Transfer Defect Learning in Software Ecosystems Using Dependency Analysis

Double Blind Review

ABSTRACT

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; *Maintaining software*; • **Information systems** → Expert systems;

KEYWORDS

Defect Predictions, Fault Fixing, Software Maintenance, Software Evolution

ACM Reference Format:
Double Blind Review. 018. Transfer Defect Learning in Software Ecosystems Using Dependency Analysis. In *Proceedings of ACM MSR Conference (MSR'18).* ACM, New York, NY, USA, Article 4, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Defect prediction is a popular field of study in academia. Defect prediction is the science of predicting if a software change, known as diff, introduced a new defect into the system. In order to build defect predictor, researchers and practitioners first extract all the historical data from a given source code repository and label changes as *buggy* or not using the informations from the project tracking system (e.g. Jira, Bugzilla). Then, a model can be built, using a wide range of techniques, in order to predict if an incoming change contains a bug. An incoming change, classified as *buggy* is known as a *risky* commit because the classification is only a prediction. The prediction can turns out to be true or not in the future.

Performant models can be build to achieve defect prediction in efficient manner using numerous tools and approaches (e.g. [16]). However, this field of study suffers from what is known as the cold start problem. The cold start problem is common to all approaches aiming to perform prediction, regardless of the data at hand. In essence, the cold start problem characterize the fact that, in order to build a model, one needs access to a large amount of historical data. Until sufficient data is acquired, no efficient model can be built and prediction acquired from partial models yield lower level of confidence. One of the solution explored in the past for tackling the cold start problem is known as transfer defect learning. Transfer defect learning aims at identifying similarity between projects and in order to adapt a model built for a project to another one.

Past approaches mainly manipulate the feature space in order to transform the model and transfer what has been learned in one project to another. While these approaches are effective they require a significant amount of machine learning knowledge and computational power in order to put in industrial environment.

In this paper, we propose to identify closely related projects using their dependencies in order to pick the right model while waiting for data to be produced and collected within the new project. In the context of software ecosystems, where an organization, produce and maintain a myriad of software systems identifying related project can be challenging. We propose to do so by automatically retrieving dependencies of projects and create clusters of similar projects. The rational behind this approach is that projects sharing the same set of dependencies are potentially trying to achieve similar behavior.

Our approach was developed in collaboration with software developers from Ubisoft La Forge. Ubisoft is one of the world's largest video game development companies specializing in the design and implementation of high-budget video games. Ubisoft software systems are highly coupled containing millions of files and commits, developed and maintained by more than 8,000 developers scattered across 29 locations in six continents.

We tested our approach on 12 major Ubisoft systems. The results show that we are able to identify closely related projects and their related models in order to alleviate the cold start problem in an industrial environment. Moreover, we replicated our study in an open-source environment.

The remaining parts of this paper are organised as follows. In Section 2, we present related work. Sections **??**, 4 and 5 are dedicated to describing our approach, the case study setup, and the case study results. Then, Sections 6 and 7 present the threats to validity and a conclusion accompanied with future work.

2 RELATED WORK

In this section, we present the work that are related the most related to ours. We first present work done in file, module and risky changes in general. Then, we present work that belong to transfer defect learning specifically.

2.1 File, Module and Risky Change Prediction

Existing studies for predicting risky changes within a repository rely mainly on code and process metrics. As discussed in the introduction section, Rosen et al. [16] developed Commit-guru a tool that relies on building models from historical commits using code and process metrics (e.g., code complexity, the experience of the developers, etc.) as the main features. There exist other studies that leverage several code metric suites such as the CK metrics suite [2]

MSR'18, May 2018, Gothemburg, Sweden

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

or the Briand's coupling metrics [1]. These metrics have been used, with success, to predict defects as shown by Subramanyam *et al.* [18] and Gyimothy *et al.* [4].

Further improvements to these metrics have been proposed by Nagappan *et al.* [9, 11] and Zimmerman *et al.* [20, 21] who used call graphs as the main artifact for computing code metrics with a static analyzer.

Nagappan *et al.* et proposed a technique that uses data mined from source code repository such as churns to assess the quality of a change [10]. Hassan *et al* and Ostrand *et al* used past changes and defects to predict buggy locations [5], [14]. Their methods rely on various heuristics to identify the locations that are most likely to introduce a defect. Kim *et al* [8] proposed the bug cache approach, which is an improved technique over Hassan and Holt's approach [5]. Rahman and Devanbu found that, in general, process-based metrics perform as good as code-based metrics [15].

Other studies that aim to predict risky changes use the entropy of a given change [6, 19] and the size of the change combined with files being changed [7].

2.2 Transfer Defect Learning

@Fabio, would you mind drafting something for this section based on the paper selected I gave you?

As mentioned in the previous section, these approaches are effective in transfer defect learning from on project to another. However, they do not leverage the fact that software systems within an organization are built to collaborate with each others and new software are built on top of other, potentially following the same guidelines and practices. Our approach take another route and tries to identify similar project and apply past models while acquiring enough data to build a dedicated model.

3 TRANSFER DEFECT LEARNING IN SOFTWARE ECOSYSTEMS USING DEPENDENCY ANALYSIS

In the following subsections, we present our approach to transfer defect learning in software ecosystems using dependency analysis. First, we present the pipeline of defect learning at Ubisoft. While this pipeline is tailored for Ubisoft's needs, it is very much based on the state of the art of defect learning. Then, we present how we collect and use dependencies to create clusters of alike projects. Finally, we explain the process by which we can select existent models for alleviate the cold start problem for new projects.

3.1 Classical Defect Learning

[Describe the approach Prediction approach in a few paragraphs and introduce the cold-start problem. Basically https://github.com/PapersOfMathieuNls/misfire/blob/master/combined.md.pdf I am waiting to see if the paper is accepted or not at ICSE to do a citation]

3.2 Clustering Projects

We cluster projects according to their dependencies. The rationale is that projects that share dependencies are most likely to contain defects caused by misuse of these dependencies. In this step, the project dependencies are analysed and saved into a single NoSQL graph database as shown in Figure 1. A node corresponds to a project that is connected to other projects on which it depends. Dependencies can be *external* or *internal* depending on whether the products are created in-house or supplied by a third-party. For confidentiality reasons, we cannot reveal the name of the projects involved in the project dependency graph. We show the 12 projects in yellow color with their dependencies in blue color in Figure 5. In total, we discovered 405 distinct dependencies that are internal and external both. The resulting partitioning is shown in Figure 4.

At Ubisoft, dependencies are managed within the framework of a single repository, which makes their automatic extraction possible. In the context of open-source ecosystem, the dependencies can be automatically retrieved if the projects use a dependency manager such as Maven.

Once the project dependency graph is extracted, we use a clustering algorithm to partition the graph. To this end, we choose the Girvan–Newman algorithm [3, 13], used to detect communities by progressively removing edges from the original network. Instead of trying to construct a measure that identifies the edges that are the most central to communities, the Girvan–Newman algorithm focuses on edges that are most likely "between" communities.

This algorithm is very effective at discovering community structure in both computer-generated and real-world network data [13]. Other clustering algorithms can also be used.

4 CASE STUDY SETUP

In this section, we present the setup of our case study in terms of repository selection, dependency analysis, comparison process and evaluation measures.

4.1 Project Repository Selection

4.1.1 Ubisoft. In collaboration with Ubisoft developers, we selected XX major software projects (i.e., systems) developed at Ubisoft to evaluate the effectiveness of XXX. These systems continue to be actively maintained by thousands of developers. Ubisoft projects are organized by game engines. A game engine can be used in the development of many high-budget games. The projects selected for this case study are related to the same game engine. For confidentiality and security reasons, neither the names nor the characteristics of these projects are provided. We can however disclose that the size of these systems altogether consists of millions of lines of code.

4.1.2 Open Source. To select the projects used to evaluate our approach, we followed three simple criteria. First, the projects need to be in Java and use Maven to manage dependencies. This way, we can automatically extract the dependencies and perform the clustering of projects. The second criterion is to have projects that enjoy a large community support and interest. We selected projects that have at least 2000 followers. Finally, the projects must have a public issue repository to be able to mine their past issues and the fixes. We queried Github with these criteria and retrieved 42 projects (see Table **??** for the list of projects), including those from some of major open-source contributors such as Alibaba, Apache Software Foundation, Eclipse, Facebook, Google and Square.

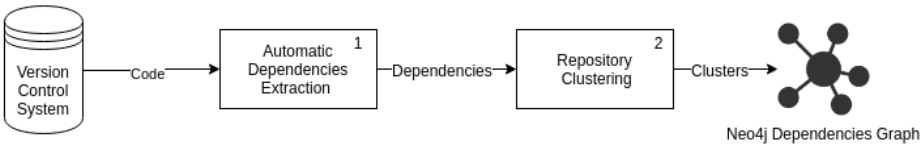

Figure 1: Clustering by dependency

4.2 Project Dependency Analysis

In this section we present how we choose the projects participating in our study.

4.2.1 Ubisoft. Figure 5 shows the projects dependency graph extracted from Ubisoft ecosystem.

As shown in Figure 5, these projects are highly interconnected.

A review of each cluster shows that this partitioning divides projects in terms of their high-level functionalities. For example, one cluster is related to a particular given family of video games, whereas the other cluster refers to another family. We showed this partitioning to 11 experienced software developers and ask them to validate it. They all agreed that the results of this automatic clustering is accurate and reflects well the various project groups of the company. The clusters are used for decreasing the rate of positive.

4.2.2 Open Source. Figure 5 shows the project dependency graph. The dependency graph is composed of 592 nodes divided into five clusters shown in yellow, red, green, purple and blue. The size of the nodes in Figure 5 is proportional to the number of connections from and to the other nodes.

As shown in Figure 5, these Github projects are very much interconnected. On average, the projects composing our dataset have 77 dependencies. Among the 77 dependencies, on average, 62 dependencies are shared with at least one other project from our dataset.

Table 1 shows the result of the Girvan–Newman clustering algorithm in terms of centroids and betweenness. The blue cluster is dominated by Storm from The Apache Software Foundation. Storm is a distributed real-time computation system. Druid by Alibaba, the e-commerce company that provides consumer-to-consumer, business-to-consumer and business-to-business sales services via web portals, dominates the yellow cluster. In recent years, Alibaba has become an active member of the open-source community by making some of its projects publicly available. The red cluster has Hadoop by the Apache Software Foundation as its centroid. Hadoop is an open-source software framework for distributed storage and distributed processing of very large datasets on computer clusters built from commodity hardware. The green cluster is dominated by the Persistence project of OpenHab. OpenHab proposes home automation solutions and the Persistence project is their data access layer. Finally, the purple cluster is dominated by Libdx by Badlogicgames, which is a cross-platform framework for game development.

A review of each cluster shows that this partitioning divides projects in terms of high-level functionalities. For example, the blue cluster is almost entirely composed of projects from the Apache Software Foundation. Projects from the Apache Software Foundation tend to build on top of one another. We also have the red cluster for Hadoop, which is by itself an ecosystem inside the Apache Software Foundation. Finally, we obtained a cluster for e-commerce applications (yellow), real-time network application for home automation (green), and game development (purple).

4.3 Process of Testing And Evaluate Models

In order to test our approach we perform two tests per cluster. In each cluster, we take the latest project added to the cluster and try to predict the firs thousand commits with the model learnt from the before last project (t_1) and one learnt from the combination of all projects in the cluster (t_2).

More formally:

$$M_{i-1} \models P_i \tag{1}$$

and

$$M_{\bigcup_{i=0}^{i-1}} \models P_i \tag{2}$$

where M is a defect model, P is a software project and $\models$ represents the application of a model on a given project.

Then, we report the performances of the tests in terms of precision, recall and F1-measure. The precision is the division of the true positives (i.e. commits correctly classified as buggy) over the true positives plus the false positive (i.e. commits incorrectly classified as buggy).

The recall is the division of the true positive over the false negatives (i.e. the commits incorrectly classified as sane). Finally, the F1-measure is the harmonic means of precision and recall:

$$2.\frac{precision.recall}{precision + recall} \tag{3}$$

We only test the first thousand commit of each project as we consider that after that mark, it will become reasonable to build a dedicated model as historical data would have been collected.

5 CASE STUDY RESULTS

The clusters computed with the dependencies of each project help to solve an important problem in defect prediction, known as cold start [17]. Indeed, as performant as a classifier can be, after training, it still needs to be train with historical data. While the system is learning, no prediction can be done.

There exist approaches have been proposed to solve this problems, however, they all require to classify *similar* projects by hand and then, manipulate the feature space in order to adapt the model learnt from one project to another one [12].

With our approach, the *similarity* between projects is computed automatically and the results show that we do not actually need to manipulate the feature space. Figures **??**, **??** and **??** show the performance of CLEVER classification in terms of ROC-curve for

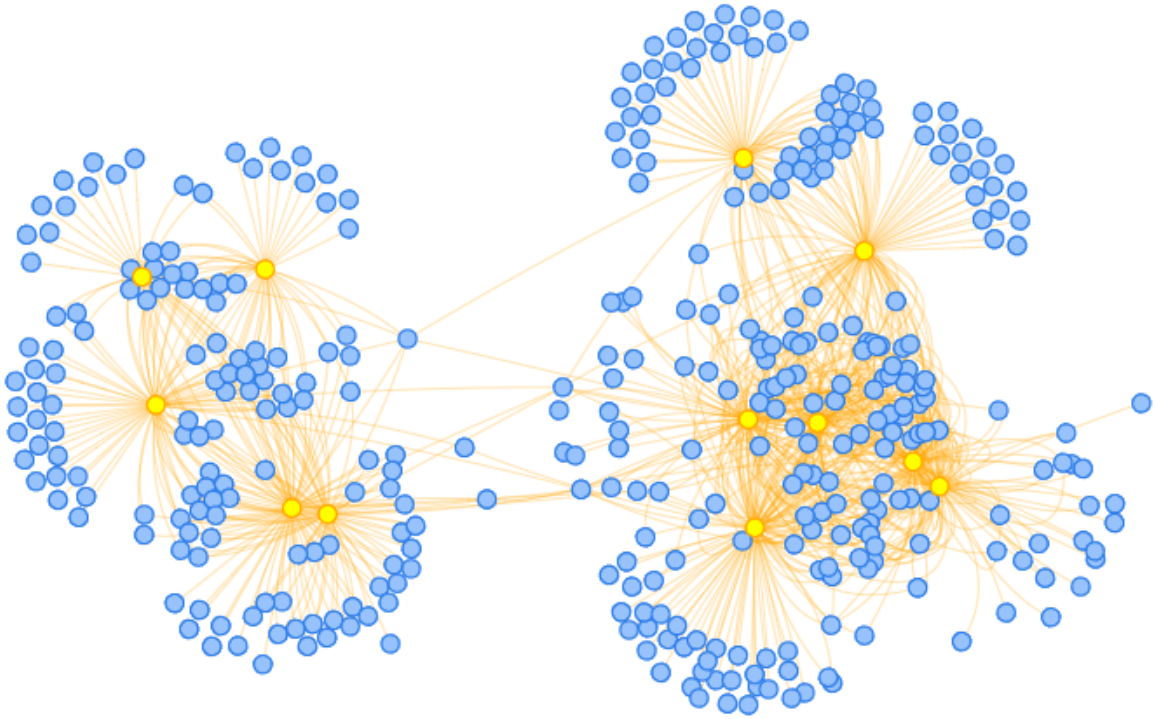

Figure 2: Ubisoft's Dependency Graph

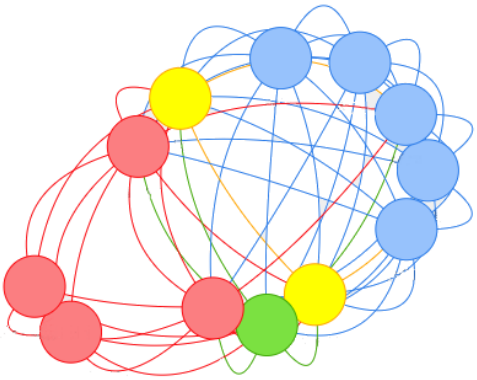

Figure 3: Ubisoft's Clusters

Table 1: Open-source communities in terms of ID, Color code, Centroids, Betweenness and number of members

#ID	Community	Centroids	Betweenness	# Members
1	Blue	Storm	24,525	479
2	Yellow	Alibaba	24,400	42
3	Red	Hadoop	16,709	37
4	Green	Openhab	3,504	22
5	Purple	Libdx	6,839	12

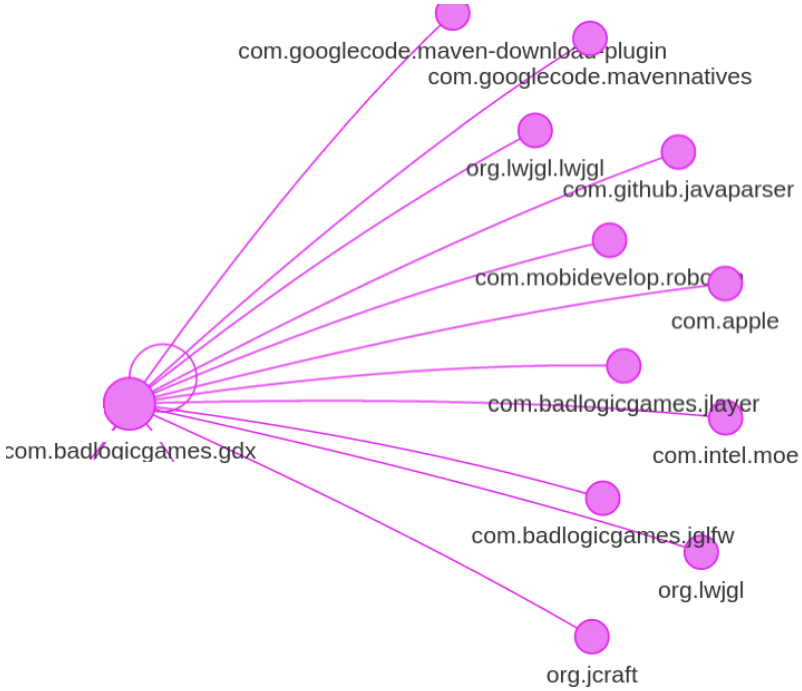

Figure 4: Simplified Open Source Dependency Graph for `com.badlogicgames.gdx`

the first thousand commits of the last project (chronologically) for the blue, yellow and red clusters presented in Figure 4. The left sides or the graph are low cutoff (aggressive) while the right sides are high cutoff (conservative). The area under the ROC curves are 0.817, 0.763 and 0.806 for the blue, yellow and red clusters, respectively. In other words, if we train a classifier with historical data from system in the same cluster as the targeted systems, then we do not have to wait to start classifying incoming commits.

6 DISCUSSION

In this section, we propose a discussion on limitations and threats to validity.

6.1 Limitations

@Wahab, can you have a go at this?

6.2 Threats to Validity

The selection of target systems is one of the common threats to validity for approaches aiming to improve the analysis of software systems. It is possible that the selected programs share common properties that we are not aware of and therefore, invalidate our results. Because of the industrial nature of this study, we had to work with the systems developed by the company.

The programs we used in this study are all based on the C#, C, C++ and Java programming languages. This can limit the generalization of the results to projects written in other languages, especially that the main component of XXX is based on code clone matching.

In conclusion, internal and external validity have both been minimized by choosing a set of 8 different systems (4 open source and 4 closed source), using input data that can be found in any programming languages and version systems (commits and changesets).

7 CONCLUSION

In this paper, we presented an approach to transfer defect learning between projects belonging to the same software ecosystem. To identify candidates that can reuse a defect model, while waiting for enough data to be acquired, using an automated clustering of projects using their dependencies. The main idea behind this choice is that projects in a given ecosystem are made to collaborate with each other and, if projects share a lot of their dependencies it is likely that they solve part of the same problem.

We shown our approach to be working on closed and open sources ecosystem both.

As a feature work, we want to investigate the performances of a model built using the historical data of an entire cluster and see if classical defect learning approaches are effective when applied from cluster to cluster.

8 REPRODUCTION PACKAGE

For security and confidentiality reasons we cannot provide a complete reproduction package that will inevitably involve Ubisoft's copyrighted source code. However, we provide a reproduction package for the open-source parts of our experimentations at TBA.

ACKNOWLEDGMENTS

We are thankful to the software development team at Ubisoft for their participations to the study and their assessment of the effectiveness of our transfer learning technique.

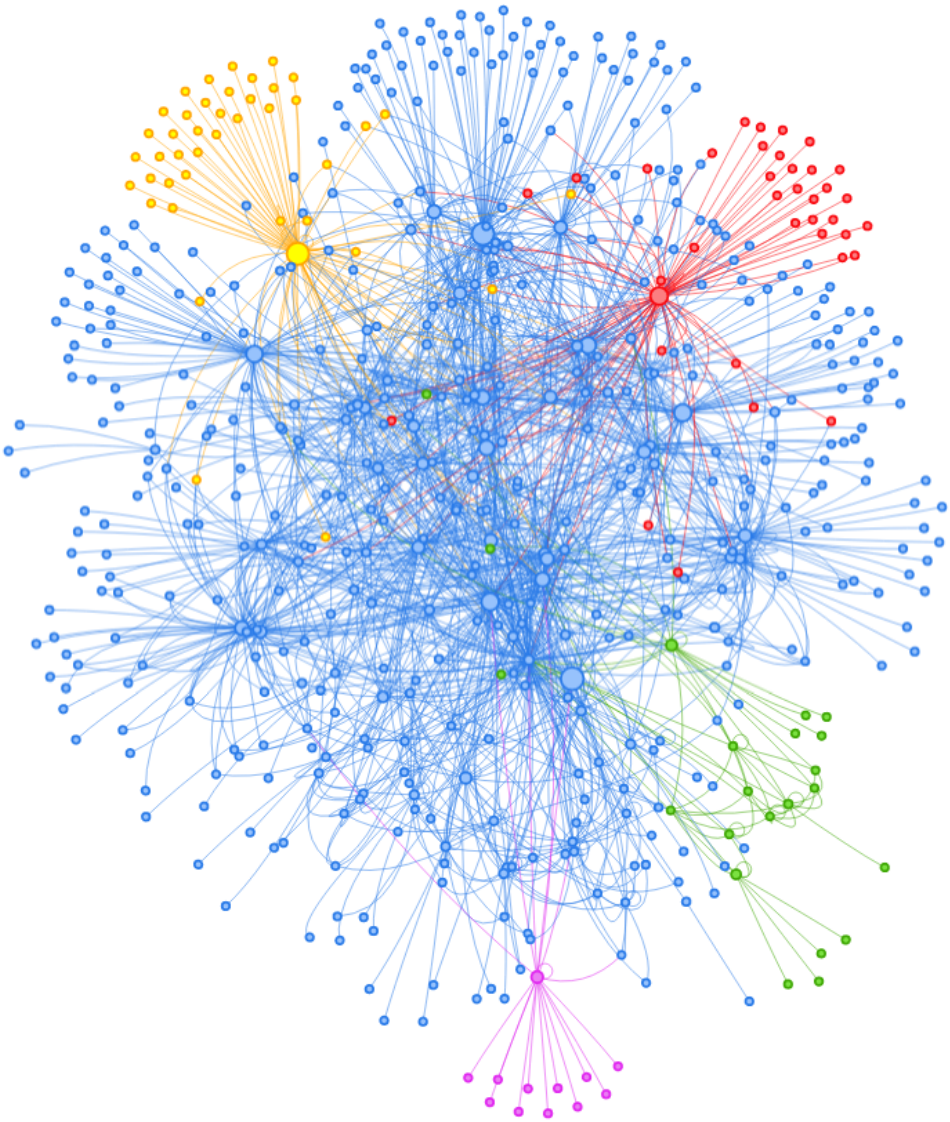

Figure 5: Open Source Dependency Graph

REFERENCES

[1] Briand, L. et al. 1999. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*. 25, 1 (1999), 91–121. DOI:https://doi.org/10.1109/32.748920.
[2] Chidamber, S. and Kemerer, C. 1994. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*. 20, 6 (Jun. 1994), 476–493. DOI:https://doi.org/10.1109/32.295895.
[3] Girvan, M. and Newman, M.E.J. 2002. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*. 99, 12 (Jun. 2002), 7821–7826. DOI:https://doi.org/10.1073/pnas.122653799.
[4] Gyimothy, T. et al. 2005. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*. 31, 10 (Oct. 2005), 897–910. DOI:https://doi.org/10.1109/TSE.2005.112.
[5] Hassan, A. and Holt, R. 2005. The top ten list: dynamic fault prediction. *Proceedings of the international conference on software maintenance* (2005), 263–272.
[6] Hassan, A.E. 2009. Predicting faults using the complexity of code changes. *Proceedings of the international conference on software engineering* (May 2009), 78–88.
[7] Kamei, Y. et al. 2013. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*. 39, 6 (Jun. 2013), 757–773. DOI:https://doi.org/10.1109/TSE.2012.70.
[8] Kim, S. et al. 2007. Predicting Faults from Cached History. *Proceedings of the international conference on software engineering* (May 2007), 489–498.
[9] Nagappan, N. and Ball, T. 2005. Static analysis tools as early indicators of pre-release defect density. *Proceedings of the international*

conference on software engineering (New York, New York, USA, May 2005), 580–586.
[10] Nagappan, N. and Ball, T. 2005. Use of relative code churn measures to predict system defect density. *Proceedings of the international conference on software engineering* (2005), 284–292.
[11] Nagappan, N. et al. 2006. Mining metrics to predict component failures. *Proceeding of the international conference on software engineering* (New York, New York, USA, May 2006), 452–461.
[12] Nam, J. et al. 2013. Transfer defect learning. *Proceedings of the international conference on software engineering* (May 2013), 382–391.
[13] Newman, M.E.J. and Girvan, M. 2004. Finding and evaluating community structure in networks. *Physical Review E.* 69, 2 (Feb. 2004), 026113. DOI:https://doi.org/10.1103/PhysRevE.69.026113.
[14] Ostrand, T. et al. 2005. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*. 31, 4 (Apr. 2005), 340–355. DOI:https://doi.org/10.1109/TSE.2005.49.
[15] Rahman, F. and Devanbu, P. 2013. How, and why, process metrics are better. *Proceedings of the international conference on software engineering* (2013), 432–441.
[16] Rosen, C. et al. 2015. Commit guru: analytics and risk prediction of software commits. *Proceedings of the joint meeting on foundations of software engineering* (New York, New York, USA, Aug. 2015), 966–969.
[17] Schein, A.I. et al. 2002. Methods and metrics for cold-start recommendations. *Proceedings of the annual international onference on research and development in information retrieval* (New York, New York, USA, 2002), 253.
[18] Subramanyam, R. and Krishnan, M. 2003. Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects. *IEEE Transactions on Software Engineering*. 29, 4 (Apr. 2003), 297–310. DOI:https://doi.org/10.1109/TSE.2003.1191795.
[19] Sunghun Kim, S. et al. 2008. Classifying Software Changes: Clean or Buggy? *IEEE Transactions on Software Engineering*. 34, 2 (Mar. 2008), 181–196. DOI:https://doi.org/10.1109/TSE.2007.70773.
[20] Zimmermann, T. and Nagappan, N. 2008. Predicting defects using network analysis on dependency graphs. *Proceedings of the international conference on software engineering* (New York, New York, USA, May 2008), 531.
[21] Zimmermann, T. et al. 2007. Predicting Defects for Eclipse. *Proceedings of the international workshop on predictor models in software engineering* (May 2007), 9.