Towards Compiler Parameter Recommendation Using Code Embeddings

Damian Garber[1,*], Sebastian Lubos[1] and Alexander Felfernig[1]

[1]Graz University of Technology, Inffeldgasse 16b, Graz, 8010, Austria

Abstract

We present a lightweight compiler autotuning approach that combines concepts from configuration space learning with recommender techniques. Our approach uses code embeddings generated by different large language models for data representation and calculation of similarity scores. The best-performing code embedding approach shows, on average, 4.11% faster binaries than the best-performing code metric-based alternative.

Keywords

Compiler Autotuning, Code Embeddings, Collaborative Filtering, Code Metrics

1. Introduction

Compilers are powerful and highly configurable tools. The C compiler GCC[1] has about 200 optimization options that can be activated or deactivated independently. Each option may positively or negatively impact different properties, such as the generated binary's runtime, size, or energy consumption. If these options are correctly utilized, the generated program binaries can be faster, smaller, or more energy-efficient without further investing resources into code refinement. However, choosing the correct options requires expertise in compiler optimization and the program to be optimized. Compiler autotuning addresses this issue by recommending optimization options for a program without any expert involvement. Most approaches for compiler autotuning are computationally expensive and take days to continuously refine the recommended options [1, 2, 3, 4, 5, 6]. Alternative lightweight approaches for compiler autotuning proposed by Burgstaller et al. [7] and Garber et al. [8], can reduce the time needed for recommendation to milliseconds allowing an interactive user experience. This lightweight approach is called Optimisation Space Learning (OSL) [7] and relies on training data collected in advance that is then used for recommendation utilizing nearest-neighbor-based collaborative filtering [9] based on extracted code metrics. The major contributions of this paper are as follows: (1) We extend OSL by incorporating and comparing different code embeddings. (2) We show that the new embeddings significantly outperform the standard compiler optimization options in terms of the runtime performance of the generated program.

The remainder of this paper is structured as follows. Related work is presented in Section 2. In Section 3, our recommendation approach is discussed in detail. We discuss our experimental setup and the evaluation in Section 4, while discussing possible future extensions in Section 5. The paper is concluded with Section 6.

2. Related Work

Compiler autotuning is the automated selection of advantageous compiler optimization options for a program. It can be divided into the *phase selection* problem and the *phase ordering* problem [10]. Phase ordering tries to find an optimal sequence to apply the options, while phase selection, the focus of this

ConfWS'25: 27th International Workshop on Configuration, Oct 25–26, 2025, Bologna, Italy

*Corresponding author.

damian.garber@tugraz.at (D. Garber); sebastian.lubos@tugraz.at (S. Lubos); alexander.felfernig@tugraz.at (A. Felfernig)
0009-0005-0993-0911 (D. Garber); 0000-0002-5024-3786 (S. Lubos); 0000-0003-0108-3146 (A. Felfernig)

[1]https://gcc.gnu.org/

work, tries to identify which optimizations should be applied. The optimality of options can be defined with different properties, the most common of which is runtime. However, space, energy, or similar measurable properties could also be employed.

The state-of-the-art in compiler autotuning is primarily dominated by iterative approaches [1, 2, 3, 4, 5, 6]. Bodin et al. [11] propose one of the first compiler autotuning approaches. They generate an initial set of optimization options to be activated, compile the program using these options, measure its performance, and refine the configuration in a loop until achieving satisfactory results. Most newer approaches build on this concept, like COBAYN [12], which uses Bayesian Networks to narrow the search space. The current state-of-the-art method, BOCA [13], employs Bayesian Optimization to identify key optimizations and streamline the search process. CompTuner [14] builds a prediction model for the runtime of different optimization options and uses a particle swarm optimization algorithm [15] to improve the search performance. Cole [4] can perform multi-target optimization by iteratively creating a Pareto front.

Performance is a key challenge for the computationally intensive, iterative, state-of-the-art approaches, as they require numerous compilations. As project sizes grow, this becomes a significant issue. For instance, Cole must construct a Pareto front, which takes 50 days on a single machine [4]. To address these limitations, newer lightweight approaches like Optimization Space Learning (OSL) [7] adopt alternative strategies to provide faster optimization recommendations, trading off a small degree of recommendation quality for improved responsiveness.

OSL achieves this by combining configuration space learning [16, 17] techniques like the t-wise feature coverage heuristic [16, 17, 18] with collaborative filtering [9]. In this context, collaborative filtering relies on code metrics (e.g., McCabe, Halstead, or counts of keywords) extracted from the optimized programs. This paper presents an alternative collaborative filtering approach based on code embeddings [19].

3. Recommendation Approach

Optimization Space Learning (OSL) is a compiler autotuning approach introduced initially by Burgstaller et al. [7]. The approach combines concepts from configuration space learning [16, 17] for data generation and collaborative filtering [9] for configuration recommendation. The key contribution of OSL is its recommendation speed, which is achieved after a one-time collection of training data within tens of milliseconds. Meanwhile, the iterative state-of-the-art compiler autotuning approaches [1, 2, 3, 4, 5, 6], report computation times of several days. These differences are due to the iterative approaches requiring a continuous refinement process of recommendation result testing, adaptation, and restarting.

3.1. Data Collection

OSL needs to collect initial training data to provide recommendations for a new hardware environment. Two decisions have to be made to generate the training data. The first is which programs to use for training. The second is the heuristic used for generating the sample configurations.

The training approach is based on configuration space learning [16, 20, 17], motivated by the infeasibility of exhaustively exploring configuration spaces due to their exponential size [21]. For example, GCC includes around 200 options, yielding a configuration space of roughly 2^{200} configurations. Even assuming $1ms$ per compilation and measurement, full exploration would take $5 * 10^{49}$ years. Therefore, following Pereira et al. [16], collecting a small, representative configuration subset is necessary.

In order to collect such a small representative set of configurations, we use sampling approaches discussed by Pereira et al. [16] and Garber et al. [17]. Burgstaller et al. [7] considered initially two sampling approaches: Uniform Random Sampling (URS) and t-wise Feature Coverage Heuristics (FCH). URS is well-established [16, 17, 22, 23, 24], but has drawbacks with scalability. The main drawback of FCH, on the other hand, is its expensive computation, which is mitigated by the unconstrained nature of the problem (options are independent of each other) and the fact that this needs to be performed only once. Therefore, OSL ultimately relies on the t-wise FCH [16, 17] to generate the samples for the

Table 1
Example user-based collaborative filtering recommendation setting in compiler autotunning. The values show programs' runtime [s] when compiled with the referenced compiler configuration c_i.

Programs	c_1	c_2	c_3	c_4	c_5	c_6
p_1	1.21	1.70	1.27	1.19	1.76	1.32
p_2	25.01	18.69	16.32	17.06	16.45	16.47
p_3	0.50	0.56	0.48	0.54	0.74	0.73
p_4	?	?	?	?	?	?

training data. The samples generated by t-wise FCH are guaranteed to contain all possible tuples of size t that can be present in the system at least once in the generated configurations. Burgstaller et al. [7] report FCH with $t = 3$ to perform best for this task, which is confirmed by Garber et al. [8] and our work presented in this paper. Next, we need a set of programs to synthesize the needed data. We use the same benchmark used in the original work by Burgstaller et al. [7] and in the improved OSL by Garber et al. [8]. The PolyBench benchmarks [25] provide 30 programs written in C and are widely used in related literature [7, 13, 14, 8].

We construct the training data by compiling an executable for each configuration provided by the sampling approach and each program in the benchmark. The performance properties of these executables are then measured using *perf-stat*.[2]

OSL extracts at this point a vector of 111 source code metrics, such as McCabe's Cyclomatic Complexity [26], Halstead Complexity [27], or simple counts like the number of times a particular keyword occurs, using the CQMetrics tool by [28]. OSL uses the first 66 of those source code metrics to calculate program similarities during the recommendation process since the latter metrics primarily are related to coding style, i.e., indentation space counts. A complete list of the metrics extracted is provided in the CQMetrics documentation.[3] We compare the performance of this code metric-based similarity with our approach of using code embeddings-based similarity. A description of the code embeddings used is provided in Section 4.

3.2. Recommendation

Essentially, we apply nearest neighbor-based collaborative filtering [9] on synthesized data [29, 22], which has been obtained using heuristics known from configuration space learning [16, 17].

Our variant of user-based collaborative filtering differs slightly from the standard setting (see Table 1). Here, programs act as users, configurations as items, and runtime serves as the rating (a lower runtime is analogous to a higher rating). Unlike typical scenarios, we have complete performance data for all program-configuration pairs generated by the data collection process, except for the target program. Thus, we require an external metric to estimate program similarity. The version of OSL used by Burgstaller et al. [7] and Garber et al. [8] computes similarity using source code metrics and the Euclidean distance [30, 7] (see Formula 1 and Formula 2). In this context, x and y are n-dimensional vectors with components x_1 to x_n and y_1 to y_n, representing programs x and y. In OSL, these vectors consist of $n = 66$ code metrics, while in our approach, they are the extracted fixed-size (n) embedding vectors.

$$dis(x, y) = \sum_{i=1}^{n} |y_i - x_i|^2 \tag{1}$$

$$sim(x, y) = \frac{1}{1 + dis(x, y)} \tag{2}$$

Table 2 shows a simplified example of the code metric vectors used, and Table 3 shows an example of how the distances and similarities, as defined in Formula 1 and Formula 2 respectively, would look like

[2]https://perf.wiki.kernel.org/index.php/Main_Page
[3]https://github.com/dspinellis/cqmetrics/blob/master/metrics.md

Table 2
Simplified example of code metric vector, limited to McCabe [26], Halstead [27], and occurrences of *const*.

Program	Halstead	McCabe	Keywordcount: *const*
p_1	110	10	4
p_2	111	7	6
p_3	108	9	4
p_4	104	13	2

Table 3
Example distance and similarity calculations based on metrics shown in Table 2.

x	y	$dis(x,y)$	$sim(x,y)$	$sim(x,y)$ [%]
p_1	p_2	14	0.067	6.7 %
p_1	p_3	5	0.167	16.7 %
p_1	p_4	49	0.020	2.0 %

Table 4
Code embeddings tested for recommendation.

Name	Description
BGE	A BAAI general embedding model that transforms any given English text into a compact vector
RoBERTa	A DistilRoBERTa-base model trained for code search

Table 5
Example aggregation of three well-performing compiler parameter configurations using a parameter-wise majority voting for final recommendation.

	opt_1	opt_2	opt_3	opt_4	opt_5
$conf_1$	1	1	0	1	0
$conf_2$	0	1	0	0	1
$conf_3$	1	0	1	0	0
recommendation	1	1	0	0	0

for these values. In the example, the highest similarity is 16.7 % between p_1 and p_3. Thus, p_3 is the most similar program to p_1. We propose the use of code embeddings extracted from the programs instead. To this end, we test the performance of two embeddings, shown in Table 4. After testing several common ways of calculating the similarities of two embedding vectors, such as the cosine similarity, we use the same Euclidian distance-based approach described earlier.

The remaining process is identical to the typical user-based collaborative filtering procedure. The best-rated (fastest runtime) configuration of the most similar program recommends a configuration for p_4.

The final recommendation step aggregates the results. Since the FCH-collected configurations cover only a small subset of all compiler settings, we generate multiple recommendations from the nearest neighbors and combine them via majority vote (Table 5). Following Burgstaller et al. [7], we set the number of top configurations and nearest neighbors to 5, a choice we confirmed and applied to all experiments. Thus, the final recommendation aggregates the 5 best configurations from the 5 nearest neighbors.

Table 6
The runtime performance (RT) of recommended optimization options and their speedup (SU) compared to O3. The runtime is given in seconds, and the best performers for each program are bold.

Program	O3	OSL		OSL N&E		BGE		RoBERTa	
	RT	RT	SU	RT	SU	RT	SU	RT	SU
correlation	1.841	1.781	**1.034**	1.868	0.985	1.804	1.021	1.840	1.001
covariance	1.817	2.471	0.735	2.502	0.726	1.903	0.955	1.857	**0.979**
2mm	2.163	2.181	0.992	3.149	0.687	2.187	0.989	2.099	**1.031**
3mm	3.795	3.684	1.030	3.885	0.977	3.664	**1.036**	3.783	1.003
atax	0.016	0.014	1.137	0.013	**1.177**	0.015	1.074	0.014	1.161
bicg	0.017	0.017	**0.968**	0.020	0.847	0.019	0.894	0.018	0.929
doitgen	0.521	0.503	1.034	0.505	1.031	0.498	**1.045**	0.499	1.044
mvt	0.018	0.018	**1.027**	0.020	0.928	0.019	0.972	0.019	0.965
gemm	1.153	1.149	1.003	0.644	1.790	1.158	0.995	0.554	**2.080**
gemver	0.026	0.024	1.087	0.026	0.988	0.024	1.063	0.023	**1.129**
gesummv	0.012	0.012	**0.985**	0.013	0.899	0.014	0.837	0.017	0.722
symm	1.625	1.619	1.003	1.615	1.006	1.614	**1.007**	1.646	0.987
syr2k	1.789	1.776	1.007	1.742	1.027	1.864	0.960	1.739	**1.029**
syrk	0.613	0.531	1.154	0.453	1.354	0.489	1.253	0.418	**1.466**
trmm	1.336	0.721	1.853	0.724	1.844	0.727	1.839	0.709	**1.884**
cholesky	12.856	12.015	1.070	12.727	1.010	12.495	1.029	10.741	**1.197**
durbin	0.004	0.003	1.288	0.003	1.260	0.002	1.898	0.002	**1.952**
gramschmidt	1.950	1.958	0.996	2.022	0.965	1.893	**1.030**	1.923	1.014
lu	16.486	13.676	**1.205**	14.898	1.107	14.915	1.105	14.963	1.102
ludcmp	13.993	12.799	**1.093**	16.469	0.850	14.546	0.962	14.384	0.973
trisolv	0.007	0.009	0.784	0.006	1.134	0.006	**1.204**	0.007	1.120
deriche	0.149	0.140	1.066	0.146	1.023	0.139	**1.073**	0.143	1.042
floyd-warshall	17.773	17.656	1.007	17.790	0.999	13.671	**1.300**	14.022	1.267
nussinov	3.418	2.751	1.242	3.129	1.093	2.665	**1.283**	2.721	1.256
adi	9.659	9.347	**1.033**	9.876	0.978	9.709	0.995	9.783	0.987
fdtd-2d	1.908	1.702	1.121	1.671	1.142	1.697	1.125	1.667	**1.145**
heat-3d	3.479	2.172	1.602	1.949	**1.784**	2.096	1.660	2.064	1.685
jacobi-1d	0.002	0.001	1.424	0.001	**1.515**	0.002	1.169	0.002	1.138
jacobi-2d	2.143	1.445	**1.483**	1.487	1.441	1.479	1.449	1.487	1.441
seidel-2d	20.167	15.467	1.304	11.566	**1.744**	20.075	1.005	20.084	1.004

Table 7
We present different aggregations (mean, median, TOP 1, and TOP 2) of the results shown in Table 6.

Function	OSL	OSL N&E	BGE	RoBERTa
MEAN	1.126	1.144	1.141	**1.191**
MEDIAN	1.050	1.025	1.040	**1.073**
TOP 1	8/30	4/30	8/30	**10/30**
TOP 2	16/30	11/30	14/30	**19/30**

4. Evaluation

In this section, we evaluate the use of code embeddings to recommend compiler optimization options and whether they outperform code metric-based approaches like OSL [7] or its enhanced version of OSL Normalized and Equalized (OSL N&E) [8]. Code embeddings represent code as fixed-sized numerical vectors containing semantic and structural information [19]. They are usually employed by machine learning or large language models when working with code. We test two embeddings BGE [4], a general

[4]https://huggingface.co/BAAI/bge-base-en-v1.5

text embedding, and RoBERTa [5], a specialized code embedding (described in Table 4).

4.1. Experimental Setup

Our evaluation uses GCC version 14.2.1 on a Lenovo ThinkPad P53s machine with an Intel i7-8665U processor and 32GB memory running Linux 6.1.119-1-MANJARO. We use the PolyBench/C Benchmark [25] for training and testing, which contains 30 programs written in C and is commonly used in compiler autotuning evaluation settings [7, 13, 14, 8]. Due to the relatively small sample size, we apply leave-one-out cross-validation [31]. Thus, each benchmark program in PolyBench was tested using a model trained with the remaining 29. In order to visualize the performance more efficiently, we define a speedup factor compared to GCC's set of default optimizations O3 in Equation 3.

$$speedup = \frac{t_{O3}}{t_{REC}} \tag{3}$$

t_{O3} and t_{REC} represent the program's runtime compiled using O3 and the recommended parameter settings respectively. A speedup of 1.1 indicates a 1.1 times faster runtime.

4.2. Results

Table 6 presents the performance of the tested approaches, with aggregated results in Table 7. Both code embeddings outperformed the baseline OSL method, which achieved an average speedup of 1.126. BGE reached 1.141, slightly below the enhanced OSL N&E at 1.144. RoBERTa achieved the highest average speedup of 1.191. Regarding frequency as a top performer, RoBERTa leads (Top 1 in 10/30 cases, Top 2 in 19/30), followed by BGE narrowly outperforming OSL, while OSL N&E comes last. These results indicate that embeddings are effective for recommending compiler optimizations, especially when using models like RoBERTa, which are specifically trained on code.

5. Future Work

The first results of using code embeddings in the context of lightweight compiler autotuning show promise. However, in future work, we would like to expand the number of evaluated code embeddings, especially further towards models specialized in coding or code manipulations, such as CodeBERT or GraphCodeBERT, potentially improving our results further.

6. Conclusion

In summary, we evaluated using code embeddings to recommend compiler optimizations. Our results show that embeddings perform comparably to code metric-based approaches and surpass them in the case of embeddings from models trained on code. The best-performing method leverages embeddings from a RoBERTa model trained for code search, achieving an average runtime speedup factor of 1.191, 4.11% faster than the enhanced code metrics baseline. Major tasks of future work include the extension of the dataset as well as the testing of additional embeddings.

Acknowledgments

This study was funded by GENRE, Austrian Research Promotion Agency (Grant No. 915086).

[5]https://huggingface.co/flax-sentence-embeddings/st-codesearch-distilroberta-base

Declaration on Generative AI

While preparing this work, the author(s) used ChatGPT-4 (GPT-4-turbo) and Grammarly to check grammar and spelling and improve formulations. After using these tool(s)/service(s), the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

References

[1] Y. Chen, S. Fang, Y. Huang, L. Eeckhout, G. Fursin, O. Temam, C. Wu, Deconstructing iterative optimization, ACM Transactions on Architecture and Code Optimization (TACO) 9 (2012) 1–30.

[2] U. Garciarena, R. Santana, Evolutionary optimization of compiler flag selection by learning and exploiting flags interactions, in: Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion, 2016, pp. 1159–1166.

[3] S. V. Gheorghita, H. Corporaal, T. Basten, Iterative compilation for energy reduction, Journal of Embedded Computing 1 (2005) 509–520.

[4] K. Hoste, L. Eeckhout, Cole: compiler optimization level exploration, in: Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization, 2008, pp. 165–174.

[5] L. Pérez Cáceres, F. Pagnozzi, A. Franzin, T. Stützle, Automatic configuration of gcc using irace, in: Artificial Evolution: 13th International Conference, Évolution Artificielle, EA 2017, Paris, France, October 25–27, 2017, Revised Selected Papers 13, Springer, 2018, pp. 202–216.

[6] S. Triantafyllis, M. Vachharajani, N. Vachharajani, D. I. August, Compiler optimization-space exploration, in: International Symposium on Code Generation and Optimization, 2003. CGO 2003., IEEE, 2003, pp. 204–215.

[7] T. Burgstaller, D. Garber, V.-M. Le, A. Felfernig, Optimization space learning: A lightweight, noniterative technique for compiler autotuning, in: Proceedings of the 28th ACM International Systems and Software Product Line Conference, SPLC '24, Association for Computing Machinery, New York, NY, USA, 2024, p. 36–46. URL: https://doi.org/10.1145/3646548.3672588. doi:10.1145/3646548.3672588.

[8] D. Garber, S. Lubos, V.-M. Le, A. Felfernig, Enhanced optimization space learning: Towards real-time compiler optimization, in: 38th International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems, 2025. Accepted.

[9] M. D. Ekstrand, J. T. Riedl, J. A. Konstan, et al., Collaborative filtering recommender systems, Foundations and Trends® in Human–Computer Interaction 4 (2011) 81–173.

[10] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, C. Silvano, A survey on compiler autotuning using machine learning, ACM Computing Surveys (CSUR) 51 (2018) 1–42.

[11] F. Bodin, T. Kisuki, P. Knijnenburg, M. O'Boyle, E. Rohou, Iterative compilation in a non-linear optimisation space, in: Workshop on profile and feedback-directed compilation, 1998.

[12] A. H. Ashouri, G. Mariani, G. Palermo, E. Park, J. Cavazos, C. Silvano, Cobayn: Compiler autotuning framework using bayesian networks, ACM Transactions on Architecture and Code Optimization (TACO) 13 (2016) 1–25.

[13] J. Chen, N. Xu, P. Chen, H. Zhang, Efficient compiler autotuning via bayesian optimization, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), IEEE, 2021, pp. 1198–1209.

[14] M. Zhu, D. Hao, J. Chen, Compiler autotuning through multiple phase learning, ACM Trans. Softw. Eng. Methodol. (2024). URL: https://doi.org/10.1145/3640330. doi:10.1145/3640330, just Accepted.

[15] J. Kennedy, R. Eberhart, Particle swarm optimization, in: Proceedings of ICNN'95-international conference on neural networks, volume 4, IEEE, 1995, pp. 1942–1948.

[16] J. Alves Pereira, M. Acher, H. Martin, J.-M. Jézéquel, G. Botterweck, A. Ventresque, Learning

software configuration spaces: A systematic literature review, Journal of Systems and Software 182 (2021) 111044.

[17] D. Garber, T. Burgstaller, A. Felfernig, V.-M. Le, S. Lubos, T. Tran, S. Polat-Erdeniz, Collaborative recommendation of search heuristics for constraint solvers, in: ConfWS'23: 25th International Workshop on Configuration, Sep 6–7, 2023, Málaga, Spain, 2023.

[18] J. Oh, P. Gazzillo, D. Batory, T-wise coverage by uniform sampling, in: Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A, SPLC '19, Association for Computing Machinery, New York, NY, USA, 2019, p. 84–87. URL: https://doi.org/10.1145/3336294.3342359. doi:10.1145/3336294.3342359.

[19] Z. Chen, M. Monperrus, A literature study of embeddings on source code, arXiv preprint arXiv:1904.03061 (2019).

[20] D. Benavides, P. Trinidad, A. Ruiz-Cortés, Automated reasoning on feature models, in: International Conference on Advanced Information Systems Engineering, Springer, 2005, pp. 491–503.

[21] M. Acher, H. Martin, J. A. Pereira, A. Blouin, J.-M. Jézéquel, D. E. Khelladi, L. Lesoil, O. Barais, Learning very large configuration spaces: What matters for Linux kernel sizes, Ph.D. thesis, Inria Rennes-Bretagne Atlantique, 2019.

[22] K. S. Meel, Counting, sampling, and synthesis: The quest for scalability., in: IJCAI, 2022, pp. 5816–5820.

[23] J. Oh, D. Batory, R. Heradio, Finding near-optimal configurations in colossal spaces with statistical guarantees, ACM Transactions on Software Engineering and Methodology 33 (2023) 1–36.

[24] Q. Plazar, M. Acher, G. Perrouin, X. Devroey, M. Cordy, Uniform sampling of sat solutions for configurable systems: Are we there yet?, in: 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST), IEEE, 2019, pp. 240–251.

[25] L.-N. Pouchet, Polybench: The polyhedral benchmark suite, http://www.cs.ucla.edu/~pouchet/software/polybench/, 2012. Accessed: 2024.

[26] T. J. McCabe, A complexity measure, IEEE Transactions on software Engineering (1976) 308–320.

[27] M. H. Halstead, Elements of Software Science (Operating and programming systems series), Elsevier Science Inc., 1977.

[28] D. Spinellis, P. Louridas, M. Kechagia, The evolution of c programming practices: a study of the unix operating system 1973–2015, in: Proceedings of the 38th International Conference on Software Engineering, ICSE '16, Association for Computing Machinery, New York, NY, USA, 2016, p. 748–759. URL: https://doi.org/10.1145/2884781.2884799. doi:10.1145/2884781.2884799.

[29] J. Alves Pereira, M. Acher, H. Martin, J.-M. Jézéquel, Sampling effect on performance prediction of configurable systems: A case study, in: Proceedings of the ACM/SPEC International Conference on Performance Engineering, 2020, pp. 277–288.

[30] G. Jain, T. Mahara, K. N. Tripathi, A survey of similarity measures for collaborative filtering-based recommender system, in: Soft Computing: Theories and Applications: Proceedings of SoCTA 2018, Springer, 2020, pp. 343–352.

[31] T.-T. Wong, Performance evaluation of classification algorithms by k-fold and leave-one-out cross validation, Pattern recognition 48 (2015) 2839–2846.