A machine learning approach to support a bottom-up evolution of microservice compositions based on the choreography of BPMN fragments

Anonymized

Abstract. Business processes (BPs) are commonly used by organizations to describe their goals. However, the existent decentralization found in many organizations forces them to build such BPs by coordinating distributed and fragmented BPs. Within this context, microservices arise as a very interesting and convenient way to address the implementation of such processes due to their low coupling characteristic. In this case, the coordination of such fragmented BPs is usually achieved by means of event-based choreographies. However, one of the main challenges to be faced by choreographies is their evolution due to the complexity that introduces the need of integrating changes among autonomous and independent partners. We face the challenge of evolving a microservice composition that is globally defined in a BPMN model but executed through a choreography of BPMN fragments. To do so, we present a solution based on machine learning. We analyse the performance of different classifiers to assist in the selection of an adaptation plan to maintain the consistency of a choreography when a participant makes a change from its local perspective. The selected classifier is integrated into a microservice architecture to validate its use from a pragmatic point of view.

Keywords: microservices, composition, evolution, machine learning

1 Introduction

Business processes (BPs) are the key instrument to organizing and understanding the interrelationships of the different activities required to produce an outcome to the market [1]. However, when these activities are performed in a decentralized way, e.g., by different departments within the same organization, the decoupling characteristic of microservices make them a very interesting and convenient way to implement such processes. Microservice architectures [2] propose the decomposition of applications into small independent building blocks (the microservices) that focus on single business capabilities. Therefore, microservices need to be composed to support the business processes of organizations. To this end, to keep a lower coupling and independence among microservices for deployment and evolution, these compositions are usually implemented by means of event-based choreographies. However, choreographies split the control flow of compositions among the different participant microservices, which makes them hard to analyse and understand when requirements change. Our previous work [3] improves this problem by proposing an approach based on the choreography of BPMN fragments. Business process engineers create the big picture of the microservice composition through a BPMN model. Then, this model is split into BPMN fragments which are executed through an event-based choreography. This composition approach is supported by a microservice

architecture developed to achieve that both descriptions of a composition, the big picture and the split one, coexist in the same system.

This solution introduces two main benefits regarding the microservice composition. First, it facilitates business engineers to analyse the control flow if the composition's requirements need to be modified since they have available the big picture of the composition. Second, it provides a high level of decoupling among the microservices that participate in a composition, since it is implemented as an event-based choreography of independent BPMN fragments. However, this solution introduces a new challenge to be faced: how to evolve a microservice composition that is globally defined in a BPMN model but executed through a choreography of BPMN fragments.

In general, when a process of a system is changed, it must be ensured that structural and behavioural soundness is not violated after the change [4]. When the process is supported by a choreography and changes are introduced from the local perspective of one partner, additional aspects must be guaranteed due to the complexity introduced by the interaction of autonomous and independent partners. For instance, when a partner introduces some change in its part of the process, it must be determined whether this change affects other partners in the choreography as well. If so, adaptations to maintain the consistency and compatibility of the choreography should be suggested to the affected partners. In our microservice composition approach, a change introduced from the local perspective of a microservice needs to be integrated with both the BPMN fragments of the affected partners and the big picture of the composition.

In this work, we propose a machine learning approach to identify the adaptation actions that are needed to maintain the integrity of the BPMN fragment choreography when a developer performs a local change in its BPMN fragment. To do so, we present a catalogue of adaptation rules created from the characterization of local changes done in [5,6]. Next, we train a machine learning classifier to suggest the adaptation rules required by the partners affected by a local change. Thus, the main contribution of this paper is the integration of machine learning techniques with BPMN modelling to support the evolution of event-based choreographies of BPMN fragments when a local change is produced.

The rest of this paper is organized as follows: Section 2 presents the previous work required to understand the current proposal. Section 3 introduces the catalogue of adaptation rules designed to support the evolution of microservices compositions. Section 4 introduces a machine learning classifier trained to assist in the definition of an adaptation plan. Conclusions are commented on in Section 5.

2 Previous work and problem statement

To properly understand our current work, this section presents an overview of the approach presented in [3] to create a microservices composition based on the choreography of BPMN fragments. This approach proposes to create a microservice composition in two main steps (see Figure 1): (1) by creating the big picture of the composition in a BPMN model following an orchestration approach and (2) by splitting such model into BPMN fragments following a choreography approach where each fragment is deployed into a separated microservice.

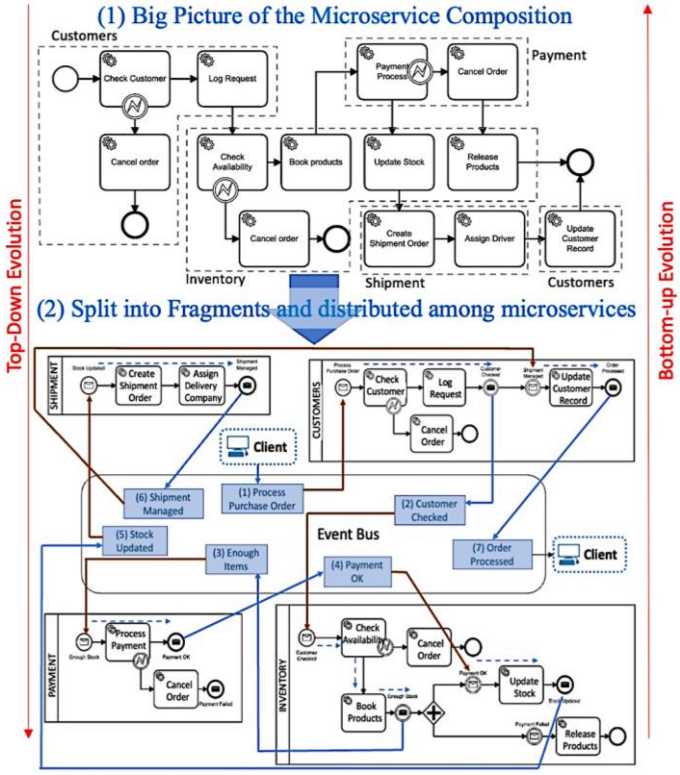

Figure 1. A composition microservice approach based on BPMN fragments.

Let's consider the process for placing an order in an online shop. This process is supported by four microservices: Customers, Inventory, Payment, and Shipment (see topside in Figure 1). The sequence of actions that the microservices must perform is the following: The *Customers* microservice checks the customer data and logs the request. If the customer data is not valid, the process of the order is cancelled and the process is finished. On the contrary, the *Inventory* microservice checks the availability of the ordered items. If there is not enough stock to satisfy the order, the order is cancelled, and the process is finished. On the contrary, this microservice books the requested items and transfers the control flow to the *Payment* microservice. The *Payment* microservice processes the payment with the customer. If the payment fails, the order is cancelled, and the *Inventory* microservice releases the booked items and finishes the process. On the contrary, the *Inventory* microservice updates the stock of the purchased items and transfers the control flow to the *Shipment* microservice, which creates a shipment order and assigns it to a delivery company. Then the *Customers* microservice updates the customer record and informs the customer about the shipment details. Afterwards, the process is finished.

After creating the big picture of the composition, the second step consists in splitting it into BPMN fragments that describe the responsibility of each microservice independently from the others. This is done automatically by a tool we developed [3]. Note

that these fragments capture both the functional responsibilities of a microservice by means of BPMN tasks and the coordination ones by means of BPMN event-based communication elements (e.g., Message start/intermediate catch event, Message end/throwing event, etc.). Each microservice is endowed with a process engine so, at runtime, can oversee the execution of its corresponding BPMN fragment and inform the other participants about it through publishing asynchronous events in a communication bus. In this way, the microservice composition is executed by means of an event-based choreography of BPMN fragments in which microservices wait for an event to execute its corresponding piece of work (see bottom side in Figure 1). Note that microservices do not transfer the control flow to another microservice explicitly but instead by publishing an event (solid blue arrows in Figure 1) in a bus to indicate that a piece of work is completed, and the microservice that is waiting for this event (solid brown arrows in Figure 1) can start the execution of its BPMN fragment.

Problem statement. Our approach allows to evolve the composition following either a top-down or a bottom-up approach as shown by the red arrows in Figure 1. On the one hand, a top-down approach proposes to address these changes from a global perspective by modifying the big picture of the composition and splitting it again into BPMN fragments. This evolution is natively supported in our previous work [3]. On the other hand, a bottom-up approach implies modifying a BPMN fragment from the local perspective of a microservice. This requires synchronizing changes with both, the BPMN fragments of the rest of the microservices and the big picture of the composition. Note that allowing local changes in a microservice composition reinforces the independence among development teams that is demanded by this type of architecture. However, modifying a microservice without considering how changes may affect the rest of the microservices can produce coordination problems that make the composition fail. In this paper, we focus on this type of evolution.

3 A catalogue of adaptation rules to support local changes.

To address the bottom-up evolution approach we identified and characterized different scenarios of modifications from the local perspective of a microservice in a previous work [5, 6]. From this characterization, we have created a catalogue of adaptation rules to support changes in a microservice composition from the local responsibilities of one participant, and maintain, when possible and needed, the functional consistency of the choreography. The whole catalogue is not presented in this paper, but it can be found in a research report [8].

As a representative example, let's consider the following scenario (see Figure 2): the developer of the *Customers* microservice decides to modify its BPMN fragment in such a way that the event "Customer Checked" is not published anymore. Then, the microservice *Inventory*, which is waiting for it, will never start and execute its tasks, and therefore, the microservice composition will never finish.

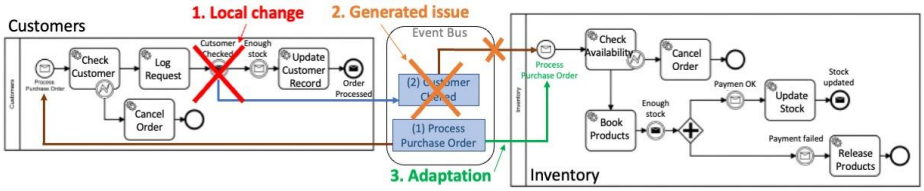

Figure 2. Example of local change, the produced inconsistency, and generated adaptation

The adaptation rule proposed to support this local change is the Rule #1 (see below), which adapts the affected microservices to wait for the event that triggered the modified microservice. Following with the example, the microservice *Inventory* is modified to wait for the event that triggers the *Customers* microservice, i.e., its message start event is modified to listen to the "Process Purchase Order" event instead.

RULE #1

- **Modified element**: End Message Event or Intermediate Throwing Event.
- **Change:** Delete the modified element.
- **Application conditions:** The deleted event does not include data; the event triggered before the deleted one is not generated by the affected microservice.
- **Affected microservice(s):** Those that have a catch event waiting for the message sent by the deleted element.
- **Generated issues:** Affected microservices will never start since their execution depends on the triggering of the event that is just deleted.
- **Proposed adaptation:** Modify the affected microservices to wait for the event that triggered the modified microservice.
- **Impact of the application:** Functional requirements are maintained. Composition flow is altered (some tasks change from sequential to parallel).

As Rule #1 shows, different aspects were considered to define the adaptation rules of the proposed catalogue, these are: 1) the type of modelling element involved in the change, 2) the performed change (i.e., create, update, or delete), 3) the conditions under which the change is performed, 4) the set of microservices that are affected by the change, 5) the composition issues generated by the change, 5) the set of changes proposed to fix the issues, and finally 5) the impact of the proposed changes at the composition level. Note also that some adaptations, such as the one presented above, can maintain the integrity of the functional requirements of a composition (i.e., all the tasks are executed after the change), but at the same time disrupt the execution order (e.g., the flow of some tasks change from sequential to parallel). To face this, an evolution protocol was presented in [4] to allow the manual acceptance of the adaptation rule application by the developers of affected microservices.

4 A machine learning approach to suggest adaptation rules.

Machine learning techniques are used to generate knowledge from data to assist in a decision-making process [7]. In this work, we want to take advantage of the knowledge

obtained from the application of the above-introduced adaptation rules to different case studies. The main goal of this approach is to provide a solution that automate the suggestion of an adaptation rule when a microservice performs a local change in its BPMN fragment and affects the coordination with the rest of partners. To this end, this section presents the machine learning techniques used (subsection 4.1), the dataset used and how it has been encoded (subsection 4.2), the procedure followed to train the selected models (subsection 4.3), and finally the evaluation performed over these models (subsection 4.4).

4.1 Selection of a machine learning technique

There are many machine learning techniques that can be used to reach the goal proposed in this work [7]. To select one, we need to identify first the type of prediction model that is required. From a general point of view, we can find two types: (1) supervised machine learning, whose goal is to learn some rules to predict a correct solution for the input data; and (2) non-supervised training machine learning, in which there is no solution to predict, and the goal is to find some type of structure or relation between the data contained in a dataset. Considering that we want to predict the adaptation rules required to solve local change issues we focused on the first type, i.e., on the supervised machine learning.

Second, it is required to identify the type of task to be done by the supervised machine learning technique. There are two types of tasks: (1) classification tasks, where the result to be predicted is a nominal value defined by two or more classes, and (2) regression tasks, where the result to be predicted is a numerical value. Since we want to predict a nominal value (the adaptation rule to be applied) we focused on classification tasks.

The most used techniques to perform classification tasks are [7]: k-NN (Nearest Neighbours), perceptron, neural networks, decision trees, and SVM (Support Vector Machines). SVM and perceptron require the input cases to be linearly separable. In addition, they can be used when there are only two classes to predict. In our case, the classes to predict (the adaptation rules) are not linearly separable, and there are more than two rules to predict. Thus, these two techniques cannot be used for our purpose. In addition, we left the neural networks techniques out for further work since they need a huge amount of input cases to train a machine learning model. In this first version of the machine learning approach, we wanted to use learning models that could be applied to the dataset we had from the application of adaptation rules to some case studies. Thus, we selected the following two techniques: k-NN and decision trees.

4.2 Dataset and encoding

Machine learning techniques need to be trained by using a dataset[1], which is a collection of data pieces that can be treated by a computer as a single unit for analytic and prediction purposes. In this work, we used a dataset with a total of 272 input cases. This dataset is an artificial dataset created by us. We have simulated 272 local changes in

[1] The dataset, the considered case studies, and the scripts used to train the machine learning models are available at: https://github.com/microserviceresearch/ml-microservice-composition-evolution

four different microservice composition based on our approach, and we have characterised each modification using the feature vector created in the analysis phase. Then, we have selected manually the adaptation rule required to solve the inconsistencies generated by the local change. Therefore, the main goal of the machine learning algorithm is to use the created dataset to identify patterns in order to be able to automatically predict and select the adaptation rule required by a local change.

To understand input cases, the machine learning techniques require them to be in a specific format. Most of the machine learning techniques are designed to process feature vectors as inputs [9]. Feature vectors are known to be the ordered enumeration of characteristics that describe the object being observed [10]. In this work, the object being observed is a local change made in the BPMN fragment of a microservice. Therefore, the machine learning technique requires that these changes are encoded as a feature vector. These vectors must include all the features that can help to describe the observed object as accurately as possible. The more information we include, the better predictions can be obtained. In this case, we have explicitly specified the aspects of a change that we analyzed when defining the adaptation rules introduced above [5, 6]. Specifically, to represent a local change as a feature vector the following features have been considered:

1. **Modified element:** Type of the BPMN element that has been modified. SendEvent (Event) (0) or ReceiveEvent(Event) (1).
2. **Change action:** Delete (0), update (1), and create (2).
3. **Does the change introduce new events?** When applying a create or update action, a new event is triggered in the composition? No (0) or Yes (1).
4. **Has the event attached data?** Used to know if the modified event had data attached to it. No (0); the event propagated data that was previously introduced by another (1); the event introduced new data in the choreography (2).
5. **Type of update:** When applying an update action, either it is updated the name of the event (0), or it is updated the data of the event (1)?
6. **Is a throwing event affected?** Used to know if the change avoids an event being triggered. No (0) or Yes (1).
7. **Is a catch event affected?** Used to know if the change affects the reception of an event. No (0) or Yes (1).
8. **Is the previous event sent by the affected microservice?** Used to know if the element triggered before the modified one can be considered in some adaptation scenarios. No (0) or Yes (1).

As a representative example, the following vector represents the local change presented in Figure 2.

F1	F2	F3	F4	F5	F6	F7	F8
1	0	0	0	0	1	0	0

4.3 Training

Once the data set has been properly encoded, we need to train the selected techniques. To do so, we used 80% of the cases as the training set and the other 20% in the testing phase (introduced in Section 0).

The training set is used to train the classifier, which learns a rule set through the comparison of the feature vectors of the training set [11]. In addition, before using this classifier in the testing phase, it is worth analysing the performance of each classifier through cross-validation. Cross-validation is a statistical method of evaluating and comparing machine learning techniques by dividing data into two segments: one is used to train a classifier, and the other one is used to validate the classifier. This technique is useful when evaluating machine learning techniques with small dataset, like in the case presented in this work, and it is also useful for predicting the results of the machine learning techniques before starting the testing phase [12]. Moreover, to reduce statistical variability (i.e., dispersion), multiple rounds of cross-validation are performed using different partitions and the results are averaged over the rounds [13].

In this work, 10 rounds have been used for each classifier. Thus, the training dataset is partitioned into 10 subsets. Then, the classifier is trained using 9 subsets and validated with the one left apart. This process is repeated ten times, each time using 9 different subsets and a different subset for validation. The final train uses all 10 subsets and consequently, the performance that the cross-validation provides is the average accuracy and the kappa considering all 10 rounds. In this phase we evaluate two metrics: *accuracy* that is the percentage of correctly classified cases out of all cases, and *kappa* that is a metric that compares an observed accuracy with an expected accuracy (random chance), which is useful to evaluate not only a single machine learning technique, but also machine learning techniques amongst themselves.

We have done the cross-validation by using the R language and the RStudio environment[2]. This environment provides mechanisms to train machine learning techniques with multiple configurations, which are automatically selected by the own environment. The k-NN technique is trained and validated with different numbers of neighbours (k). On the one hand, k-NN classifies a case based on its few nearest cases or neighbours where the k parameter indicates how many neighbours must be considered. The neighbours are obtained based on the cases provided by the training dataset.

Decision trees consist of a collection of rules if-then. Decision trees divide the input cases creating rules, until all cases are separated or until the decision tree reaches a maximum depth. The input cases are subdivided between two sub-trees (the one that fulfils the rule and the one that does not) and then, each sub-tree can be subdivided into two more. There exists a myriad of techniques based on decision trees. For the purposes of this work, we evaluated the following two: RPART and CART. The RPART technique is a variation of the classical tree presented in [14] and can provide a visualization of the decision tree created with a representation of the data, requiring a low computational demand [15, 16]. The CART technique is similar to the RPART technique, since it also requires a low computational demand and provide a visualization of the decision tree. The CART technique produces only binary trees, and it based on a numerical splitting criterion recursively applied to the data [17].

The RPART technique is trained and validated with different complexity parameters (cp). The cp is used to control the size of the decision tree, only splitting the node's tree

[2] https://www.rstudio.com/

if the relative error is improved. The CART technique is trained and validated with a different max depth parameter that controls the maximum depth of the produced trees.

The results of the cross-validation are shown in Figure 3, Figure 4 and Figure 5. The best precision and kappa for the CART technique is obtained by using a value of 4 as the max depth of the tree, with a 70% of precision and a 59% of kappa. The best precision and kappa for the RPART technique is obtained by using a cp of 0.05, with a 61% of precision and 45% of kappa. The best precision and kappa for the k-NN is obtained by using a k of 7, with an 74% of precision and an 64% of kappa. Comparing the three options, we can see that the k-NN technique with a k of 7 is the best one in the phase of training.

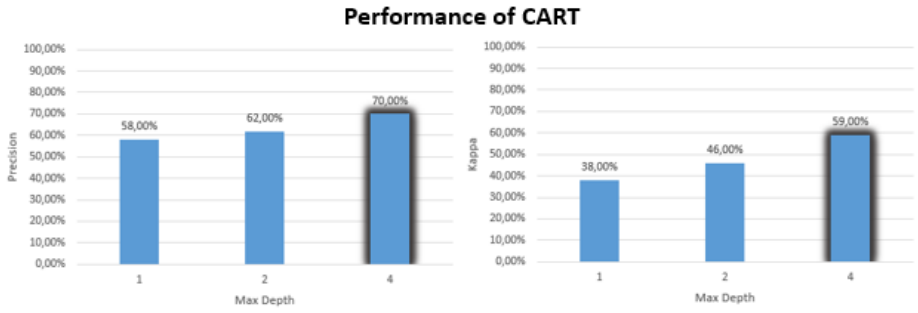

Figure 3. Performance of the CART technique.

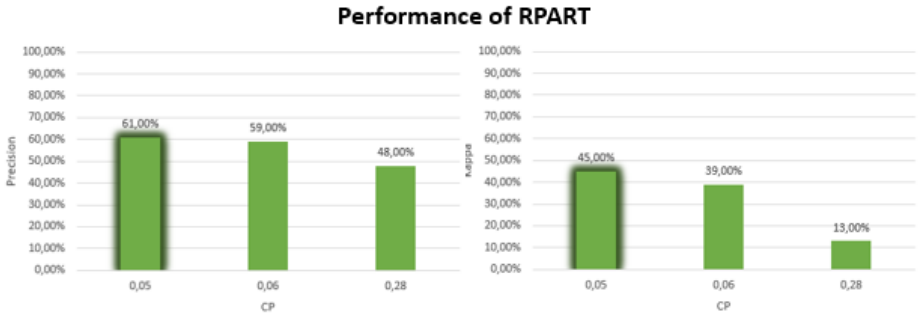

Figure 4. Performance of the RPART technique.

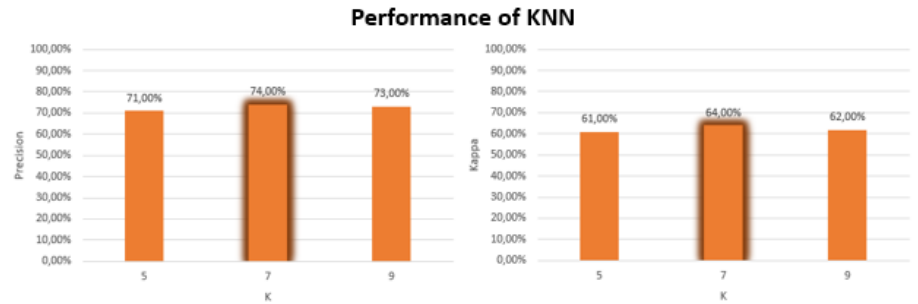

Figure 5. Performance of the KNN technique.

4.4 Testing

In this subsection we present the testing done to the algorithms by using the cases left for this purpose (20% of cases of the dataset). We analyse the performance of the classifiers using three measures: recall, precision, and the F-measure. Recall measures the proportion of elements of the correct solution that are correctly retrieved by the obtained solution. Precision measures the proportion of elements from the obtained solution that are correct according to the correct solution. Each rule that can be predicted by the algorithm has its own value of recall and precision. The F-measure corresponds to the harmonic mean of the precision and recall. The average F-measure is obtained by calculating the mean of the F-measure of each rule that can be predicted.

All three measures can range between 0% and 100% where the higher is the value, the better is the classifier. A value of 100% precision and 100% recall implies that both the predicted rule and the correct rule are the same in all the cases. Consequently, the F-measure will have a value of 100% as well. The results obtained in the testing phase are shown in Figure 6. The technique that obtains the best results is k-NN, with an overall precision of 78% and an average F-measure of 57%. With respect to the recall and the precision of each rule that can be selected by the algorithm, we have also obtained the average precision and recall of each rule, calculating the mean. In the bottom side of Figure 6 the two graphics present the average recall and precision of the different rules that can be predicted by the algorithm and that are defined in the catalogue of adaptation rules. The k-NN technique is again the one with the best performance in both measures, with 48% of average recall and 72% of average precision. These results reinforce the ones obtained in the training phase and allows us to conclude that the best classifier for our purpose is k-NN.

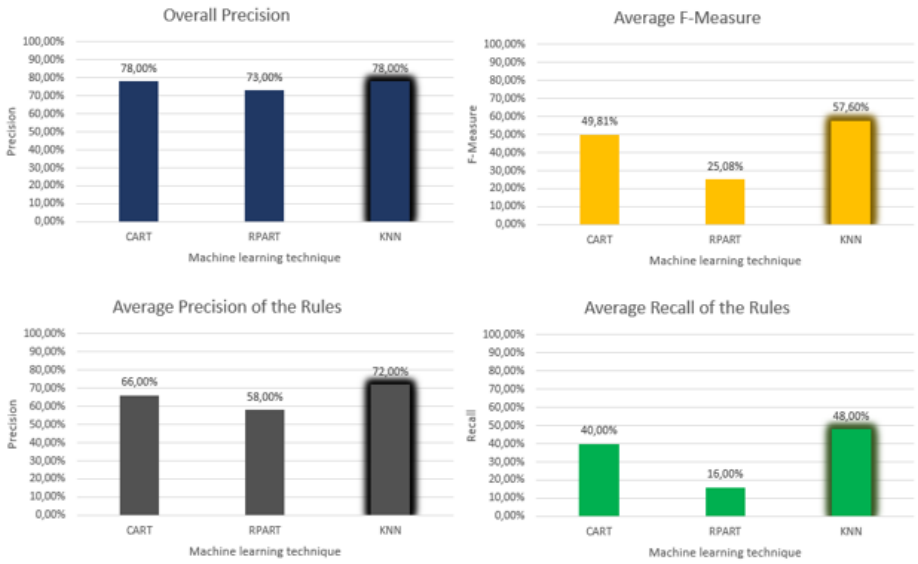

Figure 6. Test results.

5 Conclusions and further work

In this paper, we have presented a solution based on machine learning techniques to manage local modifications within a microservice composition based on the choreography of BPMN fragments. Taking as a basis a catalogue of adaptation rules, we have trained a machine learning classifier that can automatically identify the rule to be applied when a microservice performs a local change.

To train the machine learning classifier we have used a dataset that contains 272 cases of local changes obtained from the implementation of several case studies. We have analysed three classifiers based on the k-NN technique and decision trees. The classifier with the best results has been the k-NN technique.

As further work, we want to extend the dataset with more cases to train a classifier based on neural networks. We also plan to study the application of online machine learning techniques in such a way the classifier can be automatically retrained from the different results obtained progressively.

References

1. Weske, M. Business Process Management: Concepts, Languages, Architectures. Springer, 2007
2. J. Lewis and M. Fowler. Microservices. 2014. https://martinfowler.com/articles/microservices.html (accessed April 2022).
3. [Anonymized] XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
4. S. Rinderle, M. Reichert, and P. Dadam, Correctness criteria for dynamic changes in workflow systems - A survey. In Data and Knowledge Engineering, 2004, 50 (1), pp. 9–34.
5. [Anonymized] XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
6. [Anonymized] XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
7. M. Mohammed, M. B. Khan, and E. B. M. Bashie. Machine learning: Algorithms and applications. Machine Learning: Algorithms and Applications, pp. 1–204, Aug. 2016
8. [Anonymized] XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
9. Runeson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. Empir. Softw. Eng.14(2), 131–164 (2009). https://doi.org/10.1007/s10664-008-9102-8
10. Mulyar, N., van der Aalst, W.M., Russell, N.: Process flexibility patterns. Technische Universiteit Eindhoven (2008)
11. Shabtai, A., Moskovitch, R., Elovici, Y., and Glezer, C.: Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey. Information Security Technical Report, vol. 14, no. 1, pp. 16–29, 2009
12. Refaeilzadeh, P., Tang, L., and Liu, H.: Cross-Validation. In Encyclopedia of Database Systems, Springer US, 2009, pp. 532–538
13. Song, Q., Jia, Z., Shepperd, M., Ying, S., and Liu, J.: A general software defect-proneness prediction framework. IEEE Transactions on Software Engineering, 37(3), pp. 356–370, 2011.
14. Gajowniczek, K., & Ząbkowski, T. (2021). ImbTreeEntropy and ImbTreeAUC: Novel R packages for decision tree learning on the imbalanced datasets. Electronics, 10(6), 657.

15. Fernández-García, A. J., Iribarne, L., Corral, A., Criado, J., and Wang, J. Z.: A recommender system for component-based applications using machine learning techniques. Knowledge-Based Systems, vol. 164, pp. 68–84, 2019.
16. Le, T. T., & Moore, J. H. (2021). treeheatr: an R package for interpretable decision tree visualizations. Bioinformatics, 37(2), 282-284.
17. Rutkowski, L., Jaworski, M., Pietruczuk, L., & Duda, P. (2014). The CART decision tree for mining data streams. Information Sciences, 266, 1-15.