Measuring the Quality of Machine Learning and Optimization Frameworks

Ignacio Villalobos, Javier Ferrer, and Enrique Alba

Universidad de Málaga, Málaga, Spain
{nacho, ferrer, eat}@lcc.uma.es

Abstract. Software frameworks are daily and extensively used in research, both for fundamental studies and applications. Researchers usually trust in the quality of these frameworks without any evidence that they are correctly build, indeed they could contain some defects that potentially could affect to thousands of already published and future papers. Considering the important role of these frameworks in the current state-of-the-art in research, their quality should be quantified to show the weaknesses and strengths of each software package.

In this paper we study the main static quality properties, defined in the product quality model proposed by the ISO 25010 standard, of ten well-known frameworks. We provide a quality rating for each characteristic depending on the severity of the issues detected in the analysis. In addition, we propose an overall quality rating of 12 levels (ranging from A+ to D-) considering the ratings of all characteristics. As a result, we have data evidence to claim that the analysed frameworks are not in a good shape, because the best overall rating is just a C+ for Mahout framework, i.e., all packages need to go for a revision in the analysed features. Focusing on the characteristics individually, maintainability is by far the one which needs the biggest effort to fix the found defects. On the other hand, performance obtains the best average rating, a result which conforms to our expectations because frameworks' authors used to take care about how fast their software runs.

Keywords: maintainability, reliability, performance, security, quality

1 Introduction

The international research community considers essential the replicability of the experimentation carried out in an article for progress in science. Actually, when the experimental artifacts are available, other researchers can validate the published work. With this requirement in mind, researchers began to make available their algorithms, which finally have become in large software frameworks [1,2]. These frameworks are collections of algorithms designed for solving complex problems that are freely available for the research and industrial communities. The main advantage for the users of these frameworks is that they can run lots of algorithms without the effort of designing and implementing them. The main problem, at the same time, is that they do not know on how algorithms are actually implemented, and even tend to trust in the quality without any data or evidence that they are at least good software pieces.

In the artificial intelligence research field, Machine Learning and Optimization Frameworks (MLOFs) are widely used by the community. The machine learning packages provide statistical techniques to progressively improve on a specific task from a huge dataset and extract knowledge. Besides, optimization frameworks are used for obtaining optimal solutions for complex continuous and discrete optimization problems. Artificial intelligence (AI) research community is particularly prolific in these fields due to the impact of the application of artificial intelligence techniques to the existing problems. However, finding experts in software engineering in AI is not that common, thus having the undesired result that most researchers build software without even applying the basics of software engineering that all graduate programs teach across the world.

Over the last few years, researchers have offered a wide variety of open source MLOFs that have been used in lots of articles [3]. Researchers use the frameworks' algorithms to look for evidence in favour of a hypothesis they try to support, refute, or validate. Actually, lots of works cite MLOFs because they use them in their experimentation. To illustrate this fact, we only have to enumerate the citations of some MLOFs: an article on the update of WEKA [2] has 16,653 citations[1], the framework Keel [1] has 1,102 citations[1], and JMetal [4] has 712 citations[1] only accounting for the seed articles introducing the tool, and we are just mentioning a few of them. Their popularity is then clear, because many researchers just want to focus on the application and not in the algorithm.

The main problem is that, generally speaking, the open source MLOFs are offered without any warranties of any kind concerning the safety, performance, bugs, inaccuracies, typographical errors, or other harmful characteristics of the software. The user let us alone with any problem or bug he/she can experience, if he/she is actually noticing such errors at all: sometimes the result of low quality is an unnoticed lose of time, or even a more complex situation where the user thinks to be running a given algorithm that actually is not performing the numerical steps as expected. In fact, we should take into account that some MLOFs users do not understand the provided implementation due to its complexity (not all users have programming skills) or they do not devote enough time to analyse it. In practice, users blindly trust the provided implementation and use it "as is". We want to highlight that a bad quality framework potentially may affect the already published results and the future ones of thousands of researchers. This is the reason why we analyse the quality of some open MLOFs in this paper.

Due to the lack of studies about the quality level the MLOFs have, our final goal is to offer a quantitative quality study of a subset of well-known MLOFs including four key aspects of the internal quality model of the ISO 25010 standard: maintainability, reliability, security, and performance efficiency. Our goal is to be positive and give constructive hints on how to solve the found problems, as well as to guide on the fastest ways (where/how) to do so. By no means this is a critic, much on the contrary we respect and endorse these frameworks and just want to contribute to their larger future success.

[1] At the moment of writing: 17, May 2018

The main contributions of this work are the following ones: 1) To provide the first quality analysis for ten well-known MLOFs, 2) To evaluate four key attributes about the quality of the frameworks: maintainability, reliability, security, and performance efficiency, 3) To provide an estimation of the needed time to fix all the defects found in regards to each characteristic studied, and 4) To assign an overall mark for their quality so as to track their future improvements.

The rest of the paper is organized as follows. In Section 2, we define software quality and the different models proposed by the ISO 25010 standard. After that, we present the ten machine learning and optimization frameworks studied here, and we briefly describe how the analysis is performed. Additionally, we show the analysis results of the four studied quality aspects in four subsections. In Section 4, we discuss about the results obtained in the previous section and we propose an overall rating for the frameworks. Finally, in the last section we draw some conclusions and comment some interesting ideas for future work.

2 Forget on Opinions, Let's Go for Standards: ISO 25010

Informally speaking an average user wants a "good" software or service. This term could be very ambiguous and it is not quantifiable, specially when we talk about software. With the first of many definitions, from a professional point of view, quality has been described as the aptitude to accomplish to be used by a client (1970) or to conform with all the product requirements (1979). A few years later these definitions, the standard ISO 8402 (1986) introduced the now well-known definition for product and service quality. After that, with the standard ISO 9126 (1991), an standard finally add the term *Software product* to that definition. However, we want a quality development and a quality software product, but we also want to quantify the level of quality achieved.

Nowadays, the reference standard and the most used one to evaluate software quality is the standard ISO 25010. It defines a *quality in use* model and a *product quality* model. In this work we focus on the product quality model for analysing software products, in this case of the MLOFs, because of their key role in thousands of published articles. This model defines a taxonomy for the main characteristics to consider when you measure the software quality of a product. In the standard, the quality aspects are divided up in the following eight quality characteristics: maintainability, security, functionality, performance efficiency, reliability, usability, compatibility and portability. In this first paper we focus on four of them.

When you deal with a specific software project, we should keep in mind which are our priorities among the quality characteristics mentioned before. Depending on our defined target for the quality analysis, stakeholders or expected tasks of the software, we may be more interested in some quality aspects than in others. Moreover, we should update these quality requirements throughout the product life [5]. We would like to highlight that some requirements or constraints in certain characteristics could have a negative impact in others. Indeed, some aspects of the quality are opposed, e.g., if one try to get the maximum performance, the consequence could be a reduction on the security level.

3 A First Analysis of Static Features of MLOFs

Our goal in this paper is extracting insights from the source code of the MLOFs in order to quantify their overall quality. We study four software characteristics proposed in the product quality model to find possible issues. First, we analysed the maintainability due to the relevance of the capacity of the software to be modified (to extend or fix it). Second, we analysed the security aspect, owing to the relevance to ensure data integrity. After that, we analyse the performance efficiency due to the importance of saving computing resources and decreasing execution time. Finally, we analyse the *reliability* on account of the need to know whether all components perform correctly under specified conditions.

In order to perform all the analyses we have used the tool SonarQube in its version 7.0. We have chosen this tool because the wide acceptance from the developers community and the amount of extensions available. Moreover, SonarQube provides an easy way to be integrated with different tools through its API or plugins. For this study, we have created a quality profile with 295 rules to detect issues about maintainability, security, performance and reliability. Each rule has assigned a severity between the values: blocker, critical, major, minor. This rules' severity is assigned according to the defects that it detects.

But before detecting the weak points to improve in the feature, we need to know a bit the tools from a software perspective: before a deep study let us make some basic objective measurements to shape them better. As a case of study, we select ten MLOFs extracted from the literature, whose source code is developed in Java and are freely available. They have been chosen due to their extended use by the research community, their relevance and impact. Some of them are used for teaching, some for research, some for both. Some are widely known, some are humble code to guide researchers with no knowledge in computer science. We present their main characteristics in Table 1. In the first column we have the name of each software tool, after that we show the version analysed in this work. In the third column, we show the number of lines of code. In the next column we have the McAbee complexity per file, and finally, in the last column we have the number of classes. As we can see in the table, the biggest projects in number of lines of code and classes are Keel and Weka. Also they are the frameworks with the highest McAbee complexity per file. On contrary, the tiniest software packages are ssGA and Mahout but in this case, they are not the ones with the lowest Complexity per file, they are JCLEC and Watchmaker.

Table 1. Machine Learning and Optimization Frameworks.

Project	Version	#LOC	Complexity per File	#Classes
ECJ [6]	25	53,771	18.49	620
JCLEC [7]	4.0	16,652	7.72	323
Jenes [8]	765	11,508	18.06	185
jMetal [4]	5.4	43,144	13.14	609
Keel [1]	3.0	585,337	36.89	3,808
Mahout [9]	0.13.1	1,255	11.14	30
moea-frame [10]	2.12	33,888	12.97	506
ssGA [11]	1.1	672	12.77	13
Watchmaker [12]	0.3.0	5,639	4.79	140
WEKA [2]	14812	353,923	34.81	2,383

In the following subsections, we are presenting the results for the analysed quality characteristics. In each subsection we show a table with the rating obtained by each framework. This rating ranges from A to E where A is the best mark, and E is the worst. For each characteristic, it gets considering the following criterion. If the framework does not have any issue, its get the best mark, an A. Then, if it has only at least a minor issue, it will get a B qualification. If it has at least a major issue, it will get a C. If it has at least a critical issue, it will obtain a D and, finally, if it has one or more blocker issues, the framework will obtain an E. We also show the estimated effort needed (in minutes) to solve all the issues detected and the number of issues for each severity. This effort is calculated regarding to the number of implicated lines of code and an estimation of 18 minutes per line change needed.

3.1 Maintainability

In this subsection, we analyse the selected MLOFs from the point of view of the maintainability. A good source code maintainability can be measured (ISO 25010) as the degree of effectiveness and efficiency with which the software can be modified by developers, adding new functionalities or fixing existing errors. The most violated maintainability rules in this analysis are the following ones: a) Local variables, parameters, and methods should comply the Java naming convention, b) useless assignments for a local variable, and c) empty statements must be removed. Note that these rules has small impact in the rating because their severity is minor. On contrary, some of the more severe maintainability issues found are the following ones: a) clone implementation should not be overridden, and b) Child class fields should not shadow parent class fields.

In Table 2 we show the results for each framework order by their rating and effort. None of them get the best qualification nor the second one: the majority of the frameworks obtain the two worst qualification (D and E). As we can see, the larger frameworks are more likely to have severe issues. In addition, sometimes there is a large difference in number of issues between two frameworks with the same rating. The reason is that the severity prevails over number in the found issues. Regarding the effort needed to fix all the issues, Keel needs the larger expected effort: it will take more than five years of work to solve them all, however, we must consider that Keel has more than half a million lines of code. In contrast, ssGA only has four blocker issues with a total effort of less than two workdays, although it has less than one thousand lines of code.

3.2 Security

The security characteristic measures the degree to which a software protects information and data, i.e., the probability that the software has a security risk. The most common issues found in this study regarding security are: a) mutable fields should not be public static, b) do not use deprecated code and c) A method or attribute should be protected. Note again that these rules have a small impact in the rating because their severity is minor. On the other hand, some of the more severe violated rules founded are the following ones: a) Credentials may

Table 2. Rating, Effort and issues per severity for maintainability.

Project	Rating	Effort(min)	Issues			
			Blocker	Critical	Major	Minor
Mahout	C	1,057	0	0	25	118
Watchmaker	D	1,574	0	25	149	268
JCLEC	D	5,164	0	11	248	344
moea-frame	D	10,609	0	170	182	683
ssGA	E	728	4	0	12	133
Jenes	E	3,879	21	14	121	403
jMetal	E	21,111	10	73	642	1,790
ECJ	E	22,173	88	168	923	1,277
WEKA	E	117,360	48	996	5,134	13,888
Keel	E	903,509	324	1,757	17,084	62,649

be hard-coded, b) do not call the Java garbage collector explicitly and let the virtual machine manage it, and c) do not override object finalize method.

In the results showed in Table 3, we can observe that two frameworks, Weka and Keel, have blocker issues, and consequently they have the worst rating. They need a huge effort to fix all their issues related to security, specially Keel will take an expected year of work of one full time person. On the other hand, there are three frameworks (Mahout, ssGA and JCLEC) with a C rating because they do not have blocker and critical issues. Note that Watchmaker with only two issues has a D qualification due to the high severity of one of them (critical). As a consequence, Watchmaker could be improved in a short time, less than an hour, so it seems easy to improve its quality.

Table 3. Rating, effort and issues per severity for security.

Project	Rating	Effort(min)	Issues			
			Blocker	Critical	Major	Minor
Mahout	C	70	0	0	1	6
ssGA	C	280	0	0	27	1
JCLEC	C	660	0	0	21	43
Watchmaker	D	30	0	1	0	1
moea-frame	D	1,340	0	3	110	9
jMetal	D	2,185	0	2	64	122
Jenes	D	3,815	0	3	310	38
ECJ	D	14,245	0	5	274	1,093
WEKA	E	44,840	8	35	2,635	1,692
Keel	E	180,700	48	35	10,218	6,815

3.3 Performance

The performance efficiency characteristic measures the amount of computing resources (CPU, memory, I/O,...) used under some specific conditions. The most violated rules are minor issues that affect the performance are the following: a) Use different methods for variable parsing, b) method with a very high cognitive complexity and c) constructor should not be use to instantiate primitive classes. On contrary, some of the more severe performance issues detected are the following ones: a) Use of sleep instead wait when a lock is held, b) constructors with a high number of parameters, and c) do not use synchronized data structures when it is not needed.

In Table 4 we show the result of the analysis. At a first glance, we can observe that three frameworks need around one hour to solve all performance issues. On the other hand, the biggest projects need much more effort, specifically more than

five months to solve all the performance issues. In addition, all the frameworks have a D qualification, except Watchmaker, which obtains the best qualification with a C because it has not blocker or critical issues in performance.

Table 4. Rating, effort and issues per severity for performance.

Project	Rating	Effort(min)	Issues			
			Blocker	Critical	Major	Minor
Watchmaker	C	40	0	0	3	0
ssGA	D	58	0	2	2	0
Mahout	D	64	0	5	0	0
JCLEC	D	398	0	20	12	2
Jenes	D	656	0	17	19	6
moea-frame	D	1,828	0	82	14	5
jMetal	D	2,783	0	59	66	42
ECJ	D	7,659	0	188	89	72
WEKA	D	83,507	0	1,152	2,344	510
Keel	D	230,037	0	2,864	6,275	3,914

3.4 Reliability

Reliability measures the probability of failure-free software operation for a period of time. The most common violated rules found in our analysis in regards to reliability are the following ones: a) do not throw generic exceptions, b) do not write static fields from instance methods, and c) cast operators before to perform maths operations. On the other hand, some of the more severe reliability issues are the following ones: a) resources should be closed, d) zero could be a possible denominator, and c) do not use a threat instance as a monitor.

All the results for the reliability analysis are shown in Table 5. As we can see in the table, most of packages have a bad mark due to the existence of blocker issues. Despite that, three frameworks (Mahout, ssGA and Watchmaker) have a qualification of B, C and D, respectively. In the case of Mahout, it only has a minor issue and it could be resolved in only five minutes. Then, the ssGA has 18 major issues and they could be resolved in five hours. Finally, Watchmaker has only two issues but one of them is critical. From the point of view of the reliability, Keel and WEKA seem to need a deep improvement in reliability according to these evidences, because of the large number of issues and the long time needed to fix them.

Table 5. Rating, effort and issues per severity for reliability.

Project	Rating	Effort(min)	Issues			
			Blocker	Critical	Major	Minor
Mahout	B	5	0	0	0	1
ssGA	C	360	0	0	18	0
Watchmaker	D	10	0	1	0	1
Jenes	E	770	2	7	36	14
moea-frame	E	1,355	7	61	31	28
JCLEC	E	1,560	1	3	6	98
ECJ	E	1,868	9	21	99	58
jMetal	E	2,941	32	111	20	22
Keel	E	29,997	495	348	1,319	728
WEKA	E	48,238	144	300	2,051	110

4 Summary of Results and Global Discussion

In the previous sections we have presented the results obtained after analysing four key quality aspects. With the aim in mind of ranking the projects with a final qualification, we introduce an average rating summarizing the ratings of all analysed characteristics.

Given a rating $r_c \in \{A, B, C, D, E\}$ for a particular characteristic $c \in$ C, we assign a value $r'_c \in \mathbb{N}$ in the range [1,5] such that the ratings $\{A, B, C, D, E\}$ corresponds to values $\{1, 2, 3, 4, 5\}$, respectively. After that, we average the r'_c to obtain an overall rating $r_o \in \mathbb{R}$ in the range [1.0, 5.0]. We compute r_o using the following equation:

$$r_o = \frac{\sum_{c \in C} r'_c}{|C|} \tag{1}$$

Finally, we translate r_o to an ordinal scale in the range A-D. We assign an overall rating A when $r_o \in [1.0, 2.0)$, an overall rating B when $r_o \in [2.0, 3.0)$, an overall rating C when $r_o \in [3.0, 4.0)$, and an overall rating D when $r_o \in [4.0, 5.0]$. In order to obtain a more precise overall rating we add a '+' symbol when r_o is in the first tertile of the range and a '-' symbol when the value is in the last 33%.

In Table 6 we show the ratings of each characteristic, the overall rating calculated as explained above, and the debt of the project. The debt is calculated as the percentage of *estimated* effort needed to fix all found issues divided by the total *estimated* effort to implement the project. When we compare the ratings obtained for all the characteristics, reliability obtains the worst results with 7 MLOFs rated with an E, followed by maintainability with 6 MLOFs rated with an E. In addition, in Figure 1 we show the percentage of estimated effort to fix the found issues per characteristic. In the figure, it can be seen that maintainability requires almost 50% total effort in most frameworks.

Table 6. Final qualification of each framework.

Project	Rating				Overall	
	maintainability	Security	Reliability	Performance	Rating	Debt (%)
Mahout	C	C	B	D	C+	3.18
Watchmaker	D	D	D	C	C-	0.98
ssGA	E	C	C	D	C-	7.07
moea-frame	D	D	E	D	D+	1.49
JCLEC	D	C	E	D	D+	1.56
jMetal	E	D	E	D	D	2.24
Jenes	E	D	E	D	D	2.64
ECJ	E	D	E	D	D	2.85
WEKA	E	E	E	D	D-	2.77
Keel	E	E	E	D	D-	7.66

None of the MLOFs have an E rating in performance, this indicates that the authors of the MLOFs has taken care about the performance efficiency of the algorithms. Actually, performance issues should directly affect the execution times reported in the articles, that is why we guess the performance aspect was carefully treated. In Figure 1 we confirm our assumption being the performance and reliability the characteristics that require less remediation effort.

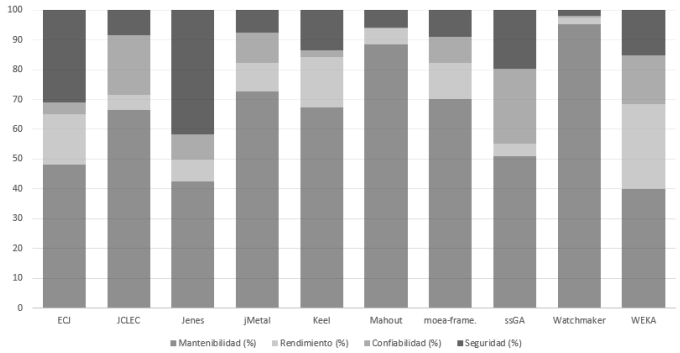

Fig. 1. Estimated effort (%) to fix the found issues per characteristic and framework.

Overall, the best project in our comparison is Mahout with a rating of C+, followed by ssGA and Watchmaker rated with C-. It seems that MLOFs with a low complexity per file are less prone to issues. In order to confirm our expectation, we performed a Spearman's rank correlation test between the complexity per file and the overall rating for the MLOFs. We obtained a high coefficient $\rho = 0.908$, what means that the more the complexity in a file the more the probability to find an issue. In Table 6 we also showed the debt of each project. This measure removes the size component of a MLOF, so it shows the *relative* effort needed to fix the issues found. In this regard, Keel is the worst ranked framework with 7.66% and Watchmaker is the best with less than 1% of debt. Something that is not surprising, because Keel has the highest value in complexity per file, meanwhile Watchmaker has the lowest value.

5 Conclusions and Future Work

In this paper we have studied some important aspects of software products such as maintainability, reliability, performance efficiency and security characteristics of ten well-known machine learning and optimization frameworks. Note that these characteristics are part of the product quality model proposed by the ISO 25010 standard. After performing the analysis and classification of the defects detected, we are more concious about the current state of each development for the characteristics studied. The analysis revealed that, overall, none of the frameworks obtain the two best overall qualifications (A and B). This may be a concern for the researchers who had used these frameworks in their experiments. Particularly, the best framework of our comparison is Mahout with a rating of C+, but if we take a look to the debt ratio, Watchmaker is the best in terms of effort needed to solve the found issues.

In conclusion, we can claim that maintainability is by far the most ignored aspect of the existing packages. A bad maintainability rating means that is more difficult for other developers to contribute or extend the frameworks. Regarding the reliability and the performance, they need a quite similar amount of time, between them, to improve their quality. From our point of view, they are more relevant when we focus on MLOFs because we prefer fast techniques that gener-

ate reliable solutions. Finally, about security we think that it is not as relevant as the rest of the characteristic because experiments are usually performed in a local and controlled environment. However, if researchers are going to use these packages in industrial real cases it could become a major issue.

There is a number of interesting findings we want to validate in a near future. We plan to study the rest of characteristics proposed in the ISO 25010 standard that we did not consider in this work such as functionality, portability, usability, and compatibility. In addition, we want to confirm that these frameworks could be improved if we fix the issues found. In this way, we plan to suggest fixes to the detected issues and then, perform a new analysis on the new version of the framework. After the analysis, we would know whether we improve all quality aspects or at least some of them. Finally, we will propose some *pull requests* to make our changes available for all the community if they are accepted by the frameworks' authors.

Acknowledgements

We would like to say thank you to all authors of these frameworks that make research easier for all of us. This research has been partially funded by CELTIC C2017/2-2 in collaboration with companies EMERGYA and SECMOTIC with contracts #8.06/5.47.4997 and #8.06/5.47.4996. It has also been funded by the Spanish Ministry of Science and Innovation and /Junta de Andalucía/FEDER under contracts TIN2014-57341-R and TIN2017-88213-R, the network of smart cities CI-RTI (TIN2016-81766-REDT) and the University of Malaga.

References

1. Alcalá-Fdez, J., Fernández, A., Luengo, J., Derrac, J., García, S., Sánchez, L., Herrera, F.: KEEL data-mining software tool: Data set repository, integration of algorithms and experimental analysis framework. Journal of Multiple-Valued Logic and Soft Computing **17**(2-3) (2011) 255–287
2. Hall, M.A., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The WEKA data mining software: an update. SIGKDD Explorations (2009) 10–18
3. Parejo, J.A., Ruiz-Cortés, A., Lozano, S., Fernandez, P.: Metaheuristic optimization frameworks: A survey and benchmarking. Soft Computing **16** (2012) 527–561
4. Durillo, J.J., Nebro, A.J.: JMetal: A Java framework for multi-objective optimization. Advances in Engineering Software **42**(10) (2011) 760–771
5. Wagner, S.: Software product quality control. (2013)
6. Luke, S.: ECJ evolutionary computation library (1998) Available for free at http://cs.gmu.edu/∼eclab/projects/ecj/.
7. Ventura, S., Romero, C., Zafra, A., Delgado, J.A., Hervás, C.: JCLEC: A Java framework for evolutionary computation. Soft Computing **12**(4) (2008) 381–392
8. Luigi Troiano, Davide De Pasquale, P.M.: Jenes genetic algorithms in java (2006) http://jenes.intelligentia.it.
9. The Apache Software Foundation: Apache Mahout Project. (2014) https://mahout.apache.org.
10. Hadka, D., Reed, P.: Borg: An Auto-Adaptive Many-Objective Evolutionary Computing Framework. Evolutionary Computation **21**(2) (2013) 231–259
11. Alba, E.: ssGA : Steady state ga (2000) http://neo.lcc.uma.es/software/ssga.
12. Dyer, D.W.: Watchmaker framework for evolutionary computation (2006) https://watchmaker.uncommons.org/.