Predicting Physical Disturbances in Organic Computing Systems Using Automated Machine Learning

Markus Görlich-Bucher(✉), Michael Heider, and Jörg Hähner

Organic Computing Group, University of Augsburg, Augsburg, Germany
{markus.goerlich-bucher,michael.heider,joerg.haehner}@uni-a.de
https://www.uni-augsburg.de/en/fakultaet/fai/informatik/prof/oc/

Abstract. Robustness against internal or external disturbances is a key competence of Organic Computing Systems. Hereby, a rarely discussed aspect are physical disturbances, therefore, failures or breakdowns that affect a systems physical components. Before experiencing such a disturbance, physical components may show various measurable signs of deterioration that might be assessed through sensor data. If interpreted correctly, it would be possible to predict future physical disturbances and act appropriately in order to prevent them from possibly harming the overall system. As the actual structure of such data as well as the behaviour that disturbances produce might not be known a priori, it is of interest to equip Organic Computing Systems with the ability to learn to predict them autonomously. We utilize the Automated Machine Learning Framework TPOT for an online-learning-inspired methodology for learning to predict physical disturbances in an iterative manner. We evaluate our approach using a freely available dataset from the broader domain of Predictive Maintenance research and show that our approach is able to build predictors with reasonable prediction quality autonomously.

Keywords: Organic Computing · Automated Machine Learning · Predictive Maintenance

1 Introduction

Organic Computing (OC) [9] is intended to solve the increasing complexity in information- and communication technology by allowing systems to freely adapt and organize themselves. OC-based systems are expected to involve various kinds of *sensors* and *actuators* and are explicitly designed to cope with plenty different types of real-world scenarios and use-cases. A notable focus in OC research over the last years lies on investigating how OC systems can be built to be *robust*, therefore, to remain functioning within a desired range of performance even though various kinds of internal or external *disturbances* may appear [14] Hereby, most of the existing research focuses on software-sided disturbances.

Only few works on how to deal with hardware-sided disturbances, termed *physical disturbances* throughout this work, exist, although being a serious, yet unresolved problem in OC, as outlined in [6]: A damaged actuator, for example, remains damaged until it is repaired or exchanged and may affect the overall performance of the entire system. Although human repair works may be necessary to replace broken hardware, it may indeed be possible and useful to reduce the amount of human participation to an absolute minimum. In order to do so, it is necessary to be able to *predict* when future physical disturbances will happen. This could allow the OC system to estimate how long it will be able to function in a desired way, therefore, to assess how long it will be robust. In order to be able to predict upcoming physical disturbances, it is both necessary to collect a suitable amount of training data as well as to choose a suitable *Machine Learning* (ML) algorithm depending on the overall structure and type of the collected data. As OC systems are intended to move design-time decisions to runtime, it is not possible to choose an appropriate algorithm a priori. We suggest the utilization of an *Automated Machine Learning* (AutoML) framework for overcoming this issue. AutoML-approaches are intended to automatically choose and parametrize an appropriate ML algorithm based on the given input data, as well as to incorporate necessary data preprocessing steps. In this work, we present an AutoML-based approach for predicting upcoming physical disturbances using the *Tree-based Pipeline Optimization Toolkit* (TPOT) [10]. We present an iterative process that gathers measurements from the OC systems' hardware components and utilizes TPOT to continuously learn and optimize until a desired prediction quality is reached.

The remainder of this paper is structured as follows. In Sect. 2, we give a brief overview on existing related research from the field of OC as well as the related field of *Predictive Maintenance*. Furthermore, we refer to various existing AutoML frameworks. Afterwards, we provide a more detailed motivation of the underlying problem of this work and provide a brief introduction to TPOT in Sect. 3. We thoroughly explain our approach in Sect. 4 before evaluating it using a simple smart factory scenario in Sect. 5. We conclude with a short outlook on possible future work in Sect. 6.

2 Related Work

There are several aspects in OC research that are relevant to our work. First of all, the concept of robustness, as already mentioned in the introduction, is related to the occurrence of disturbances. A contemporary approach on measuring robustness can be found in [14]. The latter work also gives a good introduction on this topic in general. Quite similar is the self-x property *self-healing*, therefore, the ability of an OC system to resolve disturbances by taking appropriate countermeasures. However, existing research on self-healing mostly focuses on healing software-sided disturbances, e.g. in [12]. Finally, various ML approaches have been utilized in OC research so far. Hereby, the XCS classifier system (XCS) [3] as well as some of its derivates have gained plenty of attention in the

OC community. However, they are mostly utilized to learn appropriate control strategies based on the current situation in the underlying System. We refer to [13] for a broader introduction to the usage of XCS in OC, as well as for an overview of other ML techniques used in OC so far.

Predictive Maintenance (PdM), sometimes also termed as *Condition-based Maintenance* is a quite active area of research in various other scientific disciplines. Especially over course of the last years, ML techniques have gained an important role in PdM research. We refer to [4] for a more detailed introduction and broader overview on current research in this topic. A notable difference between our proposed approach and PdM concepts lies in the inherent design-time to runtime idea of OC: ML methodologies are usually developed with a notable amount of domain knowledge, their applicability can be tested and evaluated thoroughly. Contrary, utilizing ML for predicting hardware failures in OC necessitates methodologies that do not rely on such optimal conditions.

AutoML has become a quite active domain of research over the course of the last years, too. Various state-of-the-art frameworks make use of *Bayesian Optimization* in order to optimize both chosen algorithms and preprocessing steps as well as their hyperparameters. A contemporary survey on various AutoML frameworks is given in [17].

3 Prerequisites

The overall system model can be briefly described as follows: We assume a *System under Observation and Control* (SuOC) controlled by a *Control Mechanism* (CM), for example an instance of the *Multi-Layer Observer Controller*-architecture [15]. The SuOC is associated with a set of various *components* C. Here, a component refers to possible *sensors and actuators* an OC system may be associated with in order to interact with its surrounding environment. However, the term *sensors and actuators* does not necessarily refer to sensors *or* actuators, but could also be taken as a description for more sophisticated components or machinery involving both of them, i.e. soft-sensing mechanisms.

We assume that the CM is able to assess individual health states for each single component $c \in C$ at each discrete timestep t. This means that the CM is able to decide wether a component is *functioning* or *defective* (therefore disturbed) at timestep t. This can happen, for example by a component shutting down after an error, meaning that the CM can no longer gather data from it, or by an utility metric measuring the system performance that suddenly decreases.

We assume that a component is able to return some sort of status information or *internal* sensor readings somehow reflecting the internal state of a component. The term internal sensor readings should not be confused with sensors an OC system is equipped with for interacting with its environment: The former refers to measuring, as explained, the internal state of a component, the latter refers to a designated component that is used to assess the environment. Under the assumption that the gathered measurements reflect the actual physical state of the component, this information appears useful to predict future

physical disturbances: If an upcoming physical disturbance is known a priori, the CM might be able to proactively take countermeasures to ensure a robust system state. The OC approach of moving design-time decisions to runtime can yield various problems at this point: It is not known in advance how the gathered data may look like. There exist scenarios where one has to expect quite simple, structured data (e.g. vibration and temperature sensor measurements in a simple mechanical machinery). On the other hand, there may exist scenarios where the gathered information is quite complex (e.g. images of produce taken by a camera installed for quality assurance purposes). Also, the data can be enriched with information actually useless for predicting physical disturbances. This can necessitate different types of machine learning algorithms as well as various possible preprocessing steps depending on structure and type of the incoming data. Finally, it is also necessary to determine suitable hyperparameters for the used algorithms - a non-trivial task requiring an appropriate amount of data for training and testing purposes. Accordingly, traditional OC-related learning paradigms such as XCS appear inappropriate for this kind of learning problem, as they are not necessarily applicable to e.g. unprocessed, high-dimensional or unstructured data. More precise, XCS is a single learning paradigm, whereas the described learning problem may necessitate multiple different suitable learning algorithms, depending on the actual data available within specific scenario. We therefore focus on methods from the broader field of AutoML in order to tackle previously motivated problem. Within the scope of this work, we use the TPOT framework as an AutoML framework, as it is based on *Genetic Algorithms*—a class of optimization heuristics that are commonly used in OC.

The overall idea of TPOT is to utilize *Genetic Programming* (GP) [1] for building and optimizing ML pipelines. TPOT is able to use various preprocessing and decomposition algorithms, feature selectors as well as actual ML models as *operators*. The operators are combined in tree-based structures. Both the structure of these trees as well as the parameters of the chosen operators are evolved by means of GP: In the very beginning, a population of such tree-based individuals is generated randomly. Each individual is trained and tested using an appropriate train/test-split of the given training data. Using a suitable selection scheme, some of the individuals are chosen for breeding or applying genetic operators, such as crossover, for the next generation of the population. From generation to generation, TPOT is able to iteratively generate and optimize ML pipelines for the given data. As a broader introduction would be beyond the scope of this work, we refer to [10] for a more detailed explanation of the individual parts of TPOT, as well as for an evaluation of the approach on several datasets from the UCI machine learning repository.

4 Methodology

Our overall approach is divided in three phases: Right after setting up the OC system, the *BOOTSTRAP*-phase takes place. It is used to collect an initial amount of training data that is (presumably) sufficient enough to train a first

pipeline. As this goal is reached, the system changes to the *OPTIMIZATION*-phase. In this phase, pipelines are trained in an iterative manner at certain discrete timesteps until the system reaches a desired prediction quality. Afterwards, the system switches to the *PRODUCTION*-phase, where the actual predictions of the trained pipeline should be used by the CM in order to cope with upcoming disturbances.

In order to simplify the following explanations, we assume that all components in C are of the same type (in order to allow the CM to use gathered data from all components for training one ML pipeline that is used to predict disturbances in all components). Of course, OC systems presumably feature various different kinds of components. Accordingly, in an actual real-world scenario, the CM would conduct the following process individually for each type of component existing in its SuOC.

4.1 Data Collection and Labeling

At each discrete timestep t, the CM gathers a row of sensor readings for each component $c_i \in C$ and saves them for later use. Additionally, the CM examines the current physical state of each component $c_i \in C$.

If the CM identifies a component c_i as broken, it recalls all sensor measurements gathered for c_i since the component's last breakdown (or since the installation of the overall system, if no breakdown for the corresponding component happened before), resulting in a set of chronologically ordered *rows* for c_i. Afterwards, all rows that are younger than a certain threshold Θ_{pred} are *labeled* as *positive*, all other rows are labeled as *negative*. Θ_{pred} can be described as the desired *prediction horizon* for predicting upcoming breakdowns: If the trained learner is able to make perfect predictions, an upcoming disturbance would be predicted Θ_{pred} timesteps in advance. The choice of a suitable Θ_{pred} is not a trivial task and may depend on various aspects that might not be known a priori. For now, we assume that Θ_{pred} is given in advance (e.g. during setting up the corresponding system). After the data is collected and labeled, c_i is reported to the CM, e.g. for repair or maintenance operations. This overall functionality is refered to as LABELANDREPAIR() in the algorithms later on.

At the end of a timestep, all collected sets of data rows are merged. Afterwards, a *train/test-split* is applied to the merged data: The major amount of collected data is added to the *training dataset* $\mathcal{D}_{train}$. The minor amount is added to the *test holdout dataset* $\mathcal{D}_{test}$. The split is done proportional to the positive/negative labels (that is, the proportion of positives and negatives among the data added to the training set is the same as for those added to the test holdout set). The intuition behind these sets is as follows: $\mathcal{D}_{train}$ is used as data for training actual AutoML pipelines. $\mathcal{D}_{test}$, on the other hand, is used as previously unseen data to *validate* the results of the trained pipelines after they are optimized. Both sets are extended with novel training (respectively testing) data each time a component breaks. We refer to this functionality as CONSOLIDATEDATA() in the algorithms later on.

4.2 Bootstrap Phase

The goal of the Bootstrap Phase is to gather enough data for training a first pipeline for predicting future disturbances. Here, the term *enough data* should be taken as a rough estimate rather than an explicit boundary: Deciding if enough data for training a machine learning algorithm exists is not a trivial task and of subordinate importance for our overall approach. The idea of the bootstrap phase is just to avoid too early trainings that can lead to irritating results: For example, we found that training a pipeline with very few examples can lead to unrealistic good results during the first training episodes in our evaluation. The trained pipeline apparently overfits on the training data. As both the training data as well as the test data holdout are sampled from very few components in the beginning, the test holdout dataset did not indicate an overfitting—leading to very good results which later on declined iteratively, as more and more data is added to both datasets.

Accordingly, nothing except the previously explained data collection and labeling takes place during the bootstrap phase. Whenever a component fails, its collected measurements are labeled and, at the end of a timestep, split and added to $\mathcal{D}_{train}$ and $\mathcal{D}_{test}$ respectively. We furthermore assume that the component is repaired afterwards, allowing the CM to collect data from this component again. At the end of each timestep, it is assessed if enough training data was collected as shown in Algorithm 1.

Algorithm 1. isBootstrapDone()

```
1: function isBootstrapDone()
2:     if |D_train+| * Θ_sr / Fib(fp) >= d and |D_train−| * Θ_sr / Fib(fp) >= d then
3:         return true
4:     else
5:         return false
```

Here, $|\mathcal{D}_{train+}|$ and $|\mathcal{D}_{train-}|$ refer to the number of positive (or negative) training data rows available. Θ_{sr} determines the *sample ratio* and is a fixed multiplier set in advance. Fib(i) is a function returning the ith element from a list of Fibonacci numbers. The purpose of the variable fp (Fibonacci pointer) and the corresponding list of Fibonacci numbers is explained in the next subsection. Finally, d refers to the dimensionality of the used input data. The idea of the isBootstrapDone() is as follows: It is checked if both the amount of positive labeled training data as well as negative labeled training data lies above a certain threshold. This threshold is calculated using a simple heuristic based on a fixed, predefined sample ratio, a variable Fibonacci number as well as dimensionality of the used data. The Bootstrap phase ends once these conditions hold by executing the first training and starting the optimization phase.

4.3 Optimization Phase

The optimization phase is meant to iteratively train ML pipelines using TPOT and continuously assess their performance, until the latter reaches an acceptable level for beeing used productively. The corresponding algorithm is shown in Algorithm 2.

Algorithm 2. Optimization phase

```
1: function OPTIMIZATION()
2:     for c in C do
3:         if isBroken(c) then
4:             labelAndRepair(c)
5:             evaluateMachine(c)
6:     consolidateData()
7:     if nextTrainingNecessary() then
8:         training()
9:     if isOptimizationDone() then
10:        production()
```

The overall procedure is as follows: Upon entering the optimization phase, the method TRAIN() is called in order to train a first usable pipeline, as shown in Algorithm 2 in the previous subsection. It should be denoted that the training happens *during* the current discrete timestep, accordingly, ending the Bootstrap phase, entering the Optimization phase and training the first pipeline happens at the end of the same timestep. Afterwards, the system still continues to gather labeled data similar to the Bootstrap phase. The method EVALUATEMACHINE() is used to evaluate the performance of the current trained TPOT instance on the (labeled) data of the newly broken component, therefore, to assess how good the performance of the instance would have been in an (actual) productive scenario. The calculated score is saved internally. Furthermore, if enough data was gathered, the method calculates the moving average for the last $\Theta_{OptWindow}$ scores that have been recorded. The moving average is appended to the list `movingAverageScores`, which then acts as a rolling window to assess how the overall performance changes over time. At the end of each timestep, measurements from newly broken components are added to the corresponding data sets. Besides, it is checked if another retraining is necessary using NEXTTRAININGNECESSARY(). If this is the case, TRAINING() is called again. Finally, ISOPTIMIZATIONDONE() is called in order to assess if the Optimization phase can come to an end. The individual functions are explained in the following.

Algorithm 3 shows the TRAIN()-method. First of all, in line 2, the existing training data in $\mathcal{D}_{train}$ is divided into k *stratified folds* or *splits* used internally in TPOT for training and testing. Actually, TPOT would be able to split the training data for cross-validation (CV) purposes internally. The reason why this is done in advance follows in line 3 and line 4: Due to the high class imbalance,

Algorithm 3. TRAIN()

```
1: function TRAIN()
2:     splits := StratifiedKFold(D_train)
3:     for trainSplit in splits do
4:         SMOTEENN(trainSplit)
5:     newLearner := fitTpot(splits)
6:     if currentLearner is not null then
7:         newPrediction := newLearner.predict(D_test)
8:         currentPrediction := currentLearner.predict(D_test)
9:         if score(newPrediction) > score(currentPrediction) then
10:            currentLearner := newLearner
11:            fp := fp - 1
12:        else
13:            fp := fp + 1
14:    else
15:        currentLearner := newLearner
16:        fp := fp - 1
```

TPOT tends to overfit on the majority class, as we found out in some preliminary test runs. In order to cope with this issue, we integrated suitable over- and undersampling methods. We used the SMOTEENN-algorithm, a combination of SMOTE and Edited Nearest Neigbours (ENN) [2]. Applying SMOTEENN on the whole training data and letting TPOT create the CV-folds by itself would lead to synthetical data within the (internal) test splits. The internal test splits need to be as imbalanced as the (expected) data the pipeline is confronted with afterwards in order to avoid biasing the chosen scoring function. Accordingly, only those splits that are used as training data afterwards are altered. After the data augmentation is done, a new TPOT instance is created and fitted using the splits in Line 5. The pipeline created by the new TPOT instance is now used to create predictions for the existing test holdout set $\mathcal{D}_{test}$. Additionally, the *last* TPOT instance (that is: the TPOT instance that was created in the previous training run) is also used to create predictions for $\mathcal{D}_{test}$. Afterwards, a scoring function is applied to both predictions in order to determine if the last instance or the newly trained one performs better on the test holdout dataset. If the newly trained instance performs better, it is saved and the previous instance is discarded. Additionally, the Fibonacci pointer `fp` is decremented. If the older instance performs better than the newly trained one, the latter is discarded and `fp` is incremented. This also happens when no previously trained instance exists.

The idea of incrementing/decrementing the Fibonacci pointer followed our first preliminary experiments, where we found that rather short training intervals do not show any notable change in performance. Moreover, it could happen that a newly trained pipeline performs worse than the previous one. The idea of the Fibonacci pointer is to introduce some sort of adaptive threshold for deciding if a retraining is necessary: Increasing the pointer leads to a longer interval until the next retraining, reflecting the case that no increased performance was reached

in the current training. Decreasing it leads to a shorter interval, as obviously the size of the previous interval was sufficient to increase the prediction performance. In general, the overall method would also work with fixed intervals, as a new training does not depend on the previous training. However, as the trainings are quite time expensive, it is of interest to reduce the overall number of trainings. Fibonacci numbers as multiplicator for the intervals show favourable characteristics. Their slope is larger compared to linear functions, which avoids smaller training intervals and thus expensive training cycles. Furthermore, its slope is smaller than a quadratic functions which would grow too quick. The influence of the Fibonacci pointer can be seen in the nextTrainingNecessary() method, shown in Algorithm 4.

Algorithm 4. nextTrainingNecessary()

```
1: function nextTrainingNecessary()
2:     if (|D_train+| − |D^-1_train+|) · Θ_sr/Fib(fp) >= d and (|D_train−| − |D^-1_train−|) ·
   Θ_sr/Fib(fp) >= d then
3:         return true
4:     else
5:         return false
```

The algorithm is quite similar to Algorithm 1 that is used to decide if the bootstrap phase shall end. The only difference here is the usage of $|\mathcal{D}^{-1}_{training+}|$ and $|\mathcal{D}^{-1}_{training-}|$, referring to the amount of positive (and negative) training samples that were available during the *last* training. Therefore, it is investigated if the amount of training data gathered since the last training exceeds a certain threshold, again determined by the fixed sample ratio Θ_{sr} and the a Fibonacci number gathered through the previously explained Fibonacci pointer.

Finally, Algorithm 5 shows the algorithm that is used to decide if the optimization phase has come to an end.

Algorithm 5. isOptimizationDone

```
1: function isOptimizationDone()
2:     for i=1 to Θ_OptWindow do
3:         if movingAverageScores[-i] < Θ_OptThreshold then
4:             return false
5:     return true
```

The method uses the moving average score list that is extended each time a physical disturbance occurs, as previously explained. If the last $\Theta_{OptWindow}$ scores lie above a predefined threshold $\Theta_{OptThreshold}$, the optimization phase ends, as TPOT is now able to provide predictions that are considered good enough for the CM to be used for assessing the SuOC's future state. At the end of the corresponding timestep, the production phase starts.

4.4 Production Phase

The overall goal of the production phase is to use TPOT to predict upcoming physical disturbances, report those to the CM and to continuously assess if TPOTs performance is still acceptable. The procedure executed after each timestep is shown in Algorithm 6.

Algorithm 6. Production Phase

```
1: function PRODUCTION()
2:     for c in C do
3:         if isBroken(c) then
4:             labelAndRepair(c)
5:             evaluateMachine(c)
6:         else if c.mode == EXPLOIT then
7:             prediction := currentTpot.predict(c)
8:             if prediction := true then
9:                 reportPrediction(c)
10:    consolidateData()
11:    if nextTrainingNecessary() then
12:        training()
13:        resampleExploreExploit()
```

The algorithm is quite similar to the optimization phase. The major difference is that each component in C is associated with a *mode* that can be either *EXPLOIT* or *EXPLORE*. If a component is set to *EXPLOIT*, TPOT is used to predict whether the component c is expected to break or not, as can be seen in Line 6. If so, the CM is informed by calling REPORTPREDICTION(). It is now up to the CM to decide on proper countermeasures, e.g. by repairing or changing c in advance. Components set to *EXPLORE* are not considered in the predictions: Their purpose is to run until being hit by a physical disturbance. This makes it possible to continuously assess if the TPOT instance still makes acceptable predictions. The same obviously holds for components set to *EXPLOIT* that break without the disturbance being predicted properly. For now, NEXTTRAININGNECESSARY() and TRAINING() in Line 11 and 12 follow the functionality they have in the optimization phase: Data is gathered, another training run is started, and if the prediction performance of the current TPOT instance declines, it is replaced by a newly trained one. However, it is also conceivable to, for example, return to the optimization phase if the performance drops below a certain threshold. Another possible option would be to adapt the ratio between *EXPLOIT* and *EXPLORE* towards more machines set to *EXPLORE* in order to speeden up the learning process, if necessary. Finally, the method RESAMPLEEXPLOREEXPLOIT() is called to resample the distribution of *EXPLORE*, respectively *EXPLOIT*-modes among all components in C. The ratio between the two modes is set in advance. A rather small number of *EXPLORE* components appears suitable in order to allow the overall system to benefit from the predictions made for the *EXPLOIT* components.

5 Evaluation

The overall evaluation scenario is structured as follows. The SuOC consists of 5 components in form of identical *production machines*. In general, any other number of components would be applicable too - with lesser components, the overall process would take longer, with more components, is would be faster. We assume that the CM is able to assess if a machine is working properly or is broken. In the given scenario, this could be the case by assessing the number of workpieces a machine produces in a discrete timestep. Furthermore, each machine delivers various internal measurements that can be used to assess its internal state. Additionally, no configuration changes or other disturbances except actual machine breakdowns will take place, resulting in a quite simple evaluation scenario. The Azure AI Predictive Maintenance Dataset[1] was used for simulating the measurements from the individual machines. The dataset consists of 4 different machine types in total, type number 1 was used throughout this evaluation. The CSV-files in the dataset were preprocessed such that a single CSV-file exists for each machine in the dataset (containing measurements from installation until breakdown in chronologial order). Incomplete traces (that is, machines without breakdowns and machines that were repaired although no breakdown happened) were removed. Furthermore, the *error* column was removed, therefore, the only measurements available are *volt*, *rotate*, *pressure* and *vibration*. This results in a total of 672 machines. The 5 components of the SuOC are equipped with 5 uniformly chosen machine CSV-files. A component is regarded as broken when the corresponding CSV-file reaches its last row. The component is then repaired in the next timestep by replacing the CSV-file by a newly sampled one.

5.1 Implementation and Parametrisation

We implemented our approach using Python as well as the original TPOT implementation [10]. Scikit-learn [11] was used for the stratified cross validation as well as for the scoring functions. Furthermore, the SMOTEENN-implementation from the *imblearn* library [8] was used. If not stated otherwise, the default parametrizations for the algorithms were used.

We used the *balanced accuracy* as a scoring function both internally in TPOT as well as afterwards for scoring in our own implementation due to the imbalanced nature of the available data: As the balanced accuracy is defined as the mean between sensitivity and specificity, it is less vulnerable to false predictions of the minority class, at least compared to the non-balanced accuracy metric. TPOT was configured with 10 generations, a population size of 25 and an early stopping of 2. This results in TPOT stopping the optimization process if no progress was made after 2 generations. At this point, it should be mentioned that TPOT features a *warmstart*-function. This allows TPOT to use the last population as a start population for a new run, however, we found that this

[1] https://www.kaggle.com/datasets/arnabbiswas1/microsoft-azure-predictive-maintenance.

leads to a notably worse performance compared to starting with a complete new population each time, therefore, we did not use it.

The fixed sample ratio Θ_{sr} was set to 0.025. The optimization threshold $\Theta_{optThreshold}$ as well as the optimization sliding window size $\Theta_{optWindow}$ were set to 0.9 and 20, respectively. The Fibonacci pointer fp was set to 5 for the bootstrap phase, resulting in the number 8. Additionally, we limited the fp to 6 to make sure the variable amount of the retraining interval does not get too large. The split between $\mathcal{D}_{train}$ and $\mathcal{D}_{test}$ was set to 0.9/0.1.

We evaluated 20 repetitions with different, fixed random seeds. Each repetition was limited to 80000 timesteps. The production phase was limited to 5000 timesteps, therefore, once the algorithm reaches production, it continues to run for another 5000 timesteps.

As a simple baseline, we used a naive Random Forest Classifier (RF) with sklearn-default parameters. The rest of the algorithm remains the same, therefore instead of TPOT, RF instances were trained.

5.2 Results

In order to give an idea how the overall procedure behaves over time, a single repetition is depicted in Fig. 1. The graph shows the behaviour of TPOT over the course of the execution. *Single Component* refers to the scoring that is done after a component broke, smoothed over 50 steps. *Current TPOT* and *Newly Trained TPOT* show the scores for the predictions on $\mathcal{D}_{test}$ in Line 7 and Line 8 from Algorithm 3. Figure 2 shows a similar experiment using RF classifiers.

We evaluated the balanced accuracy for all components broke during the production phase. As explained earlier, the idea of the balanced accuracy is to compensate the class imbalance in the dataset: In a worst-case scenario, the simple, non-balanced accuracy score can show quite good results when the ML algorithm simply classifies all incoming samples as the majority class, as the (wrong predictions) for the minority class simply have a unsignificant influence on the result. If the production phase was not reached, the last 5000 timesteps of the optimization phase were used. A mean balanced accuracy of 0.865 ± 0.047 was reached by the TPOT-based approach, the RF runs achieved values of 0.816 ± 0.027. The optimization phase of the former takes an average of 44829 timesteps, while the latter takes an average of 67741 timesteps. As the data appears to be not normally distributed according to a performed Shapiro-Wilk-test, both evaluated measurements were tested on significance using a Wilcoxon-rank-sum-test. It showed that the TPOT-based approach was able to reach a significantly better performance than the RF (with a p-value of 0.00043). Furthermore, it took significantly less timesteps to reach the production phase (with a p-value smaller than 0.00001).

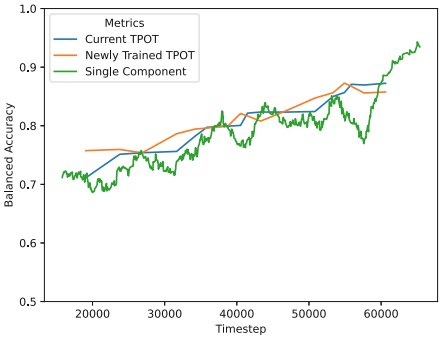

Fig. 1. A single TPOT-based run

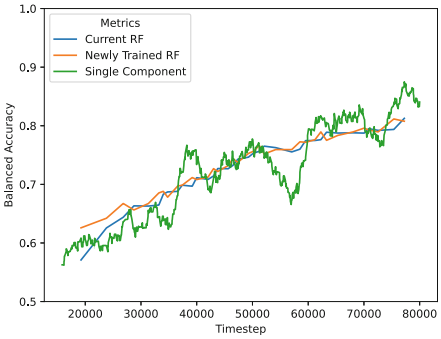

Fig. 2. A single RF-based run

5.3 Discussion

The results show that our approach is indeed able to achieve an acceptable prediction quality in a comparable amount of time. However, as the naive RF-classifier also reaches quite comparable results, one cannot necessarily state that the utilization of an AutoML framework intuitively leads to a superior performance. Rather, one can suspect that the chosen dataset is of quite simple structure, therefore, the RF-classifier is able to learn it without sophisticated preprocessing steps.

The Azure AI Predictive Maintenance Dataset is used in various other ML-related research work on PdM. In order to give an idea what performances are possible with manually optimized ML pipelines in general, a brief overview on suitable references is given in the following. Gęca [5] investigated several ML algorithms on the dataset. Various preprocessing (such as calculating statistical measures and including them in the data, as well as normalizing the whole dataset) was conducted. The labeling was done similar to our approach, a prediction window of 24 h was used. Most of the algorithm evaluated were able to reach an accuracy of over 0.99, with a Gradient boosting machine reaching 0.9993. Similar results were achieved by Hrnjica et. al. [7]. An accuracy of 0.948 was reached by their approach for the first machine type, also using 24 h as a prediction window and a gradient boosting manchine. However, a direct comparison of these results to our approach must be taken with caution: First of all, there is a quite obvious methodological difference between iterative online learning and an offline learning setting, when taking the amount of data a learner can use into account. Besides, both papers use a different prediction horizon. Furthermore, they do not use the balanced accuracy, which we do in order to compensate the class imbalance in the dataset. However, they provide solid precision and recall values, which at least allows the assumption that their approaches are able to deal with the class imbalance properly.

6 Conclusion and Outlook

In this paper, we presented a novel approach on learning to predict disturbances of physical nature in Organic Computing systems, or, to be more precise, in

the SuOC controlled by an OC-based CM. We motivated the advantages of AutoML-methodologies for such an use-case over ML-approaches used in existing OC-research so far. We introduced and explained our TPOT-based approach and evaluated it with a simple Smart Factory-simulation scenario. We showed that our approach is indeed able to learn to predict physical disturbances in an automated manner, with suitable prediction quality and better than a simple random forest-based approach. However, taking existing research work using the Azure PdM dataset into account, it can be suspected that a significantly better prediction quality could be possible–at least under the assumption, that a balanced accuracy score for both works shows a similar performance. A In order to investigate if the results from these papers are indeed as good as they appear, it is necessary to manually build and optimize a suitable ML pipeline using the balanced accuracy as a scorer. By doing so, it would be possible to assess how well the AutoML-based methodology performance compared to what is generally possible for the dataset.

As a next step, we plan to evaluate the overall concept using other AutoML-frameworks as well as Neural Architecture Search–based frameworks like AutoKeras. Furthermore, it is of interest to evaluate our approach using other datasets from the broader domain of PdM. Especially more complex dataset than the one used in the evaluation provided here will be necessary to investigate if the overall methodology can benefit from AutoML-based approaches. Hereby, a notable focus could lie on noise or wrong sensor readings. Besides, we plan to investigate how the parameters of our own methodology as well as the parametrisation of TPOT (or any other AutoML-framework we employ in the future) affects the overall prediction quality. Another interesting question is how the approach deals with an upcoming concept drift [16]: Intuitively, the continuous assessment in the production phase would lead to more frequent trainings if the existing pipelines increasingly fail on novel, drifted data. This would be a quite interesting aspect, as it can be expected that OC-related real world scenarios, due to their approach to adapt to changes, would involve some sort of drift. Another interesting aspect would be a further investigation of the actual pipelines generated by TPOT in terms analyzing if e.g. common patterns or algorithms are evolved. Finally, the evaluation in this work relates to a quite optimistic real-world setting: The (internal) sensor data from components in a SuOC might not always be reliable. Accordingly, additional methodologies for validating the information gathered for the AutoML-algorithms might be necessary. This would include mechanisms to actually identify already disturbed components in a reliable manner, in order to provide a correct labeling for the learning process.

References

1. Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D.: Genetic Programming: An Introduction: on the Automatic Evolution of Computer Programs and its Applications. Morgan Kaufmann Publishers Inc., Burlington (1998)

2. Batista, G.E., Prati, R.C., Monard, M.C.: A study of the behavior of several methods for balancing machine learning training data. ACM SIGKDD Explor. Newsl **6**(1), 20–29 (2004)
3. Butz, M.V., Wilson, S.W.: An algorithmic description of XCS. Soft. Comput. **6**(3), 144–153 (2002)
4. Carvalho, T.P., Soares, F.A., Vita, R., Francisco, R.D.P., Basto, J.P., Alcalá, S.G.: A systematic literature review of machine learning methods applied to predictive maintenance. Comput. Ind. Eng. **137**, 106024 (2019)
5. Gęca, J.: Performance comparison of machine learning algotihms for predictive maintenance. Informatyka, Automatyka, Pomiary w Gospodarce i Ochronie Środowiska 10 (2020)
6. Görlich-Bucher, M.: Dealing with hardware-related disturbances in organic computing systems. In: INFORMATIK 2019. Gesellschaft für Informatik eV (2019)
7. Hrnjica, B., Softic, S.: Explainable AI in manufacturing: a predictive maintenance case study. In: Lalic, B., Majstorovic, V., Marjanovic, U., von Cieminski, G., Romero, D. (eds.) APMS 2020. IAICT, vol. 592, pp. 66–73. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-57997-5_8
8. Lemaître, G., Nogueira, F., Aridas, C.K.: Imbalanced-learn: a python toolbox to tackle the curse of imbalanced datasets in machine learning. J. Mach. Learn. Res. **18**(17), 1–5 (2017). http://jmlr.org/papers/v18/16-365
9. Müller-Schloer, C., Tomforde, S.: Organic Computing-Technical Systems for Survival in the Real World. Springer, Cham (2017)
10. Olson, R.S., Bartley, N., Urbanowicz, R.J., Moore, J.H.: Evaluation of a tree-based pipeline optimization tool for automating data science. In: Proceedings of the Genetic and Evolutionary Computation Conference 2016, pp. 485–492. GECCO 2016, ACM, New York, NY, USA (2016)
11. Pedregosa, F., et al.: Scikit-learn: machine learning in Python. J. Mach. Learn. Res. **12**, 2825–2830 (2011)
12. Schmitt, J., Roth, M., Kiefhaber, R., Kluge, F., Ungerer, T.: Using an automated planner to control an organic middleware. In: 2011 IEEE Fifth International Conference on Self-Adaptive and Self-Organizing Systems, pp. 71–78. IEEE (2011)
13. Stein, A.: Reaction learning. In: Organic Computing - Technical Systems for Survival in the Real World, pp. 287–328. Springer (2017)
14. Tomforde, S., Kantert, J., Müller-Schloer, C., Bödelt, S., Sick, B.: Comparing the effects of disturbances in self-adaptive systems - a generalised approach for the quantification of robustness. In: Nguyen, N.T., Kowalczyk, R., van den Herik, J., Rocha, A.P., Filipe, J. (eds.) Transactions on Computational Collective Intelligence XXVIII. LNCS, vol. 10780, pp. 193–220. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-78301-7_9
15. Tomforde, S., et al.: Observation and control of organic systems. In: Müller-Schloer, C., Schmeck, H., Ungerer, T. (eds.) Organic Computing-A Paradigm Shift for Complex Systems, vol. 1, pp. 325–338. Springer, Basel (2011). https://doi.org/10.1007/978-3-0348-0130-0_21
16. Wang, S., Schlobach, S., Klein, M.: What is concept drift and how to measure it? In: Cimiano, P., Pinto, H.S. (eds.) EKAW 2010. LNCS (LNAI), vol. 6317, pp. 241–256. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16438-5_17
17. Zöller, M.A., Huber, M.F.: Benchmark and survey of automated machine learning frameworks. J. Artif. Intell. Res. **70**, 409–472 (2021)