# Deep Metrics

1st Artyom Lobanov
*JetBrains Research*
*Higher School of Economics*

2nd Mikhail Shavkunov
*JetBrains Research*
*Higher School of Economics*

3rd Oleg Svidchenko
*JetBrains Research*
*Higher School of Economics*

***Abstract*—The most of the current state-of-the-art models are based on deep neural networks. In contrast to old and classic solutions of some problems that use manually crafted features, deep learning methods mostly work with a raw data. Even further, there are methods that targets to learn an embeddings of the raw data that then can be used to solve concrete tasks. When applying such embedding to a specific task, there is a question whether this embedding contains all sufficient information to solve a task or not.**

**In this work we investigate if we are able to reconstruct some popular and informative code metrics from learned embeddings. We show that modern code encoding algorithms struggle with an ability to reconstruct code metrics that are known to be important to solve some software engineering problems. We then sketch some naive ways to avoid this issue.**

***Index Terms*—ML4SE, Code Metrics, Code Encoding**

## I. Introduction

Machine learning is a fast-growing field. Every year papers suggest numerous new approaches to popular tasks. Often these new models outperform models which were claimed as state-of-the-art and so they gradually replace them in both research and industry areas. But are we sure that old models strictly worse than new ones? Or maybe they just have another point of view to data, which is worse, but useful nevertheless? In this paper we investigate this issue on the example of code representation task.

Software metrics were used almost since the origin of software engineering [1]. Software metrics are values computed with specific algorithms aimed to describe some aspects of source code fragment (method, class, file, package) or its evolution. They were applied to a diverse set of tasks like defect prediction [2]–[5], code smell detection [6]–[8], buggy commits detection [9]–[11], refactoring recommendation [12] etc. Nowadays deep learning models became state-of-the-art approaches for many code-related tasks [13], [14]. Wei et al. [15] compared several approaches to cross-project defect prediction and showed, that deep learning models outperform metric-based approaches. Hoang et al. [16] architecture based on CNN [17] which become new state-of-the-art at task of buggy commits detection. So are metric are useless now? To answer this question, we examined if deep learning models are able to extract the same information from source code as metrics extract. We chose several popular models (Code2Vec [18], GGNN [19] and CodeBERT [13]) and trained them to predict 31 metrics values. The best model reached $R^2$ score of $0.83$ on average. However, it turned out that the random forest on top of a bag-of-words [20] performs $4\%$ better in one of experiments.

Our main contributions are:

1) Cleaned and prepared dataset for metrics recovering task
2) Comparison of metrics recovery power of three popular deep learning models
3) Made our trained models and the source code for all evaluated models open source and publicly available

## II. Background

### A. Code metrics

Metrics is an important research field of software engineering [21]. Originally they were designed to describe software internal code quality, complexity, and efforts required to develop the project, product and process, but then becomes popular as features for source code. There are two types of metrics [8]: static and dynamic. Metrics of first type analyze structure of source code and possible scenarios of program execution. They don't require any input data to be calculated. *Lines of Code*, *Cyclomatic Complexity* and *Weighted Methods per Class* are examples of such metrics. In contrast, dynamic metrics are focused on execution process. They concern with how many statements are executed and what function calls are literally taking place. Such metrics depends on input data and may differ from run to run. Examples are *Function Point* and *Loose Class Coupling*. In our study, we consider metrics of the first type due to they commonly used in machine learning tasks. Also metrics have different levels depending on which code units (methods, classes, files, packages or projects) they designed for. Since particular tasks of machine learning in software engineering often focused on methods, we decided to limit ourself to method-level code metrics, for now.

### B. GGNN

Graph Neural Network [22] is a deep learning method that operates on data represented with a graph. It works as follows: we associate a state vector with each node in the graph and then run several iterations of message exchange rounds between adjacent nodes. These messages update state vectors of the nodes depending on the node types, their current state and types of the edges between nodes. Gated Graph Neural Network (GGNN) is a special modification of GNN which uses Gated Recurrent Unit [23] to update state vectors. GGNN demonstrated good results in recent studies related to semantic code analysis [24]–[26].

Graph representation of code snippet consists of its Abstract Syntax Tree (AST) with extra edges. Every variable usage is linked to the next its read and write operation, every leaf is linked to the next leaf and so on. The full list of edge types may be found in the paper [25].

### C. Code2vec

One of the common approaches to capture text semantics is to learn distributed continuous vectors known as embeddings. Inspired by their success in the various NLP tasks [27], [28], Alon et al. [18] introduced Code2Vec model for semantic labeling of code snippets.

Code2Vec utilizes input code snippets with a corresponding name, label or caption. In this way, labels express semantics, which model is learning. During the process of learning such semantics can be represented in code embeddings as similar method names are closer to each other. Furthermore, authors give us an example, that simple math operations on embeddings provide semantic similarities: *vector*(equals) + *vector*(toLowerCase) is close to *vector*(equalsIgnoreCase).

Prior works [29], [30] use an abstract syntax tree (AST) for enhanced code representations. In Code2Vec paper authors represent a code snippet as a bag of extracted paths from the corresponding AST. In this setting, the main problem is to learn a correlation between the bag of path-contexts and snippet labels, because even similar methods with similar logic could end up with different context paths. Authors proposing attention mechanism for context path aggregation, which yields the same label for close vectors.

Neural networks learn how much "attention" should be given to each context path. Attention can be used in a soft way, meaning that wights are distributed continuously between all AST paths, and the hard way, which refers to selection of a single path at a time. Code2Vec employs the former approach that showed more efficient code modeling.

### D. CodeBERT

Recent advances in Deep Neural Language Processing field showed that transformer neural network architecture [31] are able to perform well in variety of a complex NLP tasks [32]. Transformer neural network is a sequence-to-sequence model that consists of encoder and decoder. Encoder network yields a latent representation of an input sequence. Together with a query sequence, this representation used by decoder to reconstruct an output sequence.

The idea of a CodeBERT [13] algorithm is to apply a RoBERTa [33] model to both programming and natural languages in order to produce general-purpose sequence representations that allows to solve NL-PL understanding and generation tasks. Authors of CodeBERT use two objectives to train their model:

1) Masked Language Modeling. This objective refers to reconstructing a missing parts of code or it's documentation. This parts replaced by special `[MASK]` token and should be reconstructed in the output.
2) Replaced Token Detection. This is a classification problem that refers to detecting replaced parts of the sequence. At the training stage some tokens in an input sequence replaced by other tokens and should be classified by CodeBERT model as a replaced ones.

After training on the objectives described above, authors perform model finetuning on another objective that refers to finding a code part from a code parts collection that more semantically relate to provided documentation than other code parts in this collection. To do so, authors connect output sequence part that corresponds to `[CLS]` token to a dense layer with softmax activation.

In our work we will use a final model presented by authors with a embedding that corresponds to a `[CLS]` token as an embedding of a whole method. As CodeBERT model were finetuned by authors to a relevant code search objective, we expect that an embedding for `[CLS]` token should be a meaningful embedding for a whole code part that passed to an encoder of CodeBERT.

## III. Approach

### A. Dataset

Our dataset is based on data in the paper [34]. Dataset contains files before and after refactoring is performed. Total size of the dataset is 285 Gb. For our purposes, we extracted separately $535\,474$, $78\,026$, $151\,463$ files for train, validate and test samples respectively. Every file is extracted before refactoring to ensure refactoring should be done. Total size of our sampled dataset is 13.5 GB.

In the original paper, authors studied the effectiveness of supervised models in predicting potential refactoring changes. In this study, popular software metrics are treated as features. Our goal was to predict software metrics based on different code representations.

Our target metric was $R^2$. It is a popular regression quality metric, which may be conceptually described as the proportion of target value variance described by the model. More formally, it's described by formula below:

$$R^2 = 1 - \frac{\sum\limits_i (y_i - \hat{y})^2}{\sum\limits_i (y_i - \overline{y})^2}$$

where $y_i$ are original values, $\overline{y}$ is mean target value and $\hat{y}$ are predicted values.

We normalized all metrics by subtracting mean metric value and dividing by standard deviation. This transformation balanced the impact of metrics on the loss function and so improved the stability of training. Note that this transformation doesn't affects $R^2$ value.

### B. Baselines

As far as we know its the first paper on metrics recovering through deep learning models, so we couldn't compare our results with previous works. So we trained simple regression models to get at least some reference values. We used Bag-of-words [20] as the simplest way to represent code. Our

dictionary was limited to 2 000 of the most frequent tokens, all math symbols and brackets. Linear Regression and Random Forest Regression were used as baseline models. We used their implementations from the library *sklearn* [35].

### C. GGNN

We have no pretrained model for GGNN, so we train it from scratch. Original GGNN from Allamanis et al. [25] produces vector for each node. We aggregate all these vectors with attention mechanism to get vector for whole graph, and then apply fully-connected layer to predict metrics values. Dropout regularization was used to prevent overfitting. You may find our implementation of GGNN and graph parser at repository of our project.

### D. Code2Vec

For Code2Vec model we used a pretrained java14 model for evaluating embeddings. For each file an embedding was extracted for the refactoring method. Our aim was to evaluate how code semantics is captured in the code embeddings, thus, embeddings were used as features for prediction software metrics.

To do so, Multi Layer Perceptron, XGBoost gradient boosting decision trees, Random Forest and Lasso models are used. Each model is evaluated with Pearson correlation coefficient for each software metric as a target. Best results are obtained from Random Forest on Code2Vec embeddings and presented in Tables I, II.

### E. CodeBERT

For CodeBERT algorithm we use pretrained model provided by authors of CodeBERT [13]. It were trained to produce a meaningful embedding in token `[CLS]` position as described in the previous section, However, their model have a limitation to a maximum number of token in the input sequence and not all of the code of methods match this requirements. To leverage this obstacle, we split code into a parts that model can encode and then obtain a sequence of this parts embeddings. To solve regression model, we first apply dense model to each embedding separately and then use simple attention with multiple heads to produce an embedding of a fixed size. This embedding is further processed by dense network that yields a metrics' value prediction.

## IV. Results

In order to test our model we define a two sets of metrics that we want to reconstruct. The key difference between them is a level of an abstraction and information that this metrics represent. For example, *Cyclomatic Complexity* is a high level metric that measures how complex the code are. In contrast, number of loops is a very simple metric with low level of abstraction.

The first set of metrics will be refereed as a simple metrics set. It consists of a metrics from common metrics with low abstraction level such as *LOC* (lines of code), `assignments` (count of an assignment statements), etc. Intuitively, as most of them may be computed from source code by a simple algorithms, they should be well recoverable by trainable models.

In Table I we can see the results of evaluation of models on simple metrics dataset. LASSO and RF are referring to a corresponding methods trained on bag-of-words representation, C2V is a best result for Code2Vec model, CodeBERT corresponds to a pretrained CodeBERTa model with a simple dense neural network on top of it. The best model (GGNN) reach $R^2$ score of $0.83$ which is not impressive result. It's interesting, that Random Forest outperforms such deep learning approaches as code2vec and CodeBERT. That may mean that original pretrained models lose more metric-related information than Bag-of-Words.

TABLE I
R2 REGRESSION SCORE FOR A SIMPLE METRICS SET.

| **Metric** | LASSO | RF | GGNN | C2V | CodeBERT |
|---|---|---|---|---|---|
| **Mean** | 0.44 | 0.74 | **0.83** | 0.49 | 0.69 |
| AnonymousClasses | 0.09 | 0.70 | **0.80** | 0.49 | 0.50 |
| Assignments | 0.82 | **0.91** | 0.86 | 0.55 | 0.86 |
| Cbo | 0.29 | 0.64 | 0.77 | 0.46 | **0.82** |
| Comparisons | 0.49 | 0.71 | **0.89** | 0.51 | 0.56 |
| Loc | 0.90 | **0.93** | 0.87 | 0.55 | 0.89 |
| Loop | 0.29 | 0.81 | **0.91** | 0.44 | 0.72 |
| MathOperations | 0.38 | 0.72 | **0.87** | 0.51 | 0.52 |
| MaxNestedBlocks | 0.30 | 0.86 | 0.91 | 0.41 | **0.95** |
| Numbers | 0.31 | 0.53 | **0.87** | 0.56 | 0.46 |
| Parameters | 0.02 | 0.41 | 0.93 | 0.61 | **0.95** |
| ParenthesizedExp | 0.19 | 0.36 | **0.62** | 0.37 | -0.11 |
| Return | 0.60 | 0.71 | **0.88** | 0.74 | 0.62 |
| methodRfc | 0.60 | 0.87 | 0.80 | 0.44 | **0.91** |
| StringLiterals | 0.51 | 0.65 | **0.77** | 0.51 | 0.40 |
| TryCatch | 0.17 | **0.92** | 0.89 | 0.41 | 0.77 |
| UniqueWords | 0.57 | 0.84 | 0.78 | 0.44 | **0.91** |
| Variables | 0.71 | 0.86 | **0.88** | 0.56 | 0.85 |
| Wmc | 0.73 | **0.87** | 0.85 | 0.58 | 0.81 |

The second set of metrics will be refereed as a complex metrics set. In contrast with simple metrics set, complex metrics set includes metrics that tries to measure a high-level properties of a source code (e.g. maintainability, reliability, etc.). Intuitively, it is should be harder to predict value of such complex metrics than simple ones.

Results for that experiment are shown in Table II. Surprisingly, Random Forest showed very good result and outperformed all other models. However, GGNN reached the best $R^2$ score for majority metrics ($0.83$ on average). All the more interesting is why it has such poor results for metrics *HalsteadEffort* and *ControlDensity*, and why Code2Vec is so good at *HalsteadEffort* predicting. These anomalies may be good directions for further research.

## V. Threats to Validity

We used dataset of code snippets written in Java and can't guarantee the same results for other languages. However, Java is a classical Object-Oriented language and we expect to observe the similar results for related languages such as C++, C# or Kotlin.

We used only 31 examples from numerous metrics designed by humanity for tens of years. But 18 metrics were taken from

TABLE II
R2 REGRESSION SCORE FOR A COMPLEX METRICS SET.

| Metric | LASSO | RF | GGNN | C2V | CodeBERT |
|---|---|---|---|---|---|
| **Mean** | 0.68 | **0.87** | 0.83 | 0.59 | N/A |
| CyclomaticCompl | 0.75 | 0.89 | **0.90** | 0.56 | N/A |
| HalsteadDiff | 0.59 | **0.81** | 0.77 | 0.52 | N/A |
| DesignComplexity | 0.70 | **0.87** | 0.84 | 0.53 | N/A |
| HalsteadEffort | 0.59 | 0.78 | 0.62 | **0.90** | N/A |
| HalsteadVolume | 0.82 | 0.89 | **0.91** | 0.57 | N/A |
| HalsteadBugs | 0.72 | 0.83 | **0.87** | 0.71 | N/A |
| HalsteadLength | 0.83 | 0.92 | **0.94** | 0.54 | N/A |
| HalsteadVocab | 0.65 | **0.90** | 0.88 | 0.46 | N/A |
| EssentialCycComp | 0.53 | 0.75 | **0.83** | 0.73 | N/A |
| ControlDensity | 0.07 | **0.96** | 0.45 | 0.45 | N/A |
| QCPCorrectness | 0.84 | **0.92** | **0.92** | 0.57 | N/A |
| QCPMaintainable | 0.87 | 0.92 | **0.94** | 0.55 | N/A |
| QCPReliability | 0.87 | 0.93 | **0.94** | 0.55 | N/A |

recent metrics-based machine learning paper and others from initial set of metrics of popular plugin for IDE, so we think these metrics are good representatives of actually used metrics.

We haven't compared our results with previous works due to we haven't found attempts to solve the same task, but we used diverse set of models, which used different ideas behind code representation (CodeBERT treat code as sequence of tokens, GGNN as graph, code2vec as set of path and baseline models as bag-of-words).

Overall, while these threats are important to note, we believe that we mitigated them well and they do not invalidate the results of our research.

## VI. CONCLUSION

In our experiments, we clearly showed that GGNN, Code2Vec and CodeBERT lose some common information (that can be represented as code metrics) about the code as they performance was far from ideal values.

This finding is a highly important because most of the classic methods that applying machine learning to a software engineering tasks use those metrics to improve their results and hence show that information given by this metrics are sufficient to get a good solution. Therefore at some point this lack of ability to capture this common information about code may become a problem that will prevent a models from further improving.

In this case natural solution would be is to combine metrics and learned representations in order to improve quality of resulting models. Another option may be is to investigate how can we incorporate code metrics into encoding models so that final embeddings will capture code metrics information. Also we can use task of metrics recovering at deep learning models pretraining stage.

Our implementation of models and training scripts with hyper-parameters details may be found at our publicly available repository at GitHub[1]. Our dataset and preprocessed files are stored here[2].

[1]https://github.com/shavkunov/MLSE

[2]https://doi.org/10.5281/zenodo.4395267

## REFERENCES


[1] N. E. Fenton and M. Neil, "Software metrics: successes, failures and new directions," *Journal of Systems and Software*, vol. 47, no. 2-3, pp. 149–157, 1999.

[2] S. Hosseini, B. Turhan, and D. Gunarathna, "A systematic literature review and meta-analysis on cross project defect prediction," *IEEE Transactions on Software Engineering*, vol. 45, no. 2, pp. 111–147, 2017.

[3] K. Gao, T. M. Khoshgoftaar, H. Wang, and N. Seliya, "Choosing software metrics for defect prediction: an investigation on feature selection techniques," *Software: Practice and Experience*, vol. 41, no. 5, pp. 579–606, 2011.

[4] E. Kocaguneli, A. Tosun, A. B. Bener, B. Turhan, and B. Caglayan, "Prest: An intelligent software metrics extraction, analysis and defect prediction tool." in *SEKE*, 2009, pp. 637–642.

[5] B. Turhan and A. B. Bener, "Software defect prediction: Heuristics for weighted naïve bayes." in *ICSOFT (SE)*, 2007, pp. 244–249.

[6] F. A. Fontana, V. Ferme, M. Zanoni, and A. Yamashita, "Automatic metric thresholds derivation for code smell detection," in *2015 IEEE/ACM 6th International Workshop on Emerging Trends in Software Metrics*. IEEE, 2015, pp. 44–53.

[7] F. Pecorelli, F. Palomba, D. Di Nucci, and A. De Lucia, "Comparing heuristic and machine learning approaches for metric-based code smell detection," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 93–104.

[8] S. Kaur and R. Maini, "Analysis of various software metrics used to detect bad smells," *Int J Eng Sci (IJES)*, vol. 5, no. 6, pp. 14–20, 2016.

[9] M. Nayrolles and A. Hamou-Lhadj, "Clever: combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects," in *Proceedings of the 15th International Conference on Mining Software Repositories*, 2018, pp. 153–164.

[10] C. Rosen, B. Grawi, and E. Shihab, "Commit guru: analytics and risk prediction of software commits," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 966–969.

[11] S. Kim, E. J. Whitehead, and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 181–196, 2008.

[12] A. S. Nyamawe, H. Liu, Z. Niu, W. Wang, and N. Niu, "Recommending refactoring solutions based on traceability and code metrics," *IEEE Access*, vol. 6, pp. 49 460–49 475, 2018.

[13] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[14] J. Wei, M. Goyal, G. Durrett, and I. Dillig, "Lambdanet: Probabilistic type inference using graph neural networks," *arXiv preprint arXiv:2005.02161*, 2020.

[15] D. Chen, X. Chen, H. Li, J. Xie, and Y. Mu, "Deepcpdp: Deep learning based cross-project defect prediction," *IEEE Access*, vol. 7, pp. 184 832–184 848, 2019.

[16] T. Hoang, H. K. Dam, Y. Kamei, D. Lo, and N. Ubayashi, "Deepjit: an end-to-end deep learning framework for just-in-time defect prediction," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 34–45.

[17] Y. Kim, "Convolutional neural networks for sentence classification," *arXiv preprint arXiv:1408.5882*, 2014.

[18] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[19] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *arXiv preprint arXiv:1711.00740*, 2017.

[20] Z. S. Harris, "Distributional structure," *Word*, vol. 10, no. 2-3, pp. 146–162, 1954.

[21] G. Khurana and S. Jindal, "A model to compare the degree of refactoring opportunities of three projects using a machine algorithm," *Advanced Computing*, vol. 4, no. 3, p. 17, 2013.

[22] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph neural networks: A review of methods and applications," *arXiv preprint arXiv:1812.08434*, 2018.

[23] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[24] M. Allamanis, E. T. Barr, S. Ducousso, and Z. Gao, "Typilus: neural type hints," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 91–105.

[25] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *ArXiv*, vol. abs/1711.00740, 2018.

[26] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," *arXiv preprint arXiv:1511.05493*, 2015.

[27] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *CoRR*, vol. abs/1301.3781, 2013.

[28] J. Pennington, R. Socher, and C. Manning, "GloVe: Global vectors for word representation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1532–1543. [Online]. Available: https://www.aclweb.org/anthology/D14-1162

[29] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," *SIGPLAN Not.*, vol. 49, no. 6, p. 419–428, Jun. 2014. [Online]. Available: https://doi.org/10.1145/2666356.2594321

[30] G. A. Aye and G. Kaiser, "Sequence model design for code completion in the modern ide," *ArXiv*, vol. abs/2004.05249, 2020.

[31] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[32] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[33] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[34] M. Aniche, E. Maziero, R. Durelli, and V. Durelli, "The effectiveness of supervised machine learning algorithms in predicting software refactoring," *arXiv preprint arXiv:2001.03338*, 2020.

[35] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.