Unsupervised Classifying of Software Source Code Using Graph Neural Networks

Petr Vytovtov[1,2], Kirill Chuvilin[1]
[1]Moscow Institute of Physics and Technology (State University),
[2]Samsung R&D Institute Russia,
Moscow, Russia
osanwevpk@gmail.com, kirill.chuvilin@phystech.edu

***Abstract*—Usually automated programming systems consist of two parts: source code analysis and source code generation. This paper is focused on the first part. Automated source code analysis can be useful for errors and vulnerabilities searching and for representing source code snippets for further investigating. Also gotten representations can be used for synthesizing source code snippets of certain types. A machine learning approach is used in this work. The training set is formed by augmented abstract syntax trees of Java classes. A graph autoencoder is trained and a latent representation of Java classes graphs is inspected. Experiments showed that the proposed model can split Java classes graphs to common classes with some business logic implementation and interfaces and utility classes. The results are good enough be used for more accurate software source code generation.**

I. Introduction

Automated programming systems are used everywhere assisting software engineers with syntax and code style checking (simple ones) and proposing of variables names and code blocks (more complex ones). Usually such systems consist of two big parts: source code analysis and source code generation. This paper is focused on the first part.

Automated software analysis is a research area focused on statically or dynamically processing software and its source code for finding errors, vulnerabilities, bottlenecks in programs and improving their quality and readability. Dynamic analysis is applied during program execution and can find problems appeared on specific hardware with concrete input data. On the other hand static analysis is focused on source code processing, and ones try to predict mistakes in code or better variable names with its help. In this work we focus on the last type of analysis.

There are a lot of works in static software source code analysis area. Most of them are focused on bug detection and program repair [1], [2], [3], [4].

In [1] authors use a memory neural network based model for calculating the probability of potential buffer overrun error. Their approach is similar to question answer systems, and the question they ask is “Is there a buffer overrun in this line of code?” Authors of [2] consider two approaches to bug detection: adapting natural language processing (NLP) approaches (like Bag of Words and TextCNN) and augmented control flow graph (CFG) analysis with random forest algorithm. Their results show that adapted NLP approaches work better than CFG analysis. An adapted NLP approach also was used in [3]. Authors used a recurrent neural network based model for predicting a probability of bug appearing in a function represented with a sequence of lexems. Alternative approach is proposed in [4]. Authors' recurrent neural network based model tries to replicate dynamic analysis for detecting potential software bugs. Their results are slightly better than text-only and abstract syntax tree (AST) only analysis.

On the other side there are some researches about software and source code representation [5], [6], which can be used for more accurate and interpretable software source code generation. Program synthesis interpretablity can be important for detecting potential malicious behaviour (and its reasons) in automatically generated software. For example in [5] AST and data flow graph (DFG) combination is proposed for more correct source code representation, and in [6] authors calculate attention vectors for each path in AST for detecting more important ones.

For that reason in this paper we focus on such software source code representation, which can be split with some hyperplane for logically dividing source code snippets to several groups. If there is this type representation it is possible to use simple linear models for classifying source code snippets. Simple model usage increases interpretablity of classifiers, but it should be mentioned that gotten representation must be interpreted before.

That task can be decomposed to the following steps:

1) *Mining software repositories.* It is a data collection step for forming training and test sets for developed model.
2) *Representing source code as graphs.* On this step raw source code transformed into more structural representation.
3) *Preprocessing graphs.* Gotten graphs should be translated into the machine-friendly format for passing them into the model.
4) *Training an autoencoder to form latent representation.* On this step the model is trained to approximate transition functions between graph and latent spaces.
5) *Clustering graph representation in latent space.* Finally the latent space should be investigated for checking possibilities of source code classification.

For the first two steps we use approach from [7] where authors augmented source code snippet graph representation proposed in [5] with additional back-edges. For the third and fourth steps we use graph neural network library proposed in [8] and preprocess graphs as three-dimension tensors. For the last step in our paper K-Means algorithm [9] is used.

Our experiments showed that a graph neural network based autoencoder model can provide such graphs representation in a latent space that software modules can be split to implementation and interface classes. The result can be used as a block in other static source code analysis systems or for sampling an additional feature for source code generation.

Our main contribution is in introducing the method for unsupervised building source code representation for other analysis tasks.

The structure of the paper is following: Section II describes recent work in software source code analysis with neural networks, Section III introduces background information about autoencoders and graph neural networks, Section IV presents our approach for classifying source code snippets, Section V presents our experimental results, and Section VI concludes the paper.

II. Recent work

A. Software source code analysis

Software analysis can be static, dynamic, or combined. When the talk is about source code analysis, static analysis is meant. Static analysis can be rule-based or neural network based.

Rule-based methods (e.g. [10] where authors use "bottom-top" interprocedural static analysis of the whole program based on CFG gotten from LLVM [11] intermediate representation) are simple and interpretable but has poor generalization ability, which is a main problem of this group of approaches (Fig. 1 [12]). Rule-based analyzers can detect some known bugs and problems, but unknown bugs are more important and dangerous because they are not detected yet and there are no available corrections for them.

These methods are implemented in such tools as Svace [13] (an interprocedural static analysis tool for C, C++, and Java), PVS-Studio [14] (a cross-platform static analysis tool for C, C++, C#, and Java languages which uses symbolic execution and data flow analysis along with classical pattern based analysis), cppcheck [15] (a tool for C and C++ programs static analysis focused on undefined behaviour detection), and others.

Neural network based approaches can be divided to two groups: models adapted from NLP area, and models which operates with software graph representation. In the first group of approaches commonly RNN and CNN based solutions are used [16]. Approaches of this type lost structural information of source code and have a problem with potentially unbounded vocabulary size. On the other size, graph neural-network based approaches [5], [7] work with structure information about source code. In the most cases graph-based approaches have better results than NLP-based models. This type approaches have better generalization ability but require bigger datasets to train.

Our approach presented in the paper uses graph neural networks to process software source code.

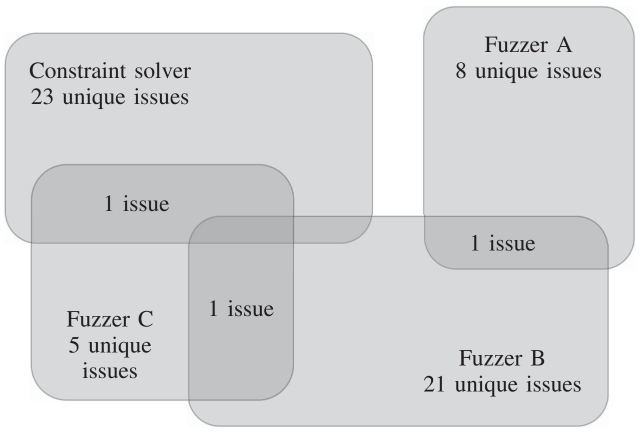

Fig. 1. Overlapping between different analyzers. As shown, different analysis tools have some common parts, but mainly each tool detects different types of error, therefore a software engineer have to use several analysis tools if maximum possible problems coverage is needed

B. Autoencoders

Firstly autoencoders were introduced in 2006 [17] and they are started to use for different tasks like dimension reduction, neural networks layers pretraining, and others. Also, as mentioned in [17], autoencoders have capability to represent data in latent space in such way, that they can be easily split to different classes.

Autoencoders are widely used in neural machine translation (NMT) tasks [18]. After their successful application for an NMT tasks, autoencoders spreaded to other areas. In software source code processing autoencoders are used for moving code snippet from one programming language to another [19] or for translating natural language to some source code [20].

In our knowledge there are no research for analysing source code representation in a latent space, and we are focusing on this aspect of autoencoders.

C. Graph neural networks

Graph neural networks are introduced in 2009 [21] as a tool for processing graph representation of data (citations, social networks, images, etc). After that graph convolutional networks appeared as an evolution [22], [23]. Now they are widely used for different tasks and most valuable approaches are implemented in different frameworks [24].

One of the last works in that area is a Graph Nets library published in 2018 [8]. In this work authors use approach based on [21] and consider a graph neural network as a combination of graph nodes, edges, and global state in such way that any graph network block can be composed with any other (which means that input and output of a graph network blocks have the same structure). Our work is based on this approach to graph processing.

III. Background

A. Autoencoders

An autoencoder is a neural network consisted of three part [17]: encoder, decoder, and latent representation (Fig. 2).

Its main task is to reconstruct the input. For that reason mean squared error (MSE) is a popular loss function for this kind of models.

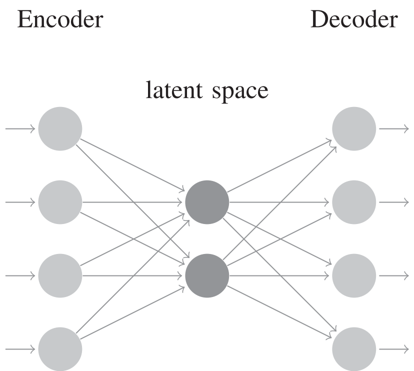

Fig. 2. The scheme of a simple autoencoder

An encoder part is a some neural network (multilayer perceptron, convolutional or recurrent neural networks) which encodes input data to some representation in a latent space. It approximates a function $Enc(\mathbf{x})\colon \mathbb{R}^{d_1} \to \mathbb{R}^{d_2}$, where $d_1 \geqslant d_2$, in such way that $X_{enc} = \{\mathbf{x}_{enc} \in \mathbb{R}^{d^2}\}$ can be split according some input data features.

A decoder part also is a neural network which is usually mirrored encoder network. E.g. if an encoder has two fully connected layers where the first layer is $FC^1_{enc}\colon \mathbb{R}^{10} \to \mathbb{R}^5$, and the second layer is $FC^2_{enc}\colon \mathbb{R}^5 \to \mathbb{R}^2$, a decoder part also will have two fully connected layers where the first layer is $FC^1_{dec}\colon \mathbb{R}^2 \to \mathbb{R}^5$, and the second layer is $FC^2_{dec}\colon \mathbb{R}^5 \to \mathbb{R}^{10}$.

It is needed to keep in mind that the output of a decoder must be as near as possible to an encoder input. For that reason sometimes ones use shared weights between encoder and decoder parts ($W_{dec} = W^T_{enc}$). So if a fully connected layer is defined as $FC(x) = f(xW + \mathbf{b})$ where x is the input data, W is the weight matrix, $\mathbf{b}$ is the bias vector, f is the activation function, $FC_{dec} = f(xW_{dec} + \mathbf{b}_{dec}) = f(xW^T_{enc} + \mathbf{b}_{dec})$.

As a result of encoder and decoder parts co-training a latent space may have such representation of input data that it can be split with some its feature as shown in Section V.

B. Graph neural networks

The main purpose of graph neural networks introduced in [21] is processing information structured with graphs. It can be images, social networks, maps, etc. In such approaches graph is considered as a system where each node depends on its neighbours. Also nodes can contain some information (e.g. type of a node) which is called label.

For processing graph with neural networks a vector representation of each node must be. On the first step all vectors (node states) are initialized with random values, and their actual values are calculated as

$$\begin{aligned} \mathbf{x}_n &= f_w(\mathbf{l}_n, \mathbf{l}_{co[n]}, \mathbf{x}_{ne[n]}, \mathbf{l}_{ne[n]}), \\ \mathbf{o}_n &= g_w(\mathbf{x}_n, \mathbf{l}_n), \end{aligned}$$

where $\mathbf{x}_n$ is the node state vector (which can be interpreted as embedding), $\mathbf{o}_n$ is the node output vector (its final value), $\mathbf{l}_n$ is the node label, $\mathbf{l}_{co[n]}$ is the adjoint edge labels, $\mathbf{x}_{ne[n]}$ is the neighbour node states, $\mathbf{l}_{ne[n]}$ is the neighbour node labels, f_w is the local transition function uses a set of parameters w, and g_w is the local output function uses a set of parameters w.

In [8] that idea was developed and edge and global states were introduced. Authors defined three "update" functions ϕ and three "aggregation" functions ρ:

$$\begin{aligned} \mathbf{e}'_k &= \phi^e(\mathbf{e}_k, \mathbf{v}_{r_k}, \mathbf{v}_{s_k}, \mathbf{u}), \\ \mathbf{v}'_i &= \phi^v(\mathbf{e}'_i, \mathbf{v}_i, \mathbf{u}), \\ \mathbf{u} &= \phi^u(\mathbf{e}', \mathbf{v}', \mathbf{u}), \\ \mathbf{e}'_i &= \rho^{e\to v}(E'_i), \\ \mathbf{e}' &= \rho^{e\to u}(E'), \\ \mathbf{v}' &= \rho^{v\to u}(V'), \end{aligned}$$

where

$$\begin{aligned} E'_i &= \{(\mathbf{e}'_k, r_k, s_k)\}, \quad r_k = i, \quad k = 1, \ldots, |E|, \\ V' &= \{(v'_i)\}, \quad i = 1, \ldots, |V|, \\ E' &= \bigcup_i E'_i = \{(e'_k, r_k, s_k)\}, \quad k = 1, \ldots, |E|, \end{aligned}$$

$\mathbf{v}_i$ is the node attributes vector, $\mathbf{e}_i$ is the edge attributes vector, $\mathbf{u}$ is the vector of the global attributes, s_k is the sender node index, r_k is the receiver node index. Sender and receivers node indexes were introduced for keeping an edges direction during graph processing.

For our purposes we use a combination of all those functions which is called "Full Graph Network Block" and shown in Fig. 3.

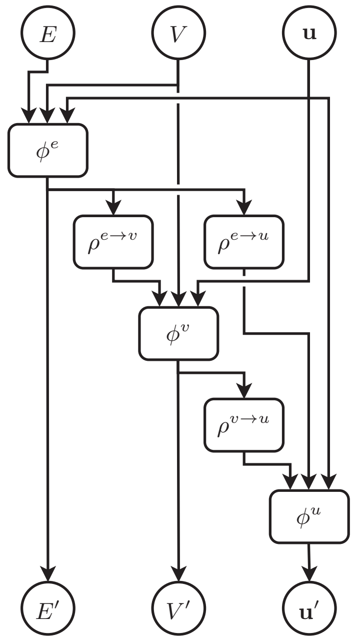

Fig. 3. Full Graph Network Block configuration

As followed from the scheme a graph network outputs a graph with the same structure and different attributes. It is clear that the idea is related to autoencoders and these two approaches can be combined.

IV. The model

Previously we tried to use metrics-based approach for classifying software source code [25], but it is good for software evaluation and cannot be generalized to program synthesis. So, as mentioned in the previous section our model is based on combination of several full graph network blocks. We use recurrent graph network architecture proposed in [8] (Fig. 4).

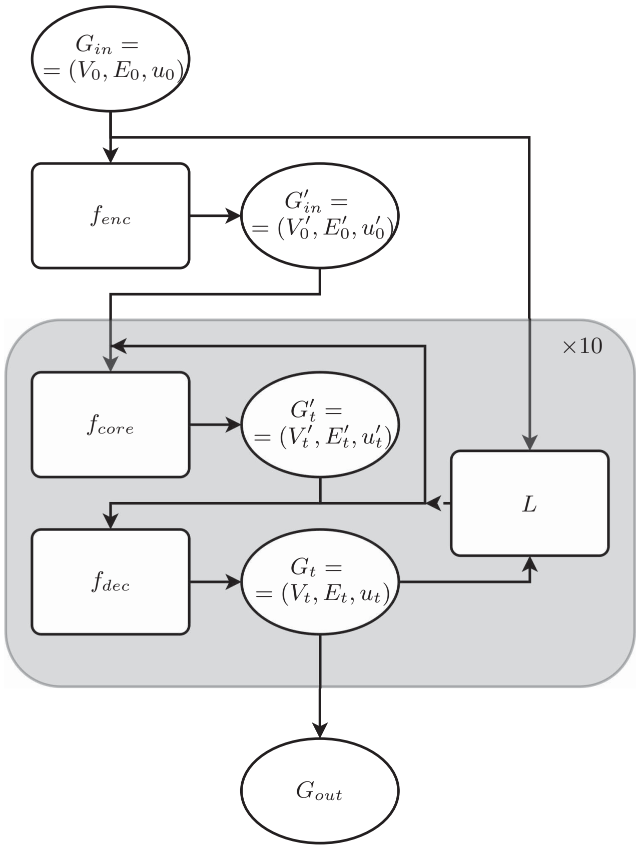
$G_{in} = (V_0, E_0, u_0)$
f_{enc}
$G'_{in} = (V'_0, E'_0, u'_0)$
$\times 10$
f_{core}
$G'_t = (V'_t, E'_t, u'_t)$
L
f_{dec}
$G_t = (V_t, E_t, u_t)$
G_{out}

Fig. 4. Recurrent Graph Network architecture

Formally on each step a model takes encoded initial graph and its latent representation and outputs a new latent representation and a graph with updated attributes:

$$\begin{aligned} V'_0, E'_0, \mathbf{u}'_0 &= f_{enc}(V_0, E_0, \mathbf{u}_0), \\ V_t, E_t, \mathbf{u}_t &= f_{dec}(V'_t, E'_t, \mathbf{u}'_t), \\ V'_{t+1}, E'_{t+1}, \mathbf{u}'_{t+1} &= f_{core}([V'_0, V'_t], [E'_0, E'_t], [\mathbf{u}'_0, \mathbf{u}'_t]), \end{aligned}$$

where f_{enc} is the encoder, f_{dec} is the decoder, f_{core} is the core processing graph network block.

The initial states V_0, E_0, and $\mathbf{u}_0$ are encoded as one-hot vectors, and the output of f_{dec} must be also one-hot vectors, so we apply $softmax$ function for each node, edge and global attributes.

In this paper we use 10 steps in recurrent graph network and analyze a latent representation after each one and the following vector dimensions:

$$\begin{aligned} V &\in R^{n\times 77}, \\ E &\in R^{e\times 20}, \\ \mathbf{u} &\in R^{1}, \\ V' &\in R^{n\times 16}, \\ E' &\in R^{e\times 16}, \\ \mathbf{u}' &\in R^{16}. \end{aligned}$$

V. Experiments

A. Dataset

For our experiments we use a dataset introduced in [7], [26]. It contains source code graph representation of 18 Java projects. Each graph in the dataset represents a Java class with augmented AST. There are 2754 graphs of Java classes in the dataset which are split to training set (2341 graphs) and test set (413 graphs).

Each node in graphs is described with 7 fields: node number, text, type, parent type, identifier, reference, and modifier. Each edge is described with 3 fields: type, sender node number, and receiver node number. We keep all fields for edges and use only number and type fields for nodes. There are 20 edge types and 77 node types in the dataset.

B. Training process

For training our model we use the standard dataset split. Both training and test parts were shuffled. We train our model as an autoencoder, so the goal is to minimize *cross entropy* between the input and predicted graph representations on each step. More formally:

$$\begin{aligned} &L(G_0, G_t) = \\ &\quad = \frac{1}{T}\sum_{t\in T}(-\sum_{v_t\in V_t}(v_0 \log v_t) - \sum_{e_t\in E_t}(e_0 \log e_t)) \to \min, \end{aligned}$$

where G_0 is the initial graph, G_t is the predicted graph on T steps, T is the number of steps (10 in our case), V_t is the set of nodes for the t-th graph, E_t is the set of edges for the t-th graph, v_0 is the attributes of a node in the initial graph, v_t is the attributes of a node in the predicted t-th graph, e_0 is the attributes of an edge in the initial graph, and e_t is the attributes of an edge in the predicted t-th graph.

Training process for 5000 iterations takes 3 days on one NVidia 1050ti GPU. We save metrics values for training and test sets each 5 minutes during training. The loss function and accuracy behaviour is shown on Fig. 5.

It could be seen from the graphics that the model successfully learned to reconstruct initial graphs during encoder-process-decoder pipeline. Loss function is exponentially decreases during training, but there are some outlier values. Both train and test loss values aim to 0 which shows stable training process without overfitting. The rate of nodes, edges, and entire graphs predicted correctly aims to 1, which shows the capability to reconstruct graphs from their latent representation.

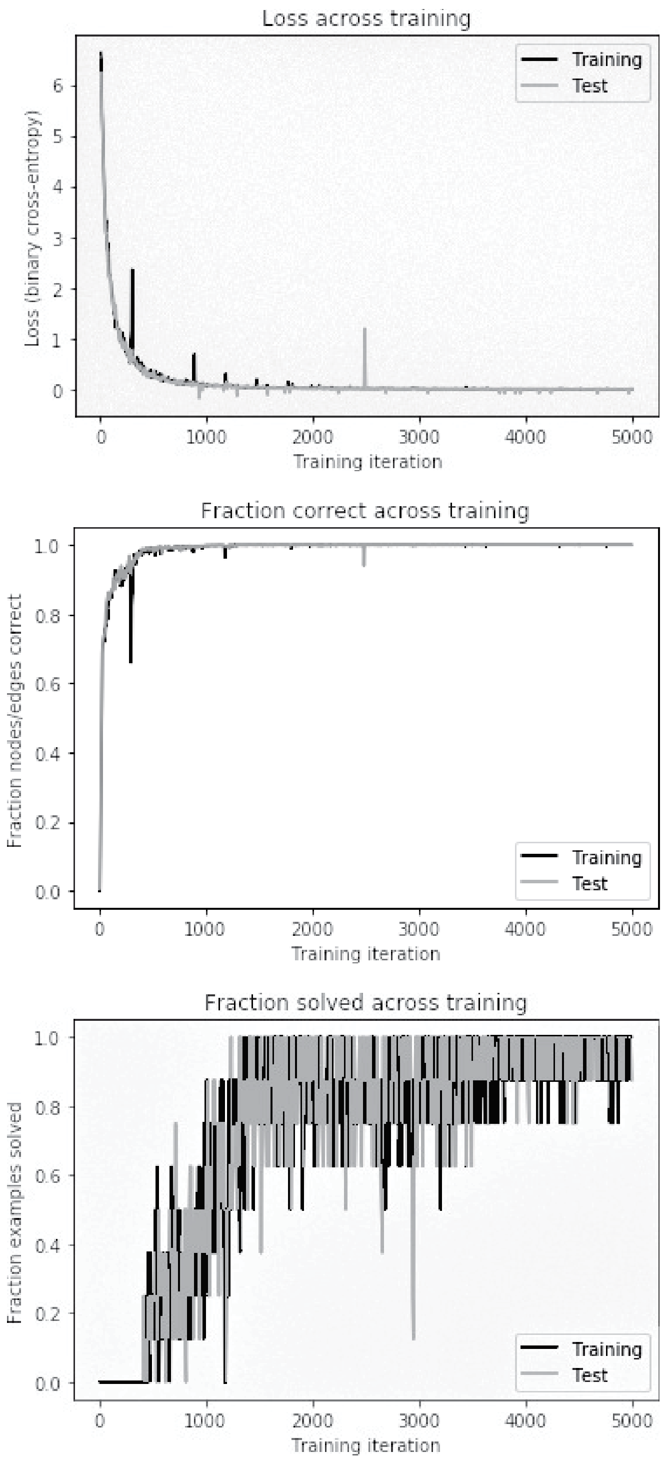

Fig. 5. Behaviour of loss function (top), correct nodes/edges accuracy (center), and solved graphs accuracy (bottom) during training process

C. Results

We apply K-Means algorithm to a graphs latent representation after each recurrent step in our model (Fig. 6). TSNE algorithm [27] is used for clear figures. After 10 recurrent steps two classes can be split with a linear model.

We calculated a mean vector of all nodes in graph, and a mean vector of all edges, after that both mean vectors were concatenated with global attributes vector, and a result vector was used for next processing:

$$\mathbf{v}^{(c)} = \frac{1}{|V|}\sum_{i=1}^{|V|}\mathbf{v}_i^{(c)}, \quad c = 1, \dots, C, \quad \mathbf{v} \in \mathbb{R}^C,$$

$$\mathbf{e}^{(c)} = \frac{1}{|E|}\sum_{i=1}^{|E|}\mathbf{e}_i^{(c)}, \quad c = 1, \dots, C, \quad \mathbf{e} \in \mathbb{R}^C,$$

$$\mathbf{l} = (\mathbf{v}, \mathbf{e}, \mathbf{u}), \quad \mathbf{l} \in \mathbb{R}^{3C},$$

where C is the number of components in vectors (16 in our case), $\mathbf{v}$ is the mean node vector, $\mathbf{e}$ is the mean edge vector, $\mathbf{l}$ is the vector for graph representation in a latent space.

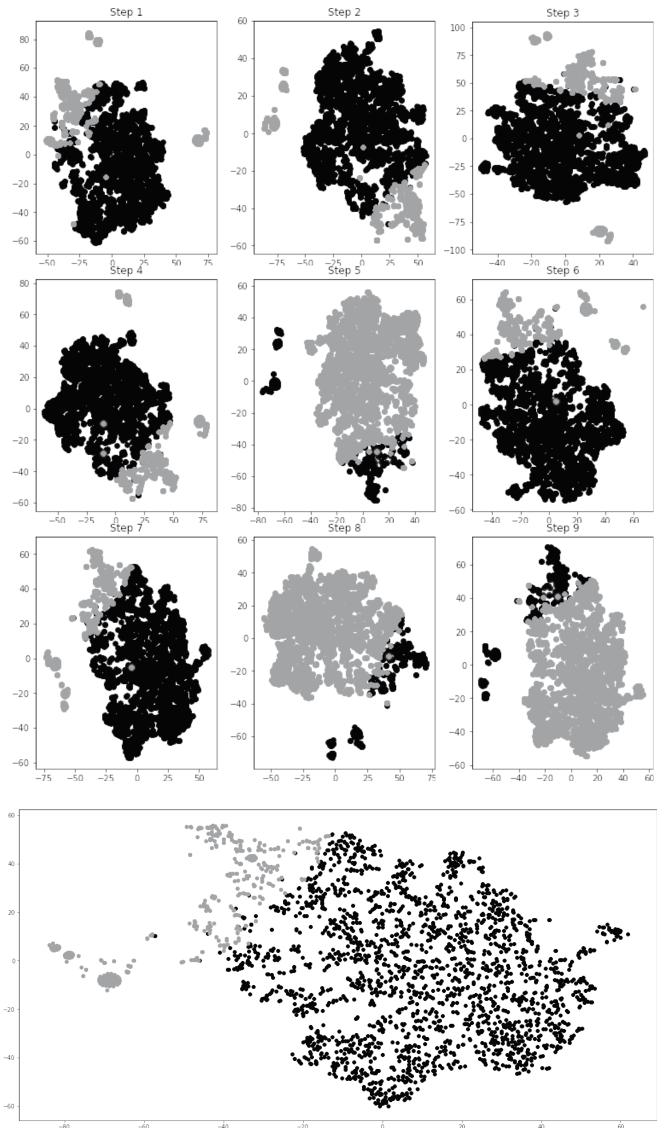

Fig. 6. The results of applying K-Means algorithm to graph representation in a latent space after each recurrent step in the model.

We find that two picked out with K-Means algorithm classes are corresponded to two separated graph types (Fig. 7).

The first group (which is bigger) contains graphs with large number of nodes and edges. The second part (which is smaller) contains graphs with small number of nodes and edges. Moreover graphs of the first type represents classes with some business logic implementation, and graphs of the second type represents interfaces and small utility classes.

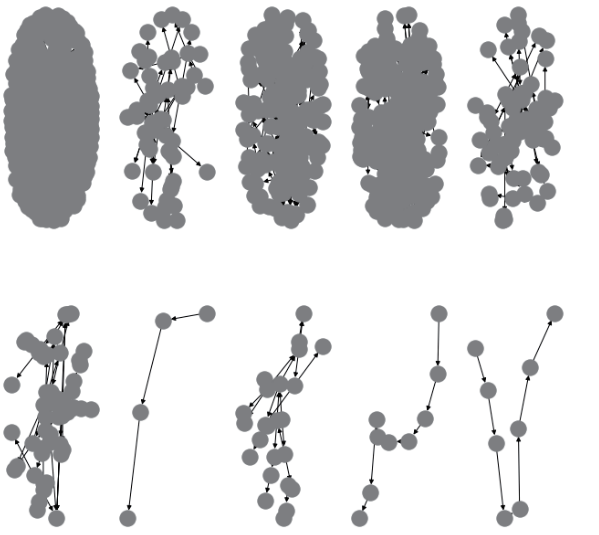

Fig. 7. The examples of graphs from black (top) and light grey (bottom) classes gotten after 10 processing steps.

VI. Conclusion

In this paper the approach to unsupervised classifying software source code is proposed, In its bounds we trained a graph neural network based autoencoder which can transform an initial augmented AST of classes into a representation in a latent space so they can be linearly split. Our experiments showed that the model can separate interface and small utility classes from classes with some business logic implementation.

Those results shows that the model can be used as a part of program synthesis and software source code static analysis systems. It is possible to use an obtained latent space for generating augmented ASTs as a step of program synthesis process, or to get an additional feature vector for source code snippets which will characterize its type.

The further work can be done in several directions. Firstly, it is possible to change a classical autoencoder to a variational autoencoder (VAE) and its modifications [28], [29]. As followed from their definition groups of data can be more clearly separated in a latent space because they consider a point neighbourhood on a decoder step, not a point itself. Secondly, taking in account note text can bring to a model additional information about parts of source code because constants, variable, function, and class names have own meaning which can characterize the analyzed source code snippet. And the third step is more deeply analysis of a gotten latent space for clustering classes according their implemented business logic.

Acknowledgment

The work was published with the financial support of the Russian Foundation for Basic Research, grant 17-57-52011 MNT_a.

References

[1] Min-je Choi, Sehun Jeong, Hakjoo Oh, Jaegul Choo, “End-to-End Prediction of Buffer Overruns from Raw Source Code via Neural Memory Networks”, *IJCAI 2017*.

[2] Jacob A. Harer, Louis Y. Kim, Rebecca L. Russell, Onur Ozdemir, Leonard R. Kosta, Akshay Rangamani, Lei H. Hamilton, Gabriel I. Centeno, Jonathan R. Key, Paul M. Ellingwood, Marc W. McConley, Jeffrey M. Opper, Peter Chin, Tomo Lazovich, “Automated software vulnerability detection with machine learning”, *IWSPA 2018*.

[3] Petr Vytovtov, Kirill Chuvilin, “Prediction of Common Weakness Probability in C/C++ Source Code Using Recurrent Neural Networks”, *FRUCT 21*

[4] Ke Wang, Rishabh Singh, Zhendong Su, “Dynamic Neural Program Embedding for Program Repair”, *ICLR 2018*.

[5] Miltiadis Allamanis, Marc Brockschmidt, Mahmoud Khademi, “Learning to Represent Programs with Graphs”, *ICLR 2018*.

[6] Uri Alon, Omer Levy, Eran Yahav, “code2seq: Generating Sequences from Structured Representations of Code”, *ICLR 2019*.

[7] Milan Cvitkovic, Badal Singh, Anima Anandkumar, “Deep Learning On Code with an Unbounded Vocabulary”, *CAV 2018*

[8] Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Caglar Gulcehre, Francis Song, Andrew Ballard, Justin Gilmer, George Dahl, Ashish Vaswani, Kelsey Allen, Charles Nash, Victoria Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matt Botvinick, Oriol Vinyals, Yujia Li, Razvan Pascanu, “Relational inductive biases, deep learning, and graph networks”, 2018, preprint arXiv:1806.01261 [cs.LG]

[9] J. MacQueen, “Some methods for classification and analysis of multivariate observations”, *In Proc. 5th Berkeley Symp. on Math. Statistics and Probability*, 1967, pp. 281297.

[10] V. P. Ivannikov, A. A. Belevantsev, A. E. Borodin, V. N. Ignatiev, D. M. Zhurikhin, A. I. Avetisyan, “Static analyzer Svace for finding of defects in program source code”, in Proc. of ISP RAS, 2014, pp. 231-250.

[11] The LLVM Compiler Infrastructure Project, Web: https://llvm.org/

[12] The History of the !exploitable Crash Analyzer — Security Research & Defence, Web: https://blogs.technet.microsoft.com/srd/2009/04/08/the-history-of-the-exploitable-crash-analyzer/

[13] A.E. Borodin, A.A. Belevancev, “A static analysis tool Svace as a collection of analyzers with various complexity levels”, in Proc. of ISP RAS, 2015, pp. 111-132.

[14] PVS-Studio: Static Code Analysis for C, C++, C# and Java, Web: https://www.viva64.com/en/pvs-studio/

[15] Cppcheck — A tool for static C/C++ code analysis, Web: http://cppcheck.sourceforge.net/

[16] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, Charles Sutton, “A Survey of Machine Learning for Big Code and Naturalness”, *ACM Computing Surveys* 51(4), September 2017

[17] G.E. Hinton, R.R. Salakhutdinov. “Reducing the Dimensionality of Data with Neural Networks”, *Science* Vol. 313, iss. 5786., 2006, pp. 504507.

[18] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, ukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, Jeffrey Dean. “Google’s Neural Machine Translation System: Bridging the Gap between Human and Machine Translation”, 2016, preprint arXiv:1609.08144 [cs.CL]

[19] Xinyun Chen, Chang Liu, Dawn Song. “Tree-to-tree Neural Networks for Program Translation”, *NeurIPS 2018*

[20] Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, Michael D. Ernst. "NL2Bash: A Corpus and Semantic Parser for Natural Language Interface to the Linux Operating System", *LREC-2018*

[21] Scarselli, F., Gori, M., Tsoi, A., Hagenbuchner, M. & Monfardini, G. "The graph neural network model", *IEEE Transactions on Neural Networks*, vol. 20, no. 1, 2009, pp. 61-80.

[22] Michal Defferrard, Xavier Bresson, Pierre Vandergheynst. "Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering", *NIPS 2016*

[23] Thomas N. Kipf, Max Welling. Semi-Supervised Classification with Graph Convolutional Networks, *ICLR 2017*

[24] rusty1s/pytorch_geometric: Geometric Deep Learning Extension Library for PyTorch, Web: https://github.com/rusty1s/pytorch_geometric

[25] Petr Vytovtov, Evgeny Markov, "Source Code Quality Classification Based On Software Metrics", *FRUCT 20*

[26] mwcvitkovic/Open-Vocabulary-Learning-on-Source-Code-with-a-Graph-Structured-Cache–Code-Preprocessor: Library for preprocessing java source code into Augmented ASTs, as per the paper Open Vocabulary Learning on Source Code with a Graph-Structured Cache, Web: https://github.com/mwcvitkovic/Open-Vocabulary-Learning-on-Source-Code-with-a-Graph-Structured-Cache–Code-Preprocessor

[27] Laurens van der Maaten, L.J.P., Hinton, G.E. "Visualizing Data Using t-SNE". *Journal of Machine Learning Research*, 9, Nov. 2008, pp. 2579-2605.

[28] Diederik P. Kingma, Max Welling. "Auto-Encoding Variational Bayes". *ICLR 2014*

[29] Ilya Tolstikhin, Olivier Bousquet, Sylvain Gelly, Bernhard Schoelkopf. "Wasserstein Auto-Encoders". *ICLR 2018*