# Detecting anomalies in Kotlin code

Timofey Bryksin
Saint Petersburg State University, JetBrains Research, Russia
t.bryksin@spbu.ru

Victor Petukhov
ITMO University, Russia
i@victor.am

Kirill Smirenko
Saint Petersburg State University, Russia
k.smirenko@gmail.com

Nikita Povarov
JetBrains, Russia
nikita.povarov@jetbrains.com

## ABSTRACT

This paper discusses code anomalies — code fragments that are written in some way that is not typical for the programming language community. Such code fragments are useful for language creators as performance tests, or they could provide insights on how to improve the language. With Kotlin as the target language, we discuss how the task of detecting code anomalies for a very large codebase could be solved using well-known anomaly detection techniques. We outline and discuss approaches to obtain code vector representation and to perform anomaly detection on such vectorized data. The paper concludes with our preliminary results.

## 1 INTRODUCTION

There are several ways to evaluate a programming language and its compiler. The most straightforward one is to create multiple compiler performance tests based on code fragments that contain typical usages of language elements. However, as with any other piece of software, there always are people trying to use programming languages in a way that was not intended or predicted by their developers. This results in code fragments that have abnormal complexity are composed of sophisticated code constructs or in any other way differ from "typical code" written in this language.

Such *code anomalies* are usually of great interest to language developers. First of all, examples of formally correct but abnormal code could reveal previously unnoticed compiler defects. They also could be used as performance tests for the compiler. So finally, analysis of such code fragments could give a hint on how to improve the language itself. All of this is beneficial for the programming language ecosystem and its community.

In this research, we focus on finding anomalies in Kotlin code. Kotlin is a relatively young object-oriented programming language with an open-source compiler and active continuously growing community. Kotlin developers[1] pay close attention to how their language evolves and to the quality of its toolchain. Several Kotlin developers that we contacted have shown vivid interest in code anomalies, and it became a significant motivation for our research.

Our work aims to create an automated toolset that will be able to analyze a large Kotlin codebase (e.g., all Kotlin code publicly available on the Internet), extract abnormal code fragments from it and group them according to the anomaly type. A huge bonus will be if such tool could also label each group in a human-comprehensible way, describing the nature of the anomalies.

## 2 RELATED WORK

First of all, let us discuss some research solving similar problems.

In [6] the DIDUCE tool for Java programs is introduced. It is based on dynamic code analysis and employs anomaly detection to find possible bugs. DIDUCE runs the target program and stores values for each expression in it. If some expression gets a value that significantly differs from all previous values of this expression, the expression is reported to the user. It is possible to apply such an approach to our task for a specific project, but it does not seem feasible for a large codebase, because we have no specific knowledge about each project (and some parts of some projects may not even compile correctly).

The work reported in [8] presents GrouMiner that uses anomaly detection techniques to detect anomalous interactions between objects. A directed acyclic graph is built from the input source code representing method calls and dependencies between them. Graph-based anomaly detection methods are used to detect unusual method calls and other atypical areas of the control flow graph.

The somewhat similar idea is presented in [11]: the authors propose mining usage models for all objects from the source code as sequences of their method calls. If an abnormal usage pattern emerges in some code fragment, it is treated as a defect candidate. Graph-based anomaly detection techniques are used to detect these anomalies.

The last two approaches are based on static code analysis, and they can be used to find object interaction anomalies on class or even project level. However, such cases refer rather to implementation details of a given program than to the abnormal use of the language itself. These approaches are helpful if one tries to find logical errors in a program and therefore target programming language users, not its developers.

## 3 DETECTION OF CODE ANOMALIES

First, we should determine what form of the code should be analyzed. There are two options for Kotlin: source code and bytecode

[1] Hereafter the term "Kotlin developers" will denote the group of people that are developing Kotlin itself, and the term "Kotlin users" will denote all users of this language.


*ISSTA Companion/ECOOP Companion'18, July 16–21, 2018, Amsterdam, Netherlands*

ACM ISBN 978-1-4503-5939-9/18/07.
https://doi.org/10.1145/3236454.3236457

produced by the compiler. Both seem to be beneficial: source code analysis gives us anomalies in the language use patterns, and bytecode analysis additionally provides data on compiler performance issues. Moreover, these two groups of anomalies can be compared to find code fragments that are anomalous in the form of bytecode and are not considered anomalous as source code. Such cases can indicate some compiler issues.

Then, structural units that will be analyzed for anomalies should be defined. It can be a whole project, a single file, a class, a function, a line of code or even a single operator. It seems like single operators and lines of code are too small and will not capture structures complex enough to form an anomaly. Functions seem to be a good choice since they are small enough to represent one single operation on a class and large enough to contain multiple lines of code (that could form an anomaly). Classes also seem to be a good choice to search for anomalies in inheritance, functions definitions or control and data flow between multiple functions (e.g., extremely long chains of function calls). Files should be considered for analysis if we are trying to find abnormal relations between classes, and projects seem too specific for a target domain and too large to analyze in a project-agnostic way (i.e., not taking into account what was this project created for).

In data science, the anomaly detection task is a well-known problem. To use the existing knowledge in this field for code anomaly detection, we decided to split the problem in the following way: (1) build a vector representation of the input source code, and (2) perform anomaly detection on vectorized data.

### 3.1 Vector representation

Multiple research papers have been published on how to build vector representations for code fragments. Basically, all Code2Vec approaches fall into two groups: explicit and implicit features. Both groups have their advantages and disadvantages.

Explicit features are mostly software metrics (e.g., abstract syntax tree or cohesion/coupling metrics), simple natural language processing features (e.g., unigrams or bag-of-words) and their derivatives. These features are very descriptive: just looking at their values often leads to a hypothesis why this particular code fragment is considered an anomaly. However, it is challenging to choose which set of metrics to use, since many of them are based on opinions on what "good code" is (and some of them even contradict each other).

An alternative is implicit features extracted via autoencoder neural networks, N-grams, abstract syntax tree encoding, feature hashing, etc. Such features lack expressiveness and are hard to interpret by humans, but they could capture code features that were not obvious beforehand.

### 3.2 Anomaly detection

Since the task requires to analyze a large codebase that we know almost nothing about, we are interested primarily in unsupervised and semi-supervised anomaly detection methods.

An extensive overview of anomaly detection techniques is presented in [2]. Local Outlier Factor [1] and Isolation Forest [7] are popular machine learning methods for outlier detection. Their advantage is that they assign anomaly scores to classified objects rather than performing binary classification, and do not make a priory assumptions about data distribution. Some clustering algorithms, such as DBSCAN [4], ROCK [5] or SNN [3] do not require every object to be a part of some cluster, and, therefore, can be used for anomaly detection.

Autoencoders also can be used for anomaly detection. Autoencoder is a neural network that performs vector compression (encoding) and subsequent recovery (decoding). For anomaly detection, decoding loss can be analyzed, and objects with large loss values are considered anomalies.

Another popular method, One-class SVM [9] is an example of a semi-supervised method. It detects novelty, not outliers, and requires a set of labeled data to train the classifier. For example, it could take anomalies detected by some other method mentioned above as such labeled data. Active learning [10] is another promising approach for semi-supervised anomaly detection that allows using interactive feedback from an expert to refine the result.

Furthermore, it would be useful to classify detected code anomalies to identify common types of anomalies. In case of explicit features, statistical analysis of feature vectors can be used to divide the anomalies based on specific factors. Clustering methods, such as hierarchical clustering, DBSCAN, etc., could be used to divide the anomalies automatically. Further research is required to determine which methods are most suitable for code anomaly classification.

## 4 PRELIMINARY RESULTS

In our preliminary research, a dataset of Kotlin code has been collected: all GitHub repositories created before March 2018 that have Kotlin detected as the main programming language and are not forks of other repositories were downloaded, which resulted in approximately 930 thousand source files.

Kotlin functions were selected for vectorization, two approaches for code vector representation were implemented: a feature vector consisting of 51 explicit code metrics and an implicit N-gram approach. Several anomaly detection techniques were tested on such dataset: Local Outlier Factor, Isolation Forest, and Autoencoder neural network. 146 code anomalies detected jointly by all algorithms were clustered into 23 types via Hierarchical agglomerative clustering and presented to a Kotlin developer, who rated each of them from 1 to 5 according to their value. 46 anomalies were rated 4 and 5, which shows both interest in such work from Kotlin developers and room for further improvements.

## REFERENCES


[1] Breunig, M. M., Kriegel, H.-P., Ng, R. T., and Sander, J. Lof: Identifying density-based local outliers. *SIGMOD Rec. 29*, 2 (May 2000), 93–104.

[2] Chandola, V., Banerjee, A., and Kumar, V. Anomaly detection: A survey. *ACM Comput. Surv. 41*, 3 (July 2009), 15:1–15:58.

[3] Ertoz, L., Steinbach, M., and Kumar, V. A new shared nearest neighbor clustering algorithm and its applications. In *Workshop on clustering high dimensional data and its applications at 2nd SIAM international conference on data mining* (2002), pp. 105–115.

[4] Ester, M., Kriegel, H.-P., Sander, J., Xu, X., et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd* (1996), vol. 96, pp. 226–231.

[5] Guha, S., Rastogi, R., and Shim, K. Rock: A robust clustering algorithm for categorical attributes. In *Data Engineering, 1999. Proceedings., 15th International Conference on* (1999), IEEE, pp. 512–521.

[6] Hangal, S., and Lam, M. S. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering* (New York, NY, USA, 2002), ICSE '02, ACM, pp. 291–301.

[7] Liu, F. T., Ting, K. M., and Zhou, Z.-H. Isolation forest. In *Proceedings of the 2008 Eighth IEEE International Conference on Data Mining* (Washington, DC, USA, 2008), ICDM '08, IEEE Computer Society, pp. 413–422.

[8] Nguyen, T. T., Nguyen, H. A., Pham, N. H., Al-Kofahi, J. M., and Nguyen, T. N. Graph-based mining of multiple object usage patterns. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (New York, NY, USA, 2009), ESEC/FSE '09, ACM, pp. 383–392.

[9] Schölkopf, B., Williamson, R., Smola, A., Shawe-Taylor, J., and Platt, J. Support vector method for novelty detection. In *Proceedings of the 12th International Conference on Neural Information Processing Systems* (Cambridge, MA, USA, 1999), NIPS'99, MIT Press, pp. 582–588.

[10] Settles, B. *Active Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.

[11] Wasylkowski, A., Zeller, A., and Lindig, C. Detecting object usage anomalies. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (New York, NY, USA, 2007), ESEC-FSE '07, ACM, pp. 35–44.