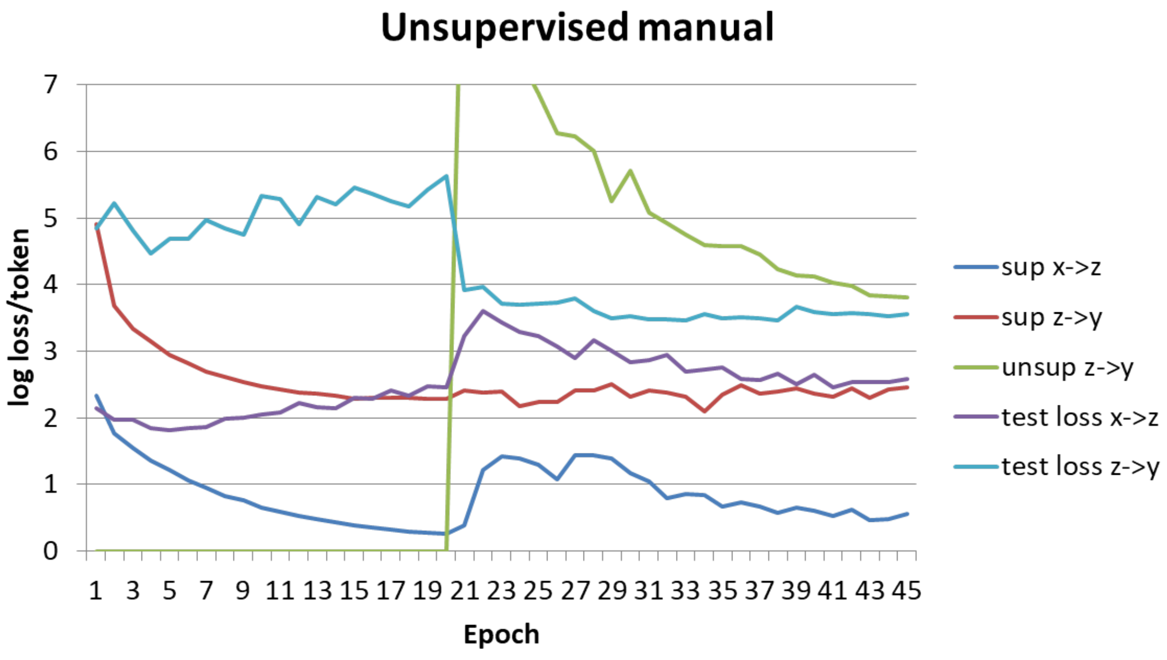

Figure 4: Results of training on the unsupervised, verified data.

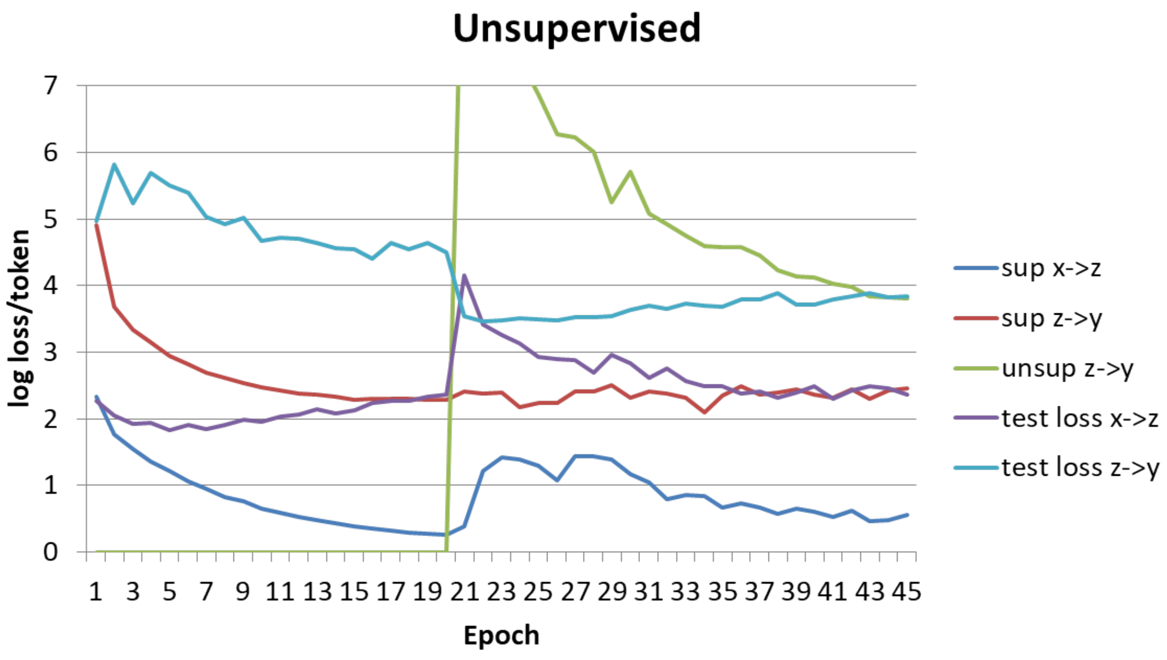

Figure 5: Results of training on the unsupervised data.

```
intent:     count the occurr of a list item
code:       dict ( ( ( x , l . count ( x ) ) for x in set ( l ) ) )
gold_rewritten: count the occurr of item in list l
pred_rewritten: count the occurr of item x <unk> is in <unk> x ) <unk> <unk>
    of <unk> key ) ) l x <unk> <unk> x
pred_code:  sorted ( l , key = lambda x : ( x [ 1 ] , x [ 1 ] ) )
------------------------------------------------------------
intent:     count the occurr of a list item
code:       l . count ( 'b' )
gold_rewritten: count the occurr of item 'b' in list l
pred_rewritten: count the occurr 'b' from list l <unk> <unk> <unk> case of of
    <unk> <unk> <unk> of <unk> of <unk> <unk> of <unk> <unk>
pred_code:  re . findall ( 'b' ( 'b' )
------------------------------------------------------------
intent:     revers a string
code:       a_string [ : : ( - 1 ) ]
gold_rewritten: revers a string a_string
pred_rewritten: revers a string a_string to 1 is save last <unk> of a_string
pred_code:  a_string . join ( { 1 ]
------------------------------------------------------------
intent:     check if a variabl exist
code:       if ( 'myvar' in globals ( ) ) :
pass
gold_rewritten: check if a <unk> variabl 'myvar' exist
pred_rewritten: check if object 'myvar' <unk> <unk> variabl 'myvar' and 1 pass
     <unk> pass <unk> pass <unk> <unk> pass
pred_code:  super ( 'myvar' ) . reverse ( )
------------------------------------------------------------
intent:     add new item to dictionari
code:       default_data . update ( { 'item4' : 4 , 'item5' : 5 , } )
gold_rewritten: add key <unk> <unk> 'item4' <unk> 4 and 'item5' <unk> 5 to
    dictionari default_data
pred_rewritten: check new item 'item4' 5 <unk> : <unk> dictionari 'item4' '
    item5' <unk> 4 'item5' <unk> , 'item5' <unk>
pred_code:  datetime . datetime . strptime ( <unk> <unk> <unk> <unk> <unk> <
    unk> 1 ] ) . 5
------------------------------------------------------------
intent:     add key to a dictionari
code:       data . update ( dict ( a = 1 ) )
gold_rewritten: add key a to dictionari data with <unk> 1
pred_rewritten: <unk> key a to a dictionari data from 1 <unk> case 1 1 <unk>
    right 1 ) )
pred_code:  datetime . datetime . strptime ( a ( a 2 ) )
```

Figure 6: Output of the program on unsupervised data.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. *Mining Software Repositories*.

Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. 2014. Recurrent neural network regularization.