Project Report

Deep Learning For Modern Code Review

Graph based approach

Supervisor: MSc. Robert Heumueller

By Govind Shukla
Matriculation Nr. :- 235192
Data and Knowledge Engineering

Table of Contents

Abstract

In this project, a study is conducted to exploit the state of the art model GSCS[1] (**G**raph structure and **S**emantic sequence for **C**ode **S**ummarization) for modern code review. To analyze the tokenized abstract syntax tree of the program, GSCS uses Graph Attention Networks. These networks use a multi-head attention mechanism to learn node characteristics in various representation sub-spaces and aggregate features by giving each node's neighboring nodes varying weights. The code summarization and code review belong to the same task space with similar evaluation metrics. Experiments are conducted on the GSCS model with publicly available code review dataset namely, ETCR-Elasticsearch and Tufano's code review dataset. Code for data processing, training and evaluation can be found here - https://github.com/govind17/Deep_Learning_For_Modern_Code_Review.

Introduction

Code Reviews are considered as one of the most essential and repetitive tasks in Software development life cycle. It requires manual human inspection but humans are found to be bad at reviewing. Deep Learning has often proven to be very effective in learning patterns in the text. Programming languages have similar repetitiveness and predictability like natural languages as described by Hindle et al., 2016. Further studies showed in Maletic and Marcus, 2001 that we have to take the structural aspect of source code and the naturalness of the code tokens to understand the code fully.

Problems with Seq2Seq models like NMT, Transformers etc. are they require a lot of resources to train and evaluate. However, code is not merely a sequence of tokens but has strong structures and massive identifiers. There are far fewer words in reviews than actual machine translation tasks.

Abstract Syntax Trees (ASTs) have become a mainstream tool to capture the syntactic structure. Graph Neural networks have the capacity to operate on ASTs and also have been proved to improve performance. Furthermore, GNNs can exploit underlying structure and semantics in code.

In the paper, **Automatic source code summarization with graph attention networks**, Zhou, Yu et al. 2022 proposed a novel approach to automatically summarize the source code using GNN with self-attention mechanism and variants of recurrent neural network. The authors of the paper claim to have overcome the short-comings of other state of the art models like Conv-GNNs (LeClair et al., 2020) and Tree-LSTMs (Shido et al., 2019) which are:

- Both do not consider the context of the current node.
- The order in source code might heavily affect the basic RNN models due to their sequential order.
- They capture unnecessary details (noise) which propagates deeper into the network.

[1] Zhou et al., "Automatic Source Code Summarization with Graph Attention Networks."

Code summarization and code review are considered similar tasks in software engineering where a piece of code is reviewed just by looking at the code before the changes are merged. The findings of code summarization can pave the way for future work (partially) to automate MCR by exploiting review data from thousands of open- source projects.

Methodology

GSCS[2] (**G**raph structure and **S**emantic sequence for **C**ode **S**ummarization) outperforms the state-of-the-art baseline code summarization models on two widely adopted java data sets.

Properties of GSCS model:

- Utilizes Graph Attention Networks to process the tokenized abstract syntax tree of the program based on a multi-head attention mechanism to learn node features in diverse representation sub-spaces, and aggregate features by assigning different weights to its neighbor nodes.

- Additionally, it has an RNN-based sequence model to obtain the semantic features and optimizes the structure by combining its output with a transformed embedding layer.

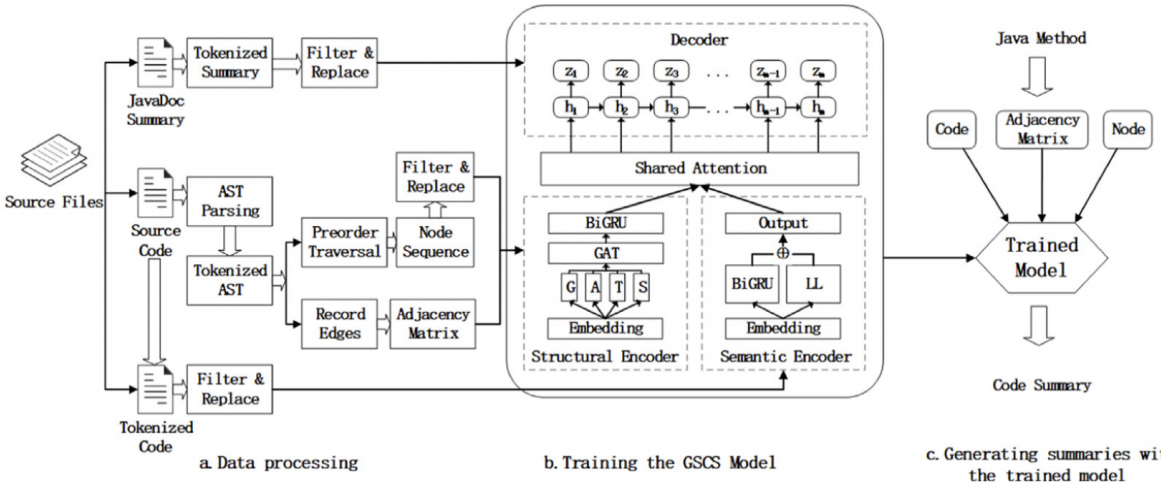

Fig. An overview of the GSCS approach by Zhou et al.

[2] Zhou et al., “Automatic Source Code Summarization with Graph Attention Networks.”

Data preprocessing

There are different kinds of input to data processing- :

Source code - Tokenized source code generated is used as an input for semantic encoder. It is also used to generate structure information i.e. node sequence and adjacency matrix.

Tokenized AST - The node sequence (a pre-order traversal of the AST) and adjacency matrix generated from the source code are sent to the structural encoder. Regular ASTs suffer from OOV words while the variant proposed by the authors of the paper captures inner representation and is also beneficial to attention-mechanism.

```
public void ClientInit(ParsingEvent pe){
    m_teamName = pe.get("team_name");
    CaseEvent ce = new CaseEvent(this, m_teamName);
    for(CaseEventListener cel:CEListeners)
        cel.Connecting(ce);
}
```

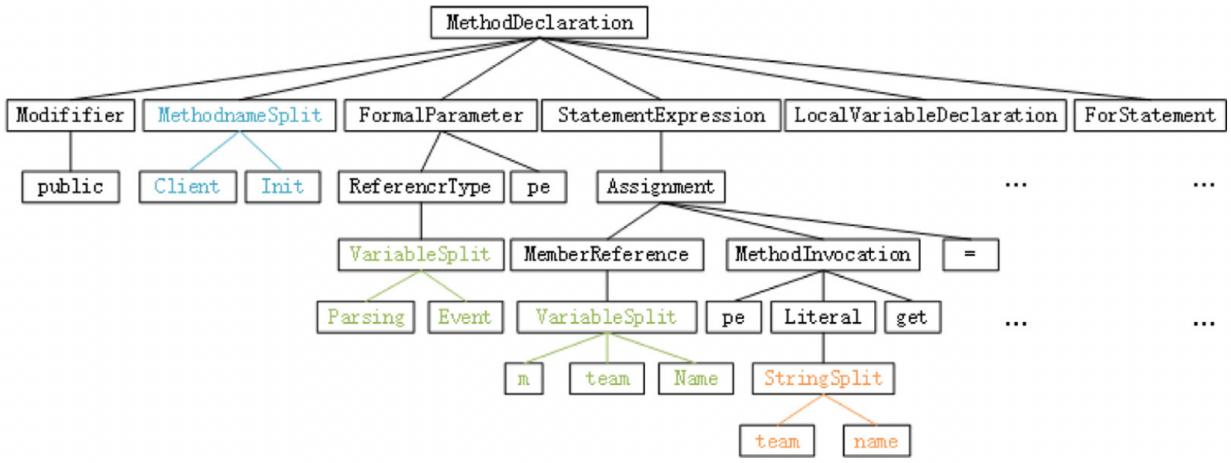

Fig. An example of tokenized AST of the given Java code by Zhou et al.

The following are the characteristics of tokenized AST trees :-

- Parse each Java method's source code into ASTs and filter out any that fail to parse.
- The authors developed rules to tokenize incomplete terminal nodes and mark them with appropriate class labels (such as method names, variable names, and long strings) based on the resulting ASTs, in order to retain the tree structure of the ASTs and provide clear semantic information to each node.
- “MethodnameSplit” is split into several subnodes like “Client”, “Init” according to the CamelCase

Review/Summary - Summary or review is parsed and tokenized for any camel_case or snake_case user defined identifiers.

GSCS Model

Consists of two encoders- structural and semantic, and a decoder.

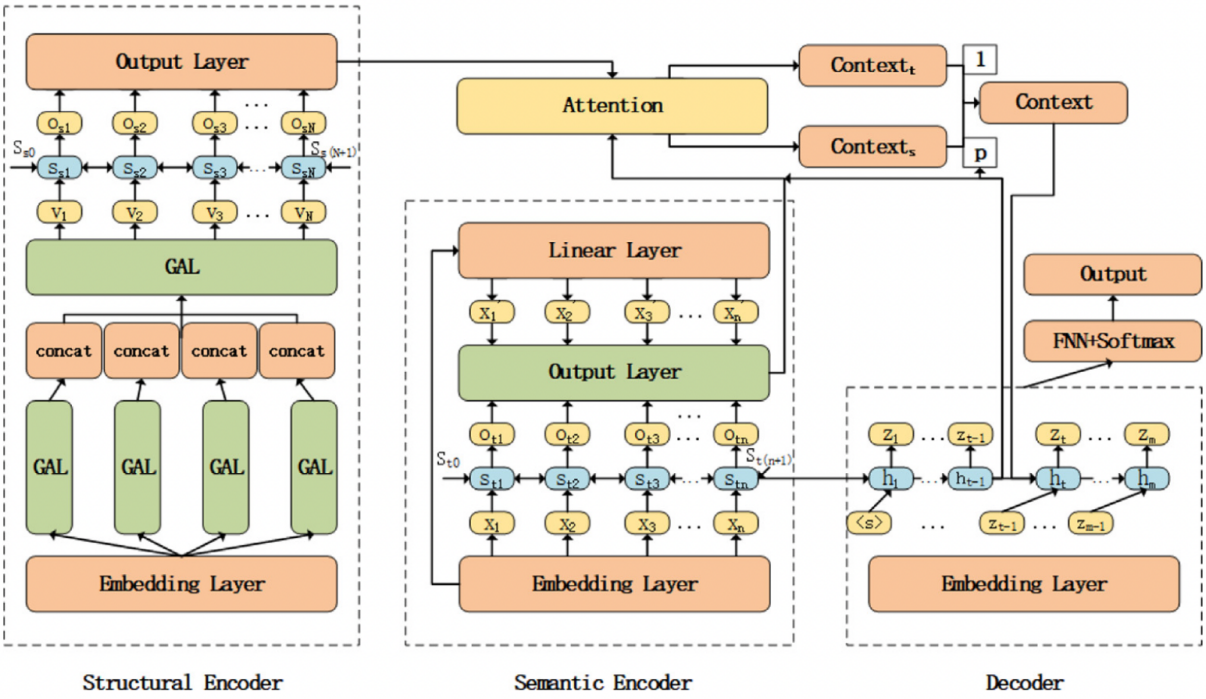

Fig. Model architecture of GSCS from Zhou et al. paper

Semantic Encoder:

Process tokenized code sequences with BiGRU

OutX, hidden = BiGRU(EM(X))
Where,

EM(X) is the embedded input,

hidden is the final hidden state of the BiGRU,

$OutX = [\overrightarrow{OutX} \parallel \overleftarrow{OutX}]$ are two independent GRU outputs in opposite direction

Structural Encoder:

Process tokenized ASTs as an undirected graph G(V,E,A) where V and E are the sets of nodes and edges, and A is the corresponding (symmetrical) adjacency matrix.
Consists of stacks of GALs to construct GAT. The goal of this GAL is to combine the properties of surrounding nodes, allowing for the discovery of more detailed information followed by Bi-directional GRU.

Properties of GAL:

- It selectively aggregates node information within 1-hop neighborhood when updating node features to produce a new set of node vector $V' = \{v'_1, \ldots, v'_N\}$

- Introduces self-attention to calculate the attention coefficient q for each pair of nodes, which is determined by the feature vector itself and other nodes.

  $$q_{i,j} = b^T[Wv_i \parallel Wv_j]$$

  $q_{i,j}$ indicates the importance of node j's features to node i,
  where // is the concatenation operation and $[W_{vi} // W_{vj}] \in R^{2F'}$ is a column vector,
  $b^T = [b_1^T, b^T_2] \in R^{2F'}$ is a row vector learned during training. b_1^T is the self attention coefficient while b^T_2 is the attention coefficient of the node j.

- The attention coefficient is assigned to the connected nodes only which means that the node feature updating only focuses on its neighbors and itself.

  $$q_{i,j} = \begin{cases} b^T[Wv_i \parallel Wv_j], & A[i][j] > 0 \\ \infty, & o/w \end{cases}$$

- Softmax is applied to normalize these scores. The attention coefficient is calculated as:

  $$\beta_{i,j} = \text{softmax}(\text{LeakyReLU}(q_{i,j}))$$
  $$= \frac{\exp(\text{LeakyReLU}(b^T[Wv_i \parallel Wv_j]))}{\sum_{k \in N_i} \exp(\text{LeakyReLU}(b^T[Wv_i \parallel Wv_k]))}$$

  Where N_i is the set of all adjacent nodes of the node i (including itself); $\beta_{i,j}$ represents the final weight coefficient assigned by the node i to the node j.

- All the nodes N_i aggregated by the weighted sum of their attention coefficients are standardized using a sigmoid function to form a high level node vector v' which comes out as an output through each GAL.

  $$v'_i = \sigma\left(\sum_{j \in N_i} \beta_{i,j} W v_j\right)$$

- K head attention mechanisms (multi-head attention) may generally be built simultaneously to capture the features of nodes from multiple perspectives.

  $$v''_i = \|_{k=1}^{K} \sigma\left(\sum_{j \in N_i} \beta^k_{i,j} W^k v_j\right)$$

  The aggregated features from each GAL are concatenated to obtain v″ i that contains KF′ features

- Another GAL layer is introduced which takes the concatenation of the outputs of the above K GALs as input and transforms the dimensions from KF' to F''

$$\widetilde{v_i} = \sigma(\sum_{j \in N_i} \beta^*_{i,j} W^* v''_j)$$

where $W^* \in R^{F'' \times KF'}$.

- Lastly, since GAT only captures local neighbors' features another BiGRU is connected to add global information. GAT produces a new set of node features which only has information up to 2 hops. A BiGRU will update each node with global information since it has capability to look into previous time-steps and later time steps.

  $\text{OutV},\ \text{hidden} = \text{BiGRU}(\widetilde{V})$ OutV is the set of new node features with Global information.

Decoder:

The purpose of the decoder is to generate the target word according to the distribution on the output vocabulary, which is guided by the shared attention and the output of the GRU at the last time step.

By giving weights to the input sequence based on the last hidden state, the attention mechanism may effectively guide decoding, allowing the model to focus on the most important bits.

- In this model, the global attention mechanism proposed by Luong et al. is used.

- A shared attention mechanism aiming to assign different weights proportional to the two encoders' hidden state.

  $$c_i = \sum_{j=1}^{n} \alpha_{ij} \text{OutputX}_j + p \sum_{j=1}^{N} \alpha'_{ij} \text{OutputV}_j$$

  where α_{ij} and α'_{ij} are obtained from the alignment model,
  n and N are the sequence lengths of the two encoders respectively,
  OutputX_i and OutputV_j represent the final outputs generated by two encoders respectively.

- Based on the last hidden state of the decoder and the concatenation of the embedded z_i with the context vector c_i, we obtain two outputs at each step of the GRU, i.e., outZ_i and h_i

- A Linear transformation on the concatenation of out_i and c_i and apply the softmax function to normalize the distribution.

  $\text{outZ}_i, h_i = \text{GRU}([\text{EM}(z_i) \ // \ c_i], h_{i-1})$

$$dist = softmax(W_z [outZ_i \ // \ c_i])$$

Now, the target vocabulary with the highest probability is selected from the distribution **dist.** Using the beam search algorithm, top 4 (in the GSCS case beam width is 4) words are chosen greedily from the vocabulary set.

Training and Evaluation

Hu Dataset (Publicly available) can be accessed through this paper: A Transformer-based Approach for Source Code Summarization. The dataset contains **87,000** Java methods with relatively complex inner structures.

Training set size: 69,696
Validation set size: 8,704
Test set size: 8,704

Hyper-parameter settings of Training	
Code vocabulary size	50,000
Node vocabulary size	50,000
Summary vocabulary size	30,000
Negative input slope of LeakyReLU nonlinearity	0.2
Amount of independent attention mechanisms	4
Dimension of the feature space	512
Output dimension of GAL	128
Output dimension of an additional GAL	512
Dropout rate	0.2
Teacher forcing ratio	0.5
Beam search size	4

Number of epochs: 200
Optimizer: StepLR
Early stopping: 20

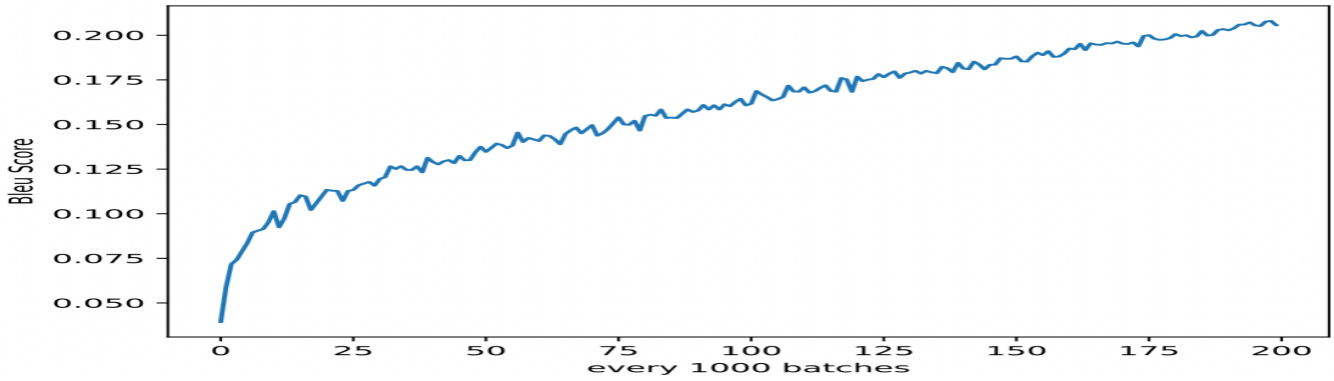

Fig. Avg. Bleu-4 score during training on validation set

Evaluation metrics

Bleu-4 score: The geometric mean of n-gram matching precision scores multiplied by a shortness penalty to discourage extremely brief produced phrases is known as BLEU. In particular, using the NLTK3 package and the smoothing function 4, we compute it at the sentence level BLEU-4.
Meteor score: It uses the synonym matching mechanism and harmonic mean values to combine the unigram matching precision and recall scores.
ROUGE-L score: ROUGE-L calculates the length of the longest common subsequence between the produced phrase and the reference. It also takes recall scores into account.

Test set results	
Corpus Bleu-4 score	0.230
Sentence Bleu-4 score	0.228
Meteor	0.365
ROUGe	0.456
Test predictions can be found here	

Remarks: The learning did not converge even after running the model for 200 epochs. It can be observed from the graph that the blue score is steadily increasing. Moreover, the process of convergence also depends on the initialization of RNN weights as observed in multiple runs.

Experiments with Code Review Datasets

ETCR ElasticSearch Dataset:

The dataset can be accessed through this paper: [Exploit those code reviews! bigger data for deeper learning]

Training set size: 10,780
Validation set size: 1,350
Test set size: 1,345

Experiment 1.1: Keeping all the hyperparameters same as before

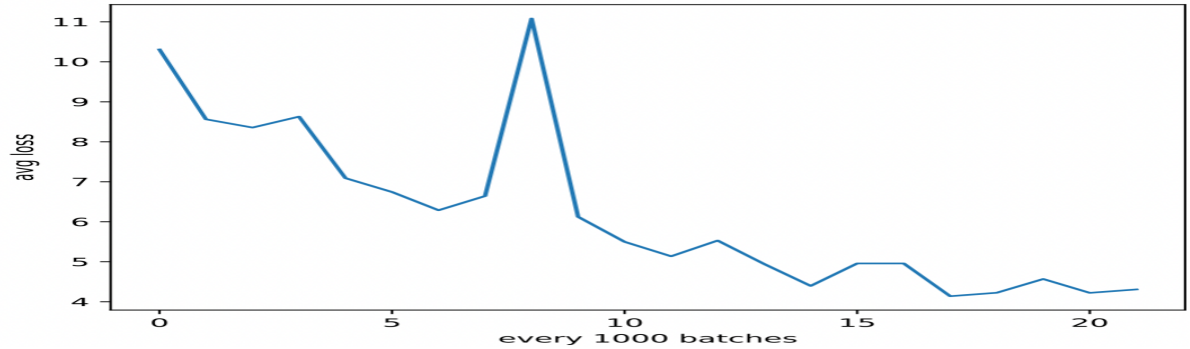

Fig. Avg. Bleu-4 score during training on validation with early stop at 23rd epoch

Test set results	
Corpus Bleu-4 score	0.0171
Sentence Bleu-4 score	0.013
Meteor	0.047
ROUGe	0.131
Test predictions can be found here	

Experiment 1.2: Beam width = 10

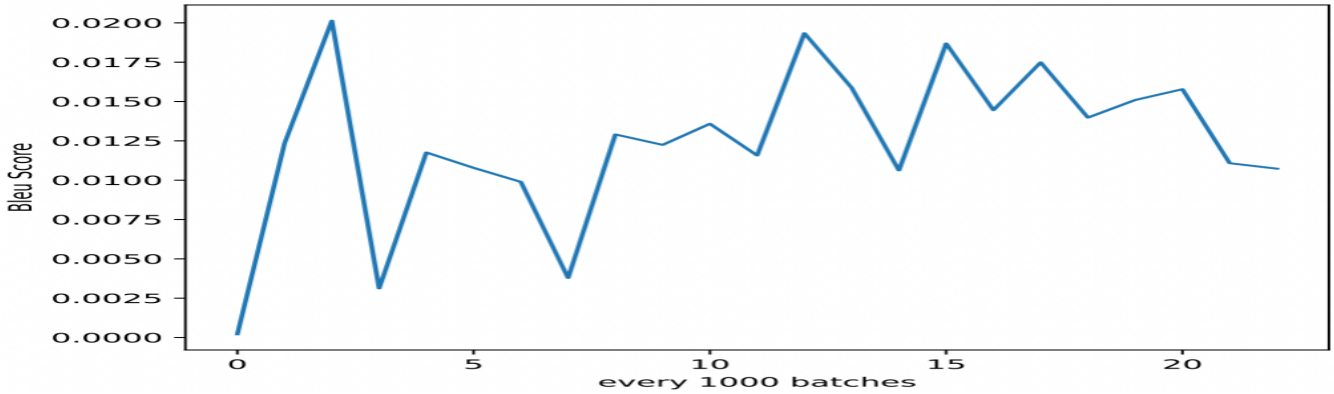

Fig. Avg. Bleu-4 score during training on validation set with early stop at 29th epoch

Test set results	
Corpus Bleu-4 score	0.0304
Sentence Bleu-4 score	0.020
Meteor	0.055
ROUGe	0.123
Test predictions can be found here	

Remarks: The size of the training dataset is too small to come to any conclusion although the beam width size has increased the scores.

Tufano's Code Review Dataset

The dataset can be accessed from the paper: Using Pre-trained Models to Boost Code Review Automation

Training set size: 134,223
Validation set size: 16,657
Test set size: 16,679

Experiment 2.1: Beam length width = 5

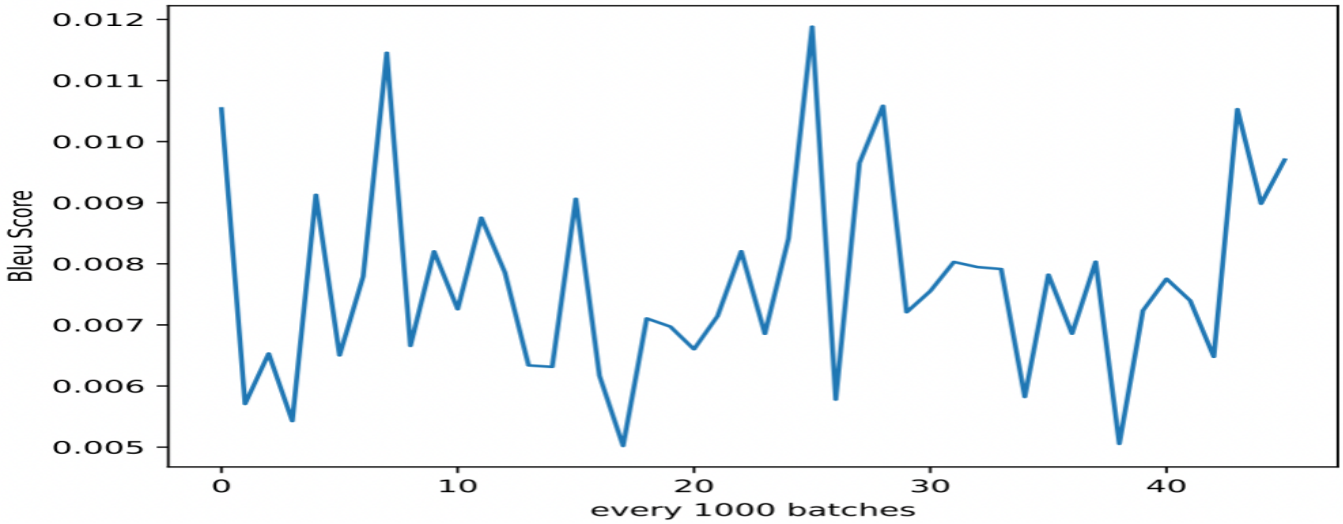

Fig. Avg. Bleu-4 score during training on validation set with early stop at 44th epoch

Test set results	
Corpus Bleu-4 score	0.0038
Sentence Bleu-4 score	0.0083
Meteor	0.034
ROUGe	0.1566
Test predictions can be found here	

Experiment 2.2: Beam length width = 7

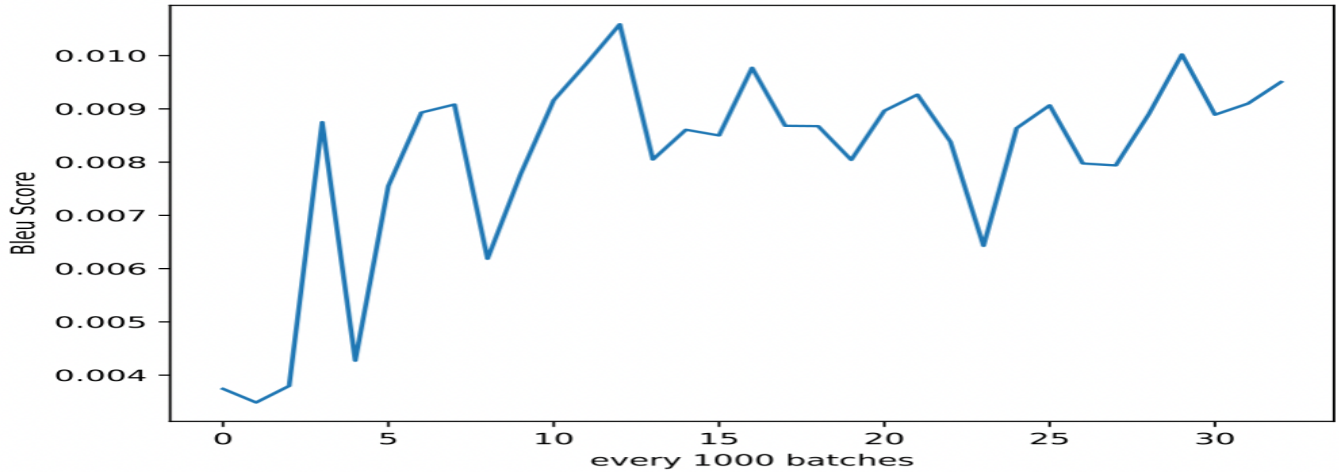

Fig. Avg. Bleu-4 score during training on validation set with early stop at 34th epoch

Test set results	
Corpus Bleu-4 score	0.0051
Sentence Bleu-4 score	0.0098
Meteor	0.0398
ROUGe	0.1614
Test predictions can be found here	

Remarks: As we can observe, the increase in beam size width also increases the performance and quality of the prediction. A lot of <UNK> tags are produced in the prediction results.

Conclusion

Overall, the state of the art model for code summarization did not perform well with code review dataset that leads to the future work where more structural as well as process changes. The following are the drawbacks of the GSCS model and possible suggestions for future automatic code review task:-

1. The GSCS model only handles JAVA 9 methods.
2. ASTs can be simplified further as redundant nodes hinders learning as mentioned here.
3. BiGRUs can be replaced by BiLSTMs since they are not performing well with the code review task.

4. Weight initialization can be optimized using He initialization as they are suited well for RNN weights.

5. For structural change, a new module can be introduced in the encoder similar to Code encoder to add information about the code change.

6. Increasing beam width certainly helps in improving prediction score but it consumes a lot of memory.

7. Regarding the reproducibility of original paper results, one of the authors pointed out that the hyper-parameters might not be consistent with the given data.

8. The target vocabulary of the code review dataset is much larger than the code summary dataset since it contains many informal words, emoticons etc. which can be sometimes random.

References

1. Hindle, A., Barr, E.T., Gabel, M., Su, Z., Devanbu, P.T., 2016. On the naturalness of software. Commun. ACM 59 (5), 122–131.

2. Maletic, J.I., Marcus, A., 2001. Supporting program comprehension using semantic and structural information. In: Software Engineering, 2001. ICSE 2001. Proceedings of the 23rd International Conference on.

3. LeClair, A., Haque, S., Wu, L., McMillan, C., 2020. Improved code summarization via a graph neural network. In: ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13–15, 2020. pp. 184–195.

4. Zhou, Yu, et al. "Automatic source code summarization with graph attention networks." *Journal of Systems and Software* 188 (2022): 111257.

5. Shido, Yusuke, et al. "Automatic source code summarization with extended tree-lstm." 2019 International Joint Conference on Neural Networks (IJCNN). IEEE, 2019.

6. LeClair, Alexander, et al. "Improved code summarization via a graph neural network." Proceedings of the 28th international conference on program comprehension. 2020.

7. Heumüller, Robert, Sebastian Nielebock, and Frank Ortmeier. "Exploit those code reviews! bigger data for deeper learning." *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2021.

8. Tufano, Rosalia, et al. "Towards automating code review activities." *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021.

9. Luong, Minh-Thang, Hieu Pham, and Christopher D. Manning. "Effective approaches to attention-based neural machine translation." *arXiv preprint arXiv:1508.04025* (2015).

10. Ahmad, Wasi Uddin, et al. "A transformer-based approach for source code summarization." *arXiv preprint arXiv:2005.00653* (2020).

11. Wu, Bingting, Bin Liang, and Xiaofang Zhang. "Turn tree into graph: Automatic code review via simplified ast driven graph convolutional network." *Knowledge-Based Systems* 252 (2022): 109450.