Automated Trainability Evaluation for Smart Software Functions

Ilias Gerostathopoulos*, Stefan Kugele*, Christoph Segler†, Tomas Bures‡ and Alois Knoll*
*Department of Informatics, Technical University of Munich, Garching b. München, Germany
Email: {ilias.gerostathopoulos, stefan.kugele}@tum.de, knoll@in.tum.de
†BMW Group Research, New Technologies, Innovations, Garching b. München, Germany
Email: christoph.segler@bmwgroup.com
‡ Charles University in Prague, Czech Republic
Email: bures@d3s.mff.cuni.cz

***Abstract*—More and more software-intensive systems employ machine learning and runtime optimization to improve their functionality by providing advanced features (e. g. personal driving assistants or recommendation engines). Such systems incorporate a number of *smart software functions* (SSFs) which gradually learn and adapt to the users' preferences. A key property of SSFs is their ability to learn based on data resulting from the interaction with the user (implicit and explicit feedback)—which we call *trainability*. Newly developed and enhanced features in a SSF must be evaluated based on their effect on the trainability of the system. Despite recent approaches for continuous deployment of machine learning systems, trainability evaluation is not yet part of continuous integration and deployment (CID) pipelines. In this paper, we describe the different facets of trainability for the development of SSFs. We also present our approach for automated trainability evaluation within an automotive CID framework which proposes to use automated quality gates for the continuous evaluation of machine learning models. The results from our indicative evaluation based on real data from eight BMW cars highlight the importance of continuous and rigorous trainability evaluation in the development of SSFs.**

***Index Terms*—trainability, smart software functions, continuous deployment**

I. Introduction

More and more computing systems feature advanced software-enabled functionalities that make them be perceived as "smart" by their users. Smartness is achieved via various machine learning and runtime optimization capabilities that aim to increase the value delivered to the users. Indeed, seen as the ability to learn and improve over time, smartness is increasingly a must-have property of many modern systems.

In an attempt to understand and aid the development of smart systems, we coin the term *trainability* to describe the key quality property of smart systems: the ability of a system to accurately, quickly, cost-effectively, and robustly learn users' behaviors and preferences with the goal to maximize the value delivered to them. In a way, trainability is the opposite notion of *learnability*, which focuses on the ability of *users* to learn to use a system [1].

As an example from the automotive domain, consider a windscreen wiper that supports two modes of operation: (i) *interval-based*, i. e., there are three levels with fixed intervals between wipes, and (ii) *automatic*, i. e., the intervals are computed based on rain intensity. In practice, customers perform in either case additional manual wipes. A *smart* windscreen wiper would try, e. g., via reinforcement learning, to adapt to the driver's preferences and minimize the number of manual interventions. Its trainability depends on how quickly, accurately, and effectively it learns such preferences.

The research leading to these results has received funding from the ECSEL Joint Undertaking (JU) under grant agreement No 783221.

Considering trainability as an important quality property of smart systems, along with traditional properties of software systems such as availability, maintainability, and performance, two important questions arise: First, how can we quantify trainability so that different versions of a system can be compared on their effect on this property? Second, how can we assess the effect of software development and evolution (integration of new or enhanced features) on the trainability of a smart system? Here, we are particularly interested in automation that allows for quickly and reliably providing feedback to developers regarding the effect of their software changes on the trainability of the system under development.

To answer the above questions, in this paper we contribute by (i) elaborating on five key facets of trainability (*solution quality*, *convergence*, *overhead*, *robustness*, and *effect*) and associated metrics for their evaluation; (ii) reporting on our on-going attempt for embedding trainability evaluation in the development of smart systems in the automotive domain.

II. Trainability Facets and Metrics

In this work, we propose five **SCORE** *trainability facets* that are of particular interest and importance during the development of smart systems:

- ***S**olution quality*, i. e., what the quality of the learned model or optimized function is;
- ***C**onvergence*, i. e., how efficient the learning is in terms of time or algorithmic steps to reach its goal;
- ***O**verhead*, i. e., what the cost of learning and execution is in terms of resources (e. g. memory, CPU);
- ***R**obustness*, i. e., how stable the quality of the learned model or optimized function is when trained with different input data or under different system conditions;

TABLE I
METRICS OF SOLUTION QUALITY TRAINABILITY FACET

Technique	Metrics
Stochastic search	
↪ Single-objective	distance from global optimum, distance from best known solution, ...
↪ Multi-objective	Hypervolume, inverted generational distance, generated spread, Euclidean distance from ideal solution,...
Supervised Learning	
↪ Classification	accuracy, balanced accuracy, precision, recall, F1 score, area under the ROC, …
↪ Regression	root mean squared error (RMSE), mean absolute error, coefficient of determination (R^2), …
Unsupervised Learning	
↪ Clustering	*Internal*: silhouette coefficient, Dunn's index, … *External*: adjusted Rand index, Jaccard index, …

- ***Effect***, i. e., what the effect of the learning is on the actual customer satisfaction and adoption.

Solution quality can be quantified via various existing metrics (Tab. I). The selection of one or more metrics depends on the specific technique used to develop a smart system. For instance, in systems employing single-objective stochastic optimization, the distance of the reported solution to the optimal one or to the best so-far known solution can be measured [2]. Similarly, in multi-objective optimization, different quality indicators can be used (e. g. hypervolume indicator) [3]. Various well-known metrics exist also for the evaluation of classification (e. g. accuracy, F1 score) and regression (e. g. RMSE) algorithms. Finally, clustering algorithms are typically evaluated using *internal* (e. g. Silhouette coefficient) or *external* (e. g. Jaccard index) metrics, depending on the availability of ground truth [4], [5]. *Convergence* refers to the time to reach an acceptable (potentially optimal) solution and can be quantified either via the number of steps or iterations an algorithm needs to reach this solution or via the computation time, which is implementation-specific. Similarly, *overhead* refers to the computational or memory resources that are needed for the solution to be executed and is highly implementation-specific since specific implementations of algorithms incur more overhead than others. *Robustness* refers to the differences in solution quality when training a machine learning model with different input data or optimizing a function under different operational conditions—with higher robustness leading to smaller differences. Robustness can be measured via evaluating the quality of multiple "runs" of the smart system. The challenge is how to generate different input datasets or ensure that the system goes through diverse operational conditions. Finally, *effect* refers to the value delivered to the users as a result of enhanced functionality. Though this alone is an important property of *any* function, we consider it also an important facet of trainability since it is particularly difficult to assess how smart a system should be in order to be accepted by the potential users. Effect can be measured only by deploying the smart system to real cars and collecting feedback from users (e. g. usage rates, ratings).

The SCORE facets and their analysis above show that trainability can be considered as a universal property of smart systems that employ machine learning and self-optimization.

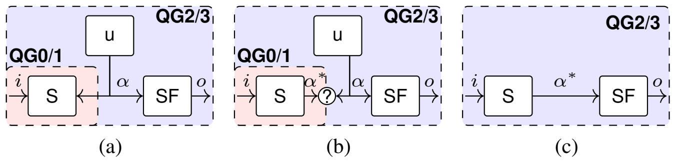

Fig. 1. Reference architecture of a SSF with user u, actions α, predicted user action α^*, input i, and output (result) o: (a) Training phase, (b) test phase; the component ? compares the user action α with the predicted action α^*. (c) The SSF S is active and controls the SF.

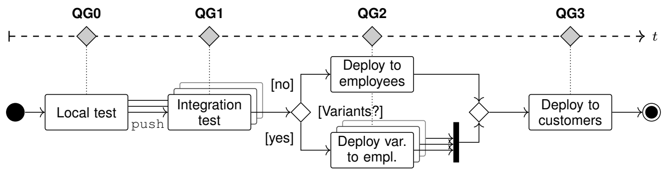

Fig. 2. Trainability Evaluation Framework for Automotive CID.

III. TRAINABILITY EVALUATION FRAMEWORK

In our initial attempt to assess the effect of software development on the trainability of a smart system, we focused on *Smart Software Functions* (SSFs) in the automotive domain. In automotive, SSFs are meant to enhance drivers' and passengers' comfort by learning and automating certain manual tasks such as turning on/off seat heating, opening/closing the driver's window, and changing the driving mode to sport/comfort.

Fig. 1 depicts the reference architecture of a SSF in three different phases. In the *training phase* (cf. Fig. 1a), the user u is interacting with the classic non-smart *Software Function* (SF). Based on these user actions α, the *Smart* learning component (denoted as S) trains a classifier to predict when to apply a user action based on vehicle data (e. g. predict "start seat heating" when the temperature is low). In the *testing phase* (cf. Fig. 1b), the architecture changes since S now outputs predicted user actions α^* that are compared to the actual user actions, which are still used for controlling the SF. Based on the predicted and actual actions, we can measure certain trainability facets without interfering with the actual system. In the *active phase* (cf. Fig. 1c), the SSF is completely active and the predictions from S are now used for controlling SF.

Given the above reference architecture and phases, we propose to evaluate the trainability of different revisions of SSFs using a *Continuous Integration and Deployment* (CID) approach that provides fast and reliable feedback to the developers on the effect of their changes on the SCORE trainability facets (cf. Fig. 2). In particular, the following four *quality gates* (**QG**) ensure that only revisions satisfying certain trainability criteria proceed to the next stages.

QG0 In this step, the developer follows guidelines that prescribe which trainability criteria should be satisfied *before* pushing a revision. For example, a criterion could be that the classification accuracy of S should be at least 60% and not lower than the accuracy of the previous revision. Obviously, the validity of such checks depends on the availability of data that can be accessed locally by the developer. In general, we assume that the developer has access to a subset of historical vehicle data in order to develop their SSFs.

QG1 This QG gets activated once a revision is pushed to the CID pipeline. In this step, the SSF can only run in the training and test phase, due to the execution on recorded datasets without any user in place yet (cf. Figs. 1a and 1b). Here, apart from the solution quality of S, its robustness and overhead are also tested. For instance, robustness can be checked by measuring the standard deviation of a solution quality metric such as accuracy. Overhead can also be realistically measured in terms of CPU cycles or RAM used in training and testing since Hardware-in-the-Loop testing environments are present. Concretely, Listing 1 shows a possible specification of the checks to be performed in QG1. First, the specification defines the type and size of input data to be used in the training and testing phases of S. Data can be retrieved belonging to either users or cars (*type*) and over a prescribed duration. For example, Listing 1 specifies that datasets from 8 users driving their car for 90 days should be used. Each dataset should be split into 5 folds and training should be performed using all the possible combinations of 4 consecutive folds (5-fold validation) and tested on the remaining fold. This creates 40 (8*5) tests. Here, the metric of solution quality is balanced accuracy which can measure the accuracy even in imbalanced datasets that are common for SSFs, as some functions are only rarely used. This value has to be larger than 60% for each test to pass. For the solution quality facet to be satisfied, more than half of the total tests should pass and the number of passing tests should not be less than the equivalent number in the previous revision. The method shown in this example is just a simple way to identify trainability regressions; Also more sophisticated methods (e. g. *t*-tests) can be easily specified. In this case, the specification also denotes that in each test the size of the classification model should not exceed 200 MB in order to satisfy the overhead criterion. Finally, as a criterion for robustness, the standard deviation of balanced accuracy values from all tests should be less than 0.1. Note that convergence cannot be evaluated at this step since this can only be done with an increasing amount of data, and for the effect, an interaction with a real user is necessary.

QG2 Once the revision passes from QG1, it gets deployed on company cars that are driven by employees of the automotive company, which can be considered as "beta testers" in this setting. We distinguish between deployment of a single revision and deployment of two or more revisions that contain variants of a SSF that should be compared against each other via A/B testing [6]. For the latter case, we assume that an online experimentation module decides which variants are deployed on each car and is responsible for gathering data for each deployment variant. At this step, along with the other facets, it becomes possible to evaluate the convergence and effect facets of a revision. Also, it now becomes possible to test not only S but the whole SSF in the following way: First, the

```
input:
  type:              user
  count:             8
  recorded_time:     90 days
  train_test_split:  5-fold
solution_quality:
  balanced_accuracy:
    - each: value >= 0.60
    - all:  passed_count >= 0.5 * total_count
    - all:  passed_count >= passed_count_previous_revision
convergence:         ~ # not applicable for QG1
overhead:
  model_size:
    - each:          value < 200 MB
    - all:           passed_count == total_count
robustness:
  balanced_accuracy:
    - all:           standard_deviation < 0.1
effect:              ~ # not applicable for QG1
```

Listing 1. Definition of QG1.

```
input:
  type:              user
  count:             100
  evaluation:        3 days
  train_test_split:  2/3
  active:            7 days
  filter:
    department:      <COMPANY>
    model:           <COMPANY>
solution_quality:    # inherited from QG1
convergence:
  training_time:
    - each:          3 evaluations <= value <= 5 evaluations
overhead:            # inherited from QG1
robustness:          # inherited from QG1
effect:
  percentage_used:
    - each:          value > 0.60
    - all:           passed_count >= 0.9 * total_count
```

Listing 2. Definition of QG2.

SSF is in passive mode (cf. Figs. 1a and 1b) and goes through a number of *evaluations* where S is first trained for a certain amount of time and then tested. After a number of evaluations, S converges to the acceptable level of solution quality. At this point, SSF changes to active mode (cf. Fig. 1c) whereby S is used as input for user actions and the effect of the revision is evaluated. All evaluation results are sent to the company CID servers to allow for testing the metrics of QG2. Concretely, Listing 2 specifies the case where a revision should be tested with 100 employees driving a specific car model and working in a specific department. Here, a single evaluation takes 3 days and is split in 2 days of training time and 1 day of testing time. The convergence constraint is that S should reach the solution quality criterion of 60% (as specified in the previous QG) within at least 9 days and at most 15 days. After this, the system should stay in active mode (where the effect can be measured) for 7 days. In this case, the effect is measured by the percentage of time SSF is on and not manually deactivated by the user (for more than 60% for at least 90% of the users).

QG3 In the last step, a single variant (if many) of the SSF revision is deployed to customer cars. The approach to trainability assessment is very similar to the previous step since the SSF should again undergo a train-test-active cycle to make the evaluation of all the SCORE facets possible. Here though there is also the opportunity for longer active phases and continuous monitoring of effect metrics.

IV. Demonstration and Indicative Evaluation

In this section, we demonstrate our approach on the development of an *Automatic Window Opener* (AWO). AWO predicts when to open or close the driver's window based on the user's actions in certain situations delineated by car data such as inside/outside temperature, road type, or location of the car key. We assume that the development of AWO goes through two revisions: in the first revision, S is trained using a Support Vector Machine (SVM) classifier on all the available car data as input; in the second, S is trained with the same classifier, but in this case based on 30 inputs from the car data selected by a feature selection algorithm (here, Fisher Score). For our demonstration, we configured QG1 as specified in Listing 1. We assume that the developer has access to a single user dataset when developing and testing locally (QG0), while QG1 has access to 8 user datasets. These datasets originate from eight BMW cars that have been equipped with hardware loggers, collecting all messages which were sent over the internal communication networks. As specified in Listing 1, each dataset contains vehicle data recorded over 90 days.

When the first revision (SVM without feature selection) is pushed to the CID pipeline, 9 out of 40 tests passed the solution quality criterion of balanced accuracy above 60% and only 15 out of 40 tests passed the overhead criterion of model size below 200 MB; thereby the revision does not pass QG1. AWO is then refined (second revision) and tested locally (QG0). After passing QG0, this revision is pushed to the CID pipeline. This revision now passes 24 out of 40 tests (60%) and hence satisfies the solution quality criterion. At the same time, all runs pass the overhead criterion and the robustness checks, since the model size of each run is below 200 MB and the standard deviation of the 40 balanced accuracy values is 0.0867 which is below 0.1 as specified in the robustness. Since all facets are satisfied, the revision passes QG1.

Our indicative evaluation shows that the different steps and QGs in the proposed CID pipeline were able to quickly detect the poorly performing first revision and prevent it from being deployed to real cars. We also show that it is feasible to capture all the SCORE facets in our framework and evaluate each one of them when and where it is most applicable.

V. Related Work

To the best of our knowledge, there is only little work on CID pipelines in the automotive domain. Vöst and Wagner [7] report on a test selection approach supporting CI. The authors consider non-smart SFs, in particular, the adaptive cruise control and show a reduction of executed tests. Their approach covers QG1 for non-smart functions. Knaus et al. [8] and van der Valk et al. [9] highlight the challenges of CI pipelines in the automotive development process. They underline the importance of tools and *automation*, especially in the building and integration stages going along with the presented CID approach of Fig. 2. Moreover, they highlight the importance of contracts within an automotive ecosystem.

The proposed notion of "trainability" can be seen in contrast to the "learnability" notion in software quality characteristics. In particular, ISO/IEC 25010:2011 defines learnability as the "degree to which a product or system can be used by specified users to achieve specified goals of learning to use the product or system with effectiveness, efficiency, freedom from risk and satisfaction in a specified context of use" [1]. Likewise, we consider trainability as a quality attribute of a SSF. However, trainability features the ability of a SSF to learn user preferences and not the ability to be learned.

Similarly to Renggli et al. [10], we use the YAML file format to specify quality gates facilitating an embedding in commonly used CI/CD systems such as Travis CI. In comparison, we do not only focus on machine learning models but generalize to a variety of solution quality trainability facets holistically. Moreover, the proposed CID approach does not only consider early phases but spans the whole product life-cycle from development to deployment of SSFs to customers.

VI. Conclusion

In this paper, we proposed the notion of SCORE—*S*olution quality, *C*onvergence, *O*verhead, *R*obustness, *E*ffect—to define the different facets of trainability for the development of smart software functions. Based on these trainability facets, we presented our approach for automated trainability evaluation within an automotive continuous integration and deployment framework. We demonstrated the applicability of our approach on the development of a smart function in the automotive domain using real vehicle data. In future work, we would like to introduce more metrics in each facet of trainability and evaluate QG2 and QG3 on real users.

References

[1] ISO/IEC, "ISO/IEC 25010:2011 systems and software engineering – systems and software quality requirements and evaluation (square) – system and software quality models," ISO/IEC, Tech. Rep., 2011.

[2] R. L. Rardin and R. Uzsoy, "Experimental Evaluation of Heuristic Optimization Algorithms: A Tutorial," *Journal of Heuristics*, vol. 7, no. 3, pp. 261–304, May 2001.

[3] S. Wang, S. Ali, T. Yue, Y. Li, and M. Liaaen, "A practical guide to select quality indicators for assessing pareto-based search algorithms in search-based software engineering," in *Proc. of ICSE '16*. Austin, Texas: ACM Press, 2016, pp. 631–642.

[4] M. Z. Rodriguez, C. H. Comin, D. Casanova, O. M. Bruno, D. R. Amancio, L. d. F. Costa, and F. A. Rodrigues, "Clustering algorithms: A comparative approach," *PLOS ONE*, vol. 14, no. 1, pp. 210–236, 2019.

[5] O. Arbelaitz, I. Gurrutxaga, J. Muguerza, J. M. Pérez, and I. Perona, "An extensive comparative study of cluster validity indices," *Pattern Recognition*, vol. 46, no. 1, pp. 243–256, Jan. 2013.

[6] R. Kohavi and R. Longbotham, "Online controlled experiments and A/B testing," in *Encyclopedia of Machine Learning and Data Mining*, C. Sammut and G. I. Webb, Eds. Springer, 2017, pp. 922–929.

[7] S. Vöst and S. Wagner, "Trace-based test selection to support continuous integration in the automotive industry," in *Proc. of CSED@ICSE 2016*. ACM, 2016, pp. 34–40.

[8] E. Knauss, P. Pelliccione, R. Heldal, M. Ågren, S. Hellman, and D. Maniette, "Continuous integration beyond the team: A tooling perspective on challenges in the automotive industry," in *Proc. of ESEM 2016*. ACM, 2016, pp. 43:1–43:6.

[9] R. van der Valk, P. Pelliccione, P. Lago, R. Heldal, E. Knauss, and J. Juul, "Transparency and contracts: continuous integration and delivery in the automotive ecosystem," in *Proc. of ICSE'18*. ACM, 2018, pp. 23–32.

[10] C. Renggli, B. Karlas, B. Ding, F. Liu, K. Schawinski, W. Wu, and C. Zhang, "Continuous integration of machine learning models with ease.ml/ci: Towards a rigorous yet practical treatment," *CoRR*, vol. abs/1903.00278, 2019.