SOURCE CODE ANOMALY DETECTION IN SEMANTIC REPRESENTATION

A PREPRINT

Akimova E.N.*∗†*

[aen15@yandex.ru](mailto:aen15@yandex.ru)

**Bersenev A.Yu.***∗ †*

[Alexander.Bersenev@urfu.ru](mailto:Alexander.Bersenev@urfu.ru)

**Deikov A.A.***∗†*

[hx0day@hackerdom.ru](mailto:hx0day@hackerdom.ru)

Kobylkin K.S.*∗†*

[kobylkinks@gmail.com](mailto:kobylkinks@gmail.com)

Konygin A.V.*∗*

[konygin@imm.uran.ru](mailto:konygin@imm.uran.ru)

Mezentsev I.P.*∗†*

[ilyamezentcev@gmail.com](mailto:ilyamezentcev@gmail.com)

Misilov V.E.*∗†*

[v.e.misilov@urfu.ru](mailto:v.e.misilov@urfu.ru)

**ABSTRACT**

In this work, we present a new unsupervised approach for predicting cross-project code defects. Code defect prediction is formulated as an anomaly detection problem. In our approach, we use the implicit semantic representation of a source code and the variational autoencoder for anomaly detection. To test the proposed approach, the implemented model was trained and evaluated on the Py150 data set.

***Keywords*** Defect prediction *·* Semantic anomaly detection *·* Code understanding *·* Variational autoencoder.

# Introduction

Let us assume that an *abnormal* behavior is a program's state, which is *not* derived from the program's original objective. In other words, the state is not a consequence of the objective for which the program was developed. The reasons this may happen include the following:

* + The program code does not match the objective.
  + The execution environment does not match the objective.
  + The output does not match the objective.

This work focuses on the program code. If we want to predict a program's abnormal state, we need to understand that the code does not correspond to the objective.

As a rule, objectives differ greatly from each other; however, we can discover similarities by decomposing them into small tasks. The finer the decomposition, the more efficiently similar tasks are combined into groups.

Following this logic, we have to split the source code into fragments (code blocks), and find atypical ones among them. We refer to these atypical fragments as *anomalous*, or simply, *anomalies*. However, it is worth noting that some constraints exist. First, anomalous code does not always contain a defect, meaning sometimes it implements rare (but accurate) logic. Second, the defect cannot be localized in one fragment, and several fragments need to be edited to correct it. At first glance, this issue can be resolved by increasing the size of the fragments; however, this approach also has specific constraints. That said, defect prediction using anomaly detection is reasonable. First, code defects occur due to violations of the programming process (misunderstanding, switching attention, etc.). We can view all these as anomalies in the development process (see [[1]).](#_bookmark14) Second, most code is defect free. As such, a strong imbalance of classes (defective and nondefective code) implies that with a suitable representation, defective code may effectively be found to be anomalous (implausible in terms of the distribution of normal code).

*∗*N.N. Krasovskii Institute of Mathematics and Mechanics

*†*Ural Federal University, Ekaterinburg, Russia

In this article, we present a new approach to cross-project prediction of source code defects.

Code defect prediction is posed as a problem that involves detecting anomalies in a semantic multilingual representation of the source code, and training occurs in an unsupervised mode using code from different repositories. The proposed approach is evaluated on the test part of the Py150 data set.

In Section [2,](#_bookmark0) we discuss existing studies and tools that apply the anomaly detection techniques for source code. Section [3](#_bookmark1) presents our proposed approach to anomaly detection, while Section [4](#_bookmark10) details the results. To conclude, Section [5](#_bookmark13) summarizes these results and provides possible directions for future work.

# Background and related work

One of the issues involved with the defect prediction problem is the imbalance of classes. Furthermore, it is both time consuming and labor intensive to create a big labelled data set. Therefore, while taking into account the progress made in machine learning, defect code prediction can be posed as an anomaly detection problem. In this case, the code representation approach plays a decisive role. Simply put, all approaches fall into two groups (each with advantages and disadvantages): *explicit* and *implicit* features. Explicit (or *traditional*) features are mostly software metrics, and are detailed in [[1,](#_bookmark14) Appendix]; whereas implicit features are generated automatically. (Note that Section 3.2 covers code representation.)

Currently, defect code prediction using anomaly detection is becoming increasingly popular. In our opinion, this is due to the progress in deep learning. In [[12]](#_bookmark26), the authors use explicit features and Gaussian distribution to model nondefective code, which is then predicted using the deviation from the model estimation. The possibility of using explicit features and *k*-nearest neighbors algorithm for anomaly detection in source code is demonstrated in [[11].](#_bookmark25) In [[17]](#_bookmark31), the stacked denoising autoencoders are used to extract deep representations from the traditional software metrics. A similar idea is implemented in [[15]](#_bookmark29) and [[1],](#_bookmark14) where the autoencoder on the explicit features is used for within-project source code anomaly detection. In [[4]](#_bookmark18), two approaches for code vector representation are implemented: a feature vector consisting of 51 explicit code metrics and an implicit *N* -gram approach. Additionally, several anomaly detection techniques are tested: local outlier factor, isolation forest, and autoencoder neural network. In the study, the authors focus on two types of anomalies: syntax tree anomalies and compiler-induced anomalies in Kotlin code.

In this work, we use semantic contextual multilingual embeddings together with a variational autoencoder to detect anomalies in Python code. To the best of our knowledge, we are the first to propose such a model, which is detailed in the next section.

# The proposed approach

We use an unlabeled source code data set, which leads us to unsupervised anomaly detection methods.

The proposed approach is as follows: Calculate the semantic contextual multilingual embeddings to represent code blocks. Then, use the variational autoencoder as an anomaly detector. A variational autoencoder not only allows you to obtain an estimate for the code anomality, but also a probabilistic model. Additionally, the variational autoencoder can be combined with the neural network used for code representation, making it possible to fine-tune the model in the end-to-end mode.

There is a certain probability that the reconstruction error of an unseen sample of code blocks will belong to the negative reconstruction error distribution, or belong to the positive reconstruction error distribution. Based on those probabilities, we classify the samples as nondefective or defective.

The proposed approach for code anomaly detection at a scale consists of the following steps:

1. Retrieve a large representative corpora of source code.
2. Extract features (calculate embeeddings).
3. Run anomaly detection.
4. Process the output of the anomaly detection algorithm.

## Data set

Our current work uses the Py150 data set (see [[13])](#_bookmark27), which consists of the Python programs collected from GitHub[3](#_bookmark4) repositories by removing duplicate files, removing project forks (copy of another existing repository), and keeping only programs that parse and have a maximum of 30,000 nodes in the AST (abstract syntax tree). Furthermore, only repositories with permissive and non-viral licenses (such as MIT, BSD, and Apache) are used. The data set is split into two parts — 100,000 files used for training and 50,000 files used for evaluation.

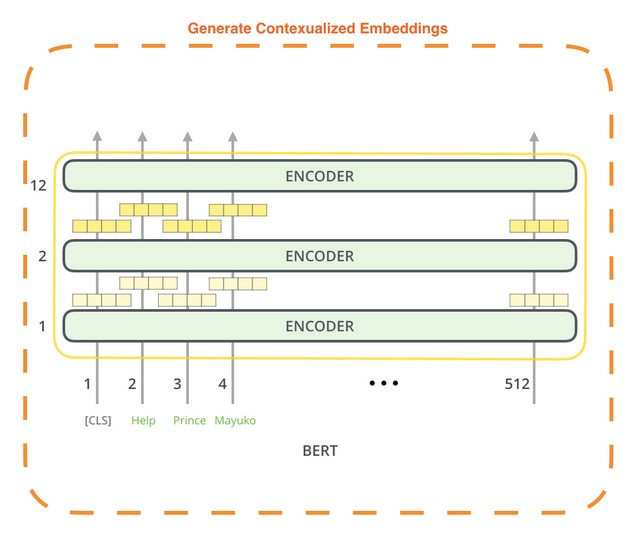
To meet our objectives, we needed to extract code blocks from Py150. Code blocks, in our case, are the functions we found with the ast[4](#_bookmark5) package. Each of these code blocks corresponds to the node in AST and has type ast.FunctionDef or ast.AsyncFunctionDef. As a result, we obtained approximately one million code blocks — inputs for the proposed model.

We use the train/test split for code blocks induced by the original train/test split for Python programs in Py150.

## Code representation

The next step is to calculate vector representations (embeddings) for the code blocks. This can be done through numerous methods, for example, code2vec, code2seq, PathPair2Vec, CuBERT, and CodeBERT (see [[3],](#_bookmark15) [[2],](#_bookmark16) [[16],](#_bookmark30) [[7],](#_bookmark21) [[6])](#_bookmark20). The proposed approach uses the CodeBERT model, which is a bimodal pre-trained model for natural language and the following programming languages: Python, Java, JavaScript, PHP, Ruby, and Go. The model follows BERT (see [[5])](#_bookmark19) and RoBERTa (see [[10]),](#_bookmark24) and uses the multi-layer bidirectorial Transformer (see [[18])](#_bookmark32) as the model architecture (see Figure [1](#_bookmark3)[5](#_bookmark6)).

Figure 1: BERT



CodeBERT captures the semantic connection between natural language and programming language, and produces general-purpose representations that can broadly support NL-PL understanding tasks (for example, natural language code search) and generation tasks (for example, code documentation generation). CodeBERT is trained on two objectives: Masked Language Modeling and Replaced Token Detection. Due to the architecture and learning method, CodeBERT allows you to obtain contextual semantic embeddings. As a result, the representation of a fragment depends more on semantics and less on syntax. In this work, we used a pre-trained model from the Huggingface[6](#_bookmark7). It is based on RobertaModel[7](#_bookmark8) — a model proposed in [[10].](#_bookmark24) The resulting embedding is the 768-dimensional outputs from the penultimate layer of the Transformer. We experimented with other layers, but the penultimate layer proved to be the most suitable. Specifically, it contains high-level information but is not as well trained as the last layer for the pre-training tasks.

3https://github.com/

4https://docs.python.org/3/library/ast.html

5The source of the image is the article <http://jalammar.github.io/illustrated-bert/>

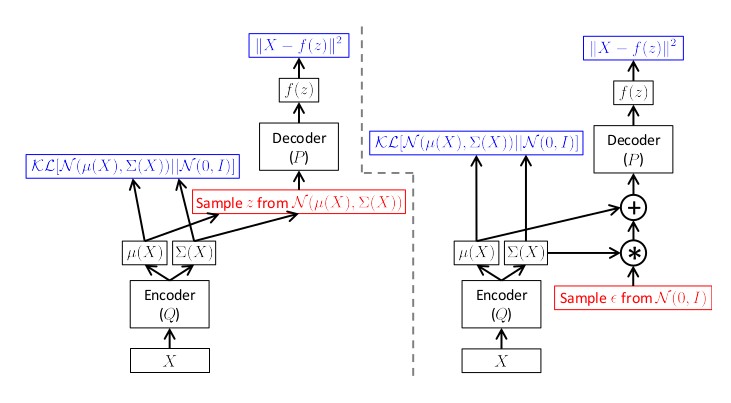
6https://huggingface.co/microsoft/codebert-base

7https://huggingface.co/transformers/model doc/roberta.html

## Anomaly detection

For anomaly detection, the model uses variational autoenconder (see [[9,](#_bookmark23) [14]).](#_bookmark28) In the variational autoencoder, the loss function is composed of two parts: *L*2-loss (generative loss) and Kullback — Leibler divergence (latent loss, see [[9,](#_bookmark23) Appendix B]). Unlike a simple autoencoder, a variational autoencoder can approximate by virtue of Bayesian Inference. This means that a variational autoencoder is more appropriate than a simple autoencoder for extrapolation tasks. Figure [2](#_bookmark9)[8](#_bookmark12) is a diagram of the variational autoencoder .

Figure 2: Variational autoencoder



The variational autoencoder describes the latent space in terms of probability distribution. Unlike other autoencoders The encoding already learned by the autoencoder describes a sample draw from some latent space, determined by the encoder. Unlike other autoencoders that represent each value of the encoding with a single value, the variational autoencoder learns to represent it as latent distributions.

The parameters of our variational autoencoder were learned with respect to the Gaussian distribution, with the hidden state size being 20.

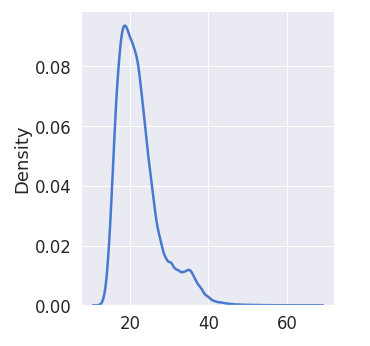
# Results

The Py150 train/test split provides approximately 641,000 data points in training and approximately 314,000 data points in testing. Every data point is represented by 768-dimensional embedding. Training was not required to obtain such a representation since we were already using a model with pre-trained weights. However, the proposed approach leaves room to fine-tune the code representation model for downstream tasks.

In the next step, we trained the variational autoencoder. Training occurred over 200 epochs, as an optimization algorithm was used Adam (see [[8])](#_bookmark22) with a learning rate of 0*.*001 and batch size of 128.

The resulting model allows for the prediction of code block anomalousness, which is provided by the reconstruction loss of the variational autoencoder. The distribution of reconstruction losses is given in Figure [3.](#_bookmark11)

Figure 3: Reconstruction loss on test data



8The source of the image is the preprint C. Doersch “Tutorial on variational autoencoders”, 2016

Therefore, the model outputs corresponding to a large reconstruction error (right-hand side) are anomalies, that is, atypical examples of code blocks. The following are the top five examples:

# 1 .

d e f f 1 ( ) :

x = ”ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ” y = ”ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ” z = ”ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ”

x += y + z

# 2 .

d e f f 2 ( ) :

x = ”ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ” y = ”ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ” z = ”ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ”

x = x + y + z

# 3 .

d e f a ( ) :

# 4 .

p a s s

d e f f 3 ( ) :

x = ”ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ” y = ”ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ” z = ”ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ”

x = ” ” . j o i n ( ( x , y , z ) )

# 5 .

d e f f o o ( ) :

” ” ”

” ” ” 1

d o c s t r i n g

Other examples of the most anomalous code blocks from Py150 with reconstruction loss of over 30*.*0 are given here[9](#_bookmark17).

# Conclusion

We present a new approach to cross-project prediction of source code defects. Code defect prediction is posed as a problem that involves detecting anomalies using a variational autoencoder for the semantic multilingual representation of the source code. Training occurs in an unsupervised mode using code from different projects, and the variational autoencoder can be combined with the neural network used for code representation. This makes it possible to fine-tune the model in the end-to-end mode.

In our future work, we aim to train a multilingual variational autoencoder, which in turn will lead to a multilingual anomaly detection model. Additionally, we aim to use the generative abilities of out variational autoencoder to interpret the detected anomalies, and plan to make the model publicly available.

This research was supported by Huawei, and special thanks go to the System Programming Laboratory (Huawei RRI). Computations were performed on the Uran supercomputer at the IMM UB RAS.

# References

[1] Petar Afric et al. “REPD: Source code defect prediction as anomaly detection”. In: *Journal of Systems and Software* 168 (2020), p. 110641. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2020.110641. URL: <http://www.sciencedirect.com/science/article/pii/S0164121220301138>.

[2] Uri Alon et al. “code2seq: Generating Sequences from Structured Representations of Code”. In: ICLR’19. 2019. [3] Uri Alon et al. “Code2vec: Learning Distributed Representations of Code”. In: *Proc. ACM Program. Lang.*

3.POPL (Jan. 2019). DOI: [10.1145/3290353](https://doi.org/10.1145/3290353). URL: <https://doi.org/10.1145/3290353>.

9https://github.com/konygin/code anomalies

[4] Timofey Bryksin et al. “Using Large-Scale Anomaly Detection on Code to Improve Kotlin Compiler”. In: *Proceedings of the 17th International Conference on Mining Software Repositories*. MSR ’20. Seoul, Republic of Korea: Association for Computing Machinery, 2020, pp. 455–465. ISBN: 9781450375177. DOI: [10.1145/](https://doi.org/10.1145/3379597.3387447)

[3379597.3387447](https://doi.org/10.1145/3379597.3387447). URL: <https://doi.org/10.1145/3379597.3387447>.

[5] Jacob Devlin et al. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”.

In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 4171–4186. DOI: [10.18653/v1/N19-1423](https://doi.org/10.18653/v1/N19-1423). URL:

<https://www.aclweb.org/anthology/N19-1423>.

[6] Zhangyin Feng et al. “CodeBERT: A Pre-Trained Model for Programming and Natural Languages”. In: *Find- ings of the Association for Computational Linguistics: EMNLP 2020*. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547. DOI: [10.18653/v1/2020.findings- emnlp.139](https://doi.org/10.18653/v1/2020.findings-emnlp.139). URL: [https](https://www.aclweb.org/anthology/2020.findings-emnlp.139):

[//www.aclweb.org/anthology/2020.findings-emnlp.139](https://www.aclweb.org/anthology/2020.findings-emnlp.139).

[7] Aditya Kanade et al. “Learning and Evaluating Contextual Embedding of Source Code”. In: *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13–18 July 2020, Virtual Event*. Vol. 119. Proceedings of Machine Learning Research. PMLR, 2020, pp. 5110–5121. URL: [http://proceedings.mlr.](http://proceedings.mlr.press/v119/kanade20a.html)

[press/v119/kanade20a.html](http://proceedings.mlr.press/v119/kanade20a.html).

[8] Diederik P. Kingma Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2015. URL: <http://arxiv.org/abs/1412.6980>.

[9] Diederik P. Kingma and Max Welling. “Auto-Encoding Variational Bayes”. In: *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14–16, 2014, Conference Track Proceedings*. 2014. arXiv: <http://arxiv.org/abs/1312.6114v10> [stat.ML].

[10] Yinhan Liu et al. “RoBERTa: A Robustly Optimized BERT Pretraining Approach”. In: *arXiv e-prints*, arXiv:1907.11692 (July 2019), arXiv:1907.11692. arXiv: [1907.11692 [cs.CL]](http://arxiv.org/abs/1907.11692).

[11] Sara Moshtari et al. “Looking for Software Defects? First Find the Nonconformists”. In: *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 2020, pp. 75–86. DOI: [10.1109/SCAM51674.2020.00014](https://doi.org/10.1109/SCAM51674.2020.00014).

[12] Kamrun Nahar Neela et al. “Modeling Software Defects as Anomalies: A Case Study on Promise Repository”.

In: *JSW* 12.10 (2017), pp. 759–772.

[13] Veselin Raychev, Pavol Bielik, and Martin Vechev. “Probabilistic Model for Code with Decision Trees”. In:

*SIGPLAN Not.* 51.10 (Oct. 2016), pp. 731–747. ISSN: 0362-1340. DOI: [10.1145/3022671.2984041](https://doi.org/10.1145/3022671.2984041). URL:

<https://doi.org/10.1145/3022671.2984041>.

[14] Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. “Stochastic Backpropagation and Approximate Inference in Deep Generative Models”. In: *Proceedings of the 31st International Conference on Machine Learning*. Ed. by Eric P. Xing and Tony Jebara. Vol. 32. Proceedings of Machine Learning Research 2. Beijing,

China: PMLR, July 2014, pp. 1278–1286. URL: <http://proceedings.mlr.press/v32/rezende14.html>.

[15] Mayu Sakurada and Takehisa Yairi. “Anomaly Detection Using Autoencoders with Nonlinear Dimensionality Reduction”. In: *MLSDA’14*. 2014.

[16] Ke Shi et al. “PathPair2Vec: An AST path pair-based code representation method for defect prediction”. In: *Journal of Computer Languages* 59 (2020), p. 100979. ISSN: 2590-1184. DOI: https : / / doi . org / 10 . 1016 / j . cola . 2020 . 100979. URL: [http : / / www . sciencedirect . com / science / article / pii /](http://www.sciencedirect.com/science/article/pii/S2590118420300393) [S2590118420300393](http://www.sciencedirect.com/science/article/pii/S2590118420300393).

[17] Haonan Tong, Bin Liu, and Shihai Wang. “Software defect prediction using stacked denoising autoencoders and two-stage ensemble learning”. In: *Information and Software Technology* 96 (2018), pp. 94–111. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2017.11.008. URL: [https://www.sciencedirect.com/](https://www.sciencedirect.com/science/article/pii/S0950584917300113)

[science/article/pii/S0950584917300113](https://www.sciencedirect.com/science/article/pii/S0950584917300113).

[18] Ashish Vaswani et al. “Attention is All You Need”. In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS’17. Long Beach, California, USA: Curran Associates Inc., 2017, pp. 6000–6010. ISBN: 9781510860964.