



Estimation of execution time for computing tasks

Jan Bielecki¹ · Michał Śmiałek¹

Received: 4 December 2021 / Revised: 7 July 2022 / Accepted: 3 October 2022 / Published online: 6 November 2022
© The Author(s) 2022

Abstract

This work aims to estimate the execution time of data processing tasks (specific executions of a program or an algorithm) before their execution. The paper focuses on the estimation of the average-case execution time (ACET). This metric can be used to predict the approximate cost of computations, e.g. when resource consumption in a High-Performance Computing system has to be known in advance. The presented approach proposes to create machine learning models using historical data. The models use program metadata (properties of input data) and parameters of the run-time environment as their explanatory variables. Moreover, the set of these variables can be easily expanded with additional parameters of the specific programs. The program code itself is treated as a black box. The response variable of the model is the execution time. The models have been validated within a Large-Scale Computing system that allows for a unified treatment of programs as computation modules. We present the process of training and validation for several different computation modules and discuss the suitability of the proposed models for ACET estimation in various computing environments.

Keywords Execution time · Algorithm performance · High-performance computing · Non-linear regression · Machine learning

1 Introduction

1.1 Context

The demand for computation resources is constantly growing. This is mostly due to the high computational requirements of various machine learning models and other tasks related to Big Data, such as the Internet of Things. The role of High-Performance Computing (HPC) systems is to perform an efficient execution of such highly resource-consuming computations. One of the key elements in the usage of HPC is the capability to determine the consumption of resources and the associated cost. An important issue in this context is the capability to predict this cost in advance. Such predictions can be used to

determine proper resource allocation, e.g. to institute a prepaid computation model.

Cost prediction can be facilitated if the computation algorithms are implemented in a modularized and standardized form. Such an approach is taken in the Baltic Large Scale Computing system (BalticLSC [1, 2], <https://www.balticlsc.eu>). It uses a visual language, called the Computation Application Language (CAL), to define data-driven computation applications. Each application consists of several Computation Modules (CM) that can be executed according to a graph of data flows. Every CM has to implement a standard API and is run as a standard container. Various CM instances can be executed in parallel on a distributed system consisting of potentially many computation clusters.

Figure 1 shows an example application that consists of a few CM calls to perform face recognition on the input video data. Input films are sent to the Video Splitter module that extracts individual frames (images) from the film. Then, each frame is sent to an instance of the Face Recogniser module. This module detects and marks faces on an input image. Then, all the individual frames are joined by the Image Merger module and sent back to the user as a film with marked faces.

✉ Jan Bielecki
jan.bielecki.dokt@pw.edu.pl

✉ Michał Śmiałek
michal.smialek@pw.edu.pl

¹ Faculty of Electrical Engineering, Warsaw University of Technology, ul. Koszykowa 75, 00-662 Warszawa, Poland

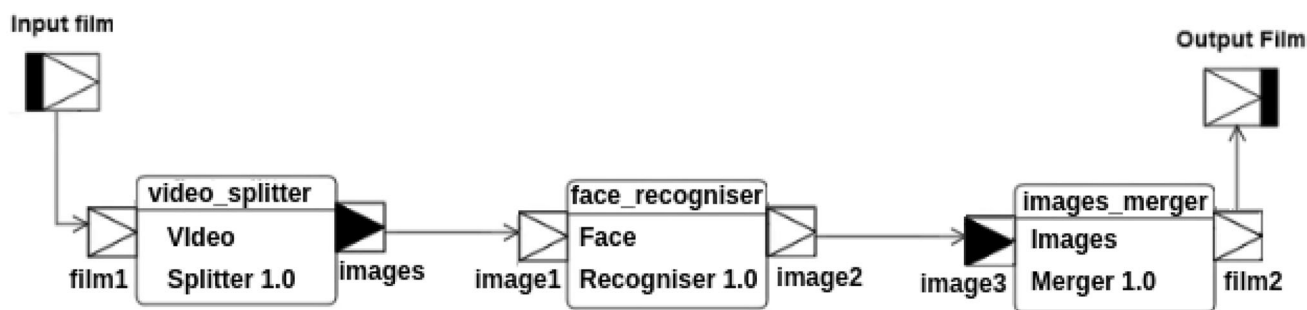


Fig. 1 The Face Recogniser application written in the CAL language

Note that this application has a significant capability of parallelization. The run-time engine can create many instances of the Face Recogniser to detect faces on many film frames in parallel. In this case, the usage of resources depends on how many frames have to be processed. In general, when an application is executed, CM instances are executed with some input data. This data can have different sizes and types, e.g. consisting of data frames or images. The execution of a specific module instance is handled by a virtual machine with specific resources (RAM, CPUs, GPUs). This, together with the properties of the input data, determines execution time and associated computation cost.

Standardized modules, like the Face Recogniser module, can be shared in many different applications. They can be optimized for working in a distributed environment with parallel processing capabilities and with different data sets. Application users would like to know in advance how long and how costly would it be to apply the particular module to their data. This leads us to the key problem of time and cost estimation for such Computation Modules.

1.2 Problem to be solved

Frequently, the worst-case execution time (WCET) metric is used for time estimations. This metric determines the greatest number of operations needed to solve the problem over input data of a given size. In this paper, we do not strictly focus on the WCET because we do not consider applications where strict maximum time estimations are necessary. Instead, we concentrate on the average-case execution time (ACET) metric which determines typical and not maximal times. For a specific CM, these times depend on certain execution parameters. In our models, we use the properties of the input data and the execution environment. Our goal is to be able to predict the execution time and price of a particular CM, given the particular input data set and the particular computational resources.

Our research can be situated in the field of algorithmic complexity analysis that has emerged as a scientific topic

during the 1960s and has been quickly established as one of the most active fields of study [3]. The most common way to describe the complexity of an algorithm is the *big O* notation (collectively called the Bachmann–Landau notation or the asymptotic notation) which is a universal formula that describes how the execution time of an algorithm grows as the input data size grows. This is usually related to the worst-case execution of a given algorithm. Here we do not need such strict analysis as we deal with average execution times.

Since execution time estimation is a long-known issue, various approaches have already been proposed. In the analytical method, a task is broken down into basic component operations or elements. If standard times are available from some data source, they are applied to these elements. Where no such times are available, they are estimated based on certain selected features of the elements. In the statistical approaches, execution time is estimated on the basis of past observations. In this work, we analyze CM executions in order to extract appropriate features that affect their execution times. Following this, we use the statistical method to estimate the execution times based on collected observations of these extracted features.

In summary, we propose a method that takes input data properties and run-time environment properties as its arguments. We assume that the source code of a CM may be unavailable (e.g. due to licensing). This assumption has the consequences of treating a CM as a black box and does not allow for the extraction of features from the source code.

The models we introduce estimate computation execution time based on historical data. This can be contrasted with the algorithm complexity formulas as in the *big O* notation. To validate our method, we have used simple machine learning algorithms: *SVR*, *KNN* and *POL*¹. More information about the algorithms and how they are suitable can be found in Sect. 3. The conducted experiment

¹ *POL*—Polynomial Regression.

shows the suitability and drawbacks of the listed algorithms to estimate the average execution times.

1.3 Contribution and related work

Our approach to estimation of program execution times is based on the creation of individual estimation models for specific computation programs. As a set of explanatory variables for our models, we take basic properties of the input data (the metadata) together with the properties of the applied run-time environment. Our approach allows the set of explanatory variables to be extended by specific parameters of a program. What is important, no knowledge of the program code is needed. Moreover, creating a separate model for each program instead of embedding the program structure to a vector of features or control flow graphs (CFGs) allows for focusing on each module separately. To our best knowledge, no previous method allows for estimation of execution time based on available resources and input data properties at the same time.

An overview of the execution time estimation problem with the list of possible, known solutions is presented by Kozyrev [4]. This work focuses on the WCET estimation for the purpose of determining the upper limit for computation execution that cannot be exceeded. By contrast, our work attempts at estimating the ACET, which can be used for estimating execution time in typical cases.

Phinjaroenphan et al. [5] note that it is unrealistic to assume that the execution time of a task to be mapped on a node can be precisely calculated before the actual execution. In their method, it is not necessary to know the internal design and algorithm of the application. The estimation is based upon past observations of the task executions. For each execution, they collect a vector of predictors. Each predictor has an impact on the execution time. The employed estimation technique is the K-Nearest-Neighbours algorithm [6]. A similar method is proposed by Iverson et al. [7]. The execution time is treated as a random variable and is statistically estimated from past observations. The method predicts the execution time as a function of several input data parameters. It does not require any direct information about the algorithms used by the tasks or the architecture of the machines. This method also uses the *KNN* algorithm. Our work extends these approaches by introducing new types of predictors: program parameters and execution environment properties. Moreover, we use the *SVR* [8] (Support Vector Regression, originally named *support-vector networks*) and *POL* (Polynomial Regression) algorithms as an alternative to the *KNN* algorithm that has certain drawbacks (see Fig. 6 and its description in Subsect. 3.1) in the context of execution time estimation.

Ermedahl and Engblom have introduced another approach to execution time estimation [9]. They propose to

use static timing analysis tools that work by statically analyzing the properties of a program that affect its timing behavior. These parameters are collected into a specific structure like a vector or a graph. In our approach, we propose to apply static timing analysis to the properties of input data, the run-time environment resources, and the program parameters. This allows focusing on the characteristics of a specific program and its executions, thus potentially receiving more accurate results. However, we have to provide a sufficient amount of model input data for each program, which is challenging. We use the fact that the analyzed programs run in the uniform environment of the BalticLSC system and are reusable. This allows us to collect enough data through repeated executions of CM under various conditions.

In Sect. 3.6 of the work by Haugli [10], we can find a similar approach for carrying out regression by using the features of input data. This method uses the number of image pixels as the only explanatory variable² to estimate the response variable, which is the execution time of an algorithm. A few different image processing algorithms were analyzed. It is worth noting that the choice of model input data properties can be crucial for the proper estimation of execution time. In the estimation models we propose, the set of explanatory variables is fixed for all the CMs. However, this set can easily be extended with other variables for specific types of modules, thus potentially improving the prediction results.

To get more precise results of execution time estimation, one can study the program with similar input data. Shah et al. [11] have done an experiment by generating random lists of data with the same properties (the same metadata) and then training an artificial neural network to predict the *WCET*. They used the *Gem5* system [12] to simulate the run-time environment. By contrast, we have used a real-life container execution environment to receive more realistic data with natural noise introduced to the response variable (execution time). Shah et al. extend their work in the next paper [13], using the concept of surrogate models as a solution to the problem of generating training data. Such generation can increase overheads of the execution time estimation for processing algorithms with heavy input data. It is worth considering as an extension of our approach to generate some input data when the module is new and does not have any executions yet.

Another approach is presented by Meng et al. [14]. They estimate the *WCET* by extracting the program features from object code. Estimation is started when the source code of a program is successfully compiled into the object

² An explanatory variable is a variable that one can manipulate to observe changes, while a response variable is a variable that changes as the result of these manipulations.

code. Then, the extracted features are used for subsequent sample optimization and *WCET* estimation. The authors use the SVR algorithm with the RBF kernel [15] which is similar to what we did in the current research. Having a set of features for each program, we can unify the estimation model to a single instance and provide the features as the explanatory variables for the model. Another solution to the problem of the unsatisfying amount of input data for a model is described by Jenkins and Quintana-Ascencio [16]. The authors explore the problem of a small amount of training data which also affects the results of our work. Generation of testing data and simple algorithms for the execution time estimation is used to reduce the impact of the “small *n*” problem on the final results.

Huang et al. [17] present a more complex approach. They generate a set of features based on the program code (e.g. the number of loops in the source code along with their depth) and include them in the models. In our work, the program code itself is treated as a black box. We stick to the lack of possibility to reach the source code (e.g. for license reasons). Moreover, we use more extensive programs compared to those analyzed by Huang et al. [17]. Extending our models with such features would make the models very complicated and the extension could be rather the subject of whole new research.

2 Data acquisition from past observations

The approach for estimating the execution time of a specific Computation Module (CM) is to create a separate machine learning model. Training data consists of data points (observations) collected during several executions of the CM within some applications inside the BalticLSC system. Each data point consists of a set of explanatory variable values and a response variable value. The collected set of data points makes up the input data³ for the machine learning model. To unify the data acquisition process for the modules analyzed in the article, we have established the following set of explanatory variables.

1. *CPUS* (or *mCPUs*, called *mili-cores*) *limit* The fraction of a physical CPU used to carry out a CM execution,
2. *OVER* The total size of CM input data in bytes,
3. *PART* The number of input data elements,
4. *AVG* The average size of a CM input element,
5. *MAX* The largest element size in model input data (if a CM input data is a set of files, it is the largest size of a file within the set).

³ Under the expression *Input data*, we mean the data forming the input to the machine learning model. Not to be confused with the data input to the CM for data processing (*module input data*).

Figure 2 illustrates the process of extracting a data point from an execution of a CM. Some of the variables are collected only if the input data for a specific CM is a set (of files, database entries, etc.). Otherwise, these variables can be treated as metadata carrying more detailed information about input data of the CM than only the total size. For the data frame type, the number of input elements can be equal to the number of columns. The maximum element size will be a quotient of the total size and the number of columns, and the average size will be equal to that quotient. In other cases (other kinds of CM input data), one can prepare specific, additional features and store their values for each execution to enable their use for training in the future. Figure 3 shows an example of data point acquisition from a specific execution of a module. As one can see, the set of explanatory variables can be extended with specific parameters of a module. In this work, we simplified the input data of a model to the explanatory variables mentioned above. Our dependent value (response variable), which we will estimate, is the execution time of a CM.

We have created two CAL applications based on four CMs with different types of module input data. The first application consists of three modules. It takes the movie as an input, marks people’s faces on each frame, and then returns the movie with marked people’s faces as an output. The second one consists of just a single module, and it searches for the best hyperparameters of the XGBoost algorithm [18] within the parameters grid.

Table 1 describes the modules that we used in the research. For each CM, we have created a set of 20 different module input data sets. Next, we ran the modules with all the mentioned data sets using various CPU resources (from 0.5 CPUs up to 4.0 CPUs with the 0.5 step). Finally, we have received a data frame with the number of 4 (number of modules)*8 (different CPUS resources)*20 (number of module input data sets) = 640 rows (160 per module) that we used to train and validate our models. Part of the model input data is presented in Table 2. The complete data is publicly available on *Mendeley Data* [19].

To make the input data for our models more understandable, we present two figures that show interactions between the explanatory variables and the response variable. Figure 4 shows how much each variable is correlated with the other ones (green indicates positive correlation, red indicates negative correlation). As a correlation coefficient between two variables, we have used Spearman’s rank correlation coefficient [20]. As we can see, the features of the module input data are mostly positively correlated with each other to some extent. They are positively correlated with the execution time as well. The CPUS variable is negatively correlated with the execution time for each module (highest correlation for the *face_recogniser*).

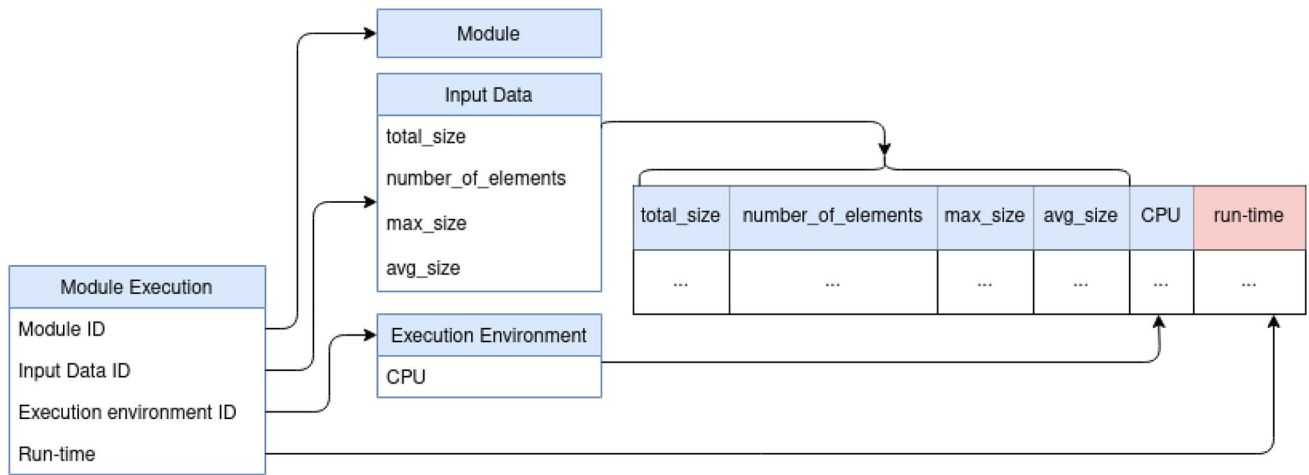


Fig. 2 Visualization of data point acquisition (the same explanatory variables for each module)

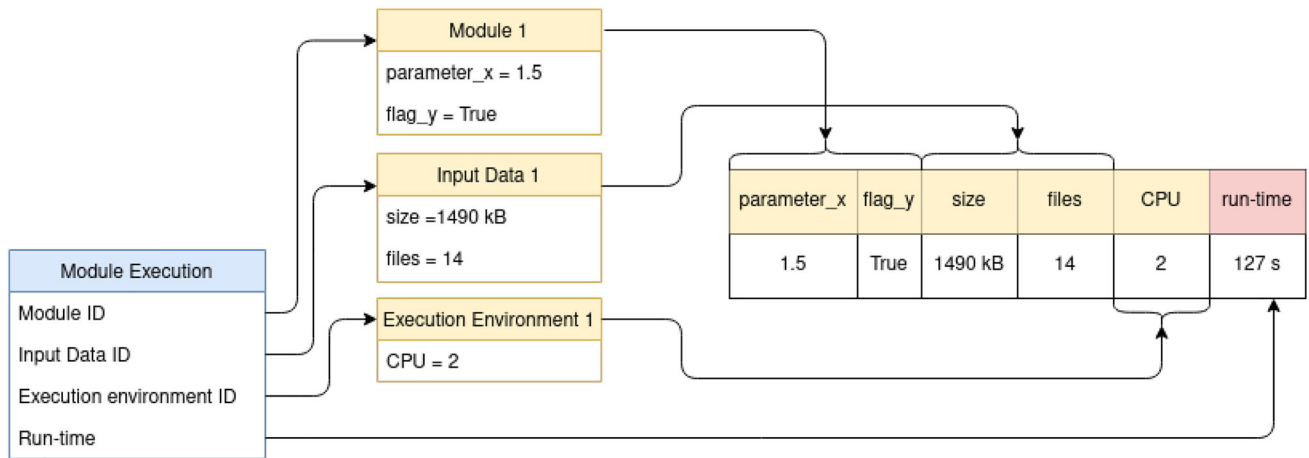


Fig. 3 Extracting a data point from a module execution

Table 1 Modules used in this work

| CM ID (application ID) | Name | Input data type |
|------------------------|-----------------|-----------------|
| 1(1) | Video_splitter | Video file |
| 2(1) | Face_recogniser | Image files |
| 3(1) | Images_merger | Image files |
| 4(2) | XGB_grid_search | CSV file |

As the input data features are strongly correlated with each other, we have performed the *PCA* analysis [21] to determine how specific variables affect the variation of the input data set for a model. Figure 5 shows the results. The *x*-axis denotes *PCA* components (starting from the left with the component having the highest impact on the data set variance). The colors in Fig. 5 show the share of explanatory variables in each of the *PCA* components. CPUS, the only environment-based variable, is not

Table 2 Part of the data frame for model training and validation

| CM ID | CPU | Total size (B) | Avg | Max size (B) | Max size (B) | Time (s) |
|-------|-----|----------------|------|--------------|--------------|----------|
| 1 | 4.0 | 1703379 | 544 | 3131 | 3131 | 3.909 |
| 1 | 4.0 | 809881 | 548 | 1477 | 1477 | 1.981 |
| 1 | 4.0 | 1711796 | 1392 | 1229 | 1229 | 11.371 |
| ... | ... | ... | ... | ... | ... | ... |

Fig. 4 Spearman’s correlation between the explanatory variables and the response variable (Color figure online)

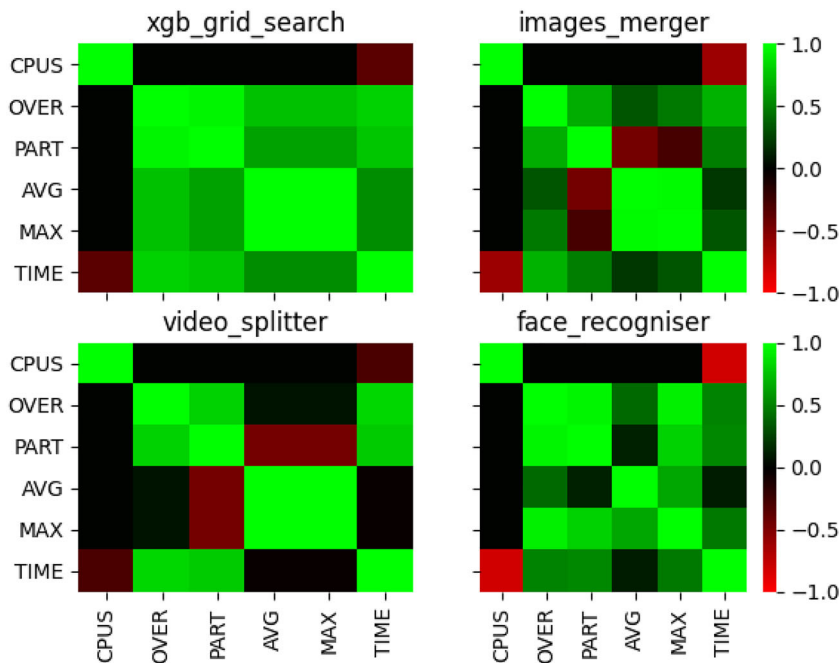


Fig. 5 Importance of PCA variables (Color figure online)



correlated with the other ones and impacts the data variation with a value equal to $1/(\text{number of explanatory variables}) = 0.2$ for each module. The rest of the variance falls onto variables that represent features of the module input data. The *PCA* analysis shows that we could compress our input data to only three components and still keep almost all the variance (the same situation for each module). We kept the original input data for more clarity, but one should consider the reduction of dimensions in the case where more explanatory variables are involved.

3 Choice of the machine learning algorithms

3.1 Selection rationale

The problem we are trying to solve is a small-p regression type. This means that we operate on a low-dimensional space, and our models are based only on a few explanatory variables. Moreover, in most cases, the problem can be classified as a small-n type because of the small number of data points for model training. We do not need complex

machine learning tools like XGBoost [18] or neural networks to receive satisfactory results. The algorithms described in the following two subsections were selected because of their properties and simplicity.

We assume that most of the modules have polynomial time complexity. Following the given assumption, *Polynomial Regression (POL)* algorithm is a natural choice for the type of problem we are trying to solve. For the same reason, we also chose to investigate the *Epsilon-Support Vector Regression (SVR)* algorithm [22]. *SVR* has the flexibility to model any function that is a combination of different polynomials. Moreover, *SVR* can have a better performance when it comes to the regulation of a model.

We have also decided to use the *KNN* algorithm for comparison. The simplest way to predict the execution time is to look at the historical executions of a CM with the model input data that have similar properties to the executing one. That is what the *KNN* algorithm does. A huge disadvantage of the algorithm is the lack of ability to provide acceptable estimations for outstanding data points (data points that fall outside of the range of the training data set). The problem is visualized in Fig. 6 which presents regression on some example input data. As we can see, *KNN* is not able to generalize the estimation for the range above the last training point. At the same time, it can still provide acceptable results for the points within the training range.

Vural and Guillemot [23] study the performance of different machine learning algorithms in low-dimensional space. They compare various algorithms using five different, low-dimensional space data sets. The results of their evaluation show that the *KNN* and the *SVR* perform well in this comparison, at the same time being relatively simple to apply.

3.2 Polynomial regression

In order to perform model training with the use of the *POL* algorithm, we have introduced the following pipeline:

1. Create the polynomial features.
2. Scale the features.
3. Perform the linear regression.

The following hyperparameters have to be established:

1. *Degree* The maximal degree of the polynomial features,
2. *Interaction_only* If *True*, only interaction features for the larger degree are produced,
3. *Include_bias* If *True*, then include a bias column (feature in which all polynomial powers are zero).

All of the hyperparameters concern the creation of the polynomial features.

3.3 Support vector regression

We have used the *SVR* model with the RBF kernel [15] which is the most known flexible kernel, and it could project the vectors of explanatory variables into infinite dimensions. It uses Taylor expansion [24] which is equivalent to the use of an infinite sum over polynomial kernels. It allows the modeling of any function that is a sum of unknown degree polynomials.

When using the kernel, the resulting algorithm is formally similar to the base one, except that every dot product is replaced by a non-linear kernel function. This allows the algorithm to fit the maximum-margin hyperplane in a transformed variables space. The RBF kernel has the following form:

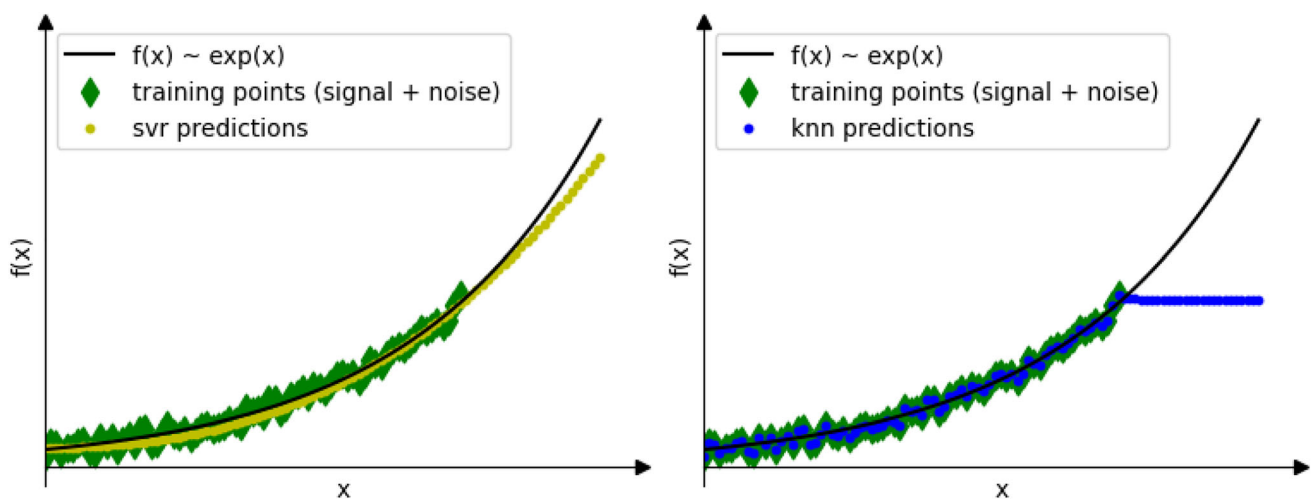


Fig. 6 Using SVR and KNN for regression on the same example data

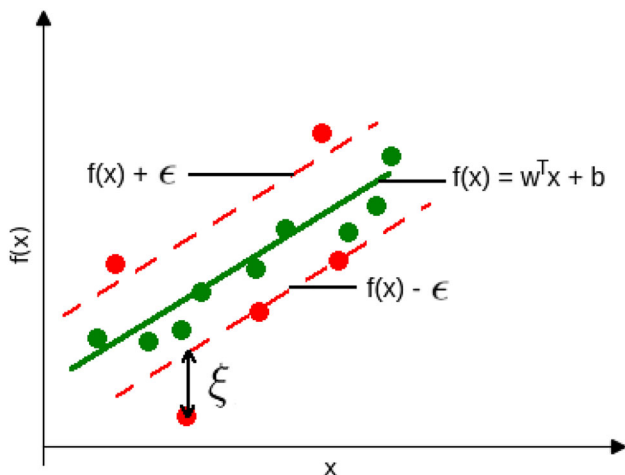


Fig. 7 Visualization of the SVR’s epsilon hyperparameter (Color figure online)

$$K_{RBF}(\vec{x}, \vec{x}') = \exp(-\gamma \|\vec{x} - \vec{x}'\|^2), \tag{1}$$

where $\|\vec{x} - \vec{x}'\|$ euclidean distance between the two vectors of explanatory variables, γ -gamma hyperparameter (see below).

In the SVR algorithm we look for a hyperplane y in the following form:

$$y = \vec{w} \vec{x} + b, \tag{2}$$

where \vec{x} is the vector of explanatory variables, \vec{w} is the vector normal to the hyperplane y ; when using a kernel, the \vec{w} vector is also in the transformed space.

Training the original SVR means solving:

$$\frac{1}{2} w^2 + C \sum_i^N (\xi_i + \xi_i^*) \tag{3}$$

with the following constraints:

$$y_i - \vec{w}x_i - b \leq \epsilon + \xi_i \quad -y_i + \vec{w}x_i + b \leq \epsilon + \xi_i^* \quad \xi_i \xi_i^* \geq 0 \tag{4}$$

Figure 7 shows an example of a hyperplane with marked ξ and ϵ parameters. As we have already chosen the *RBF* kernel for the *SVR* algorithm, our modeling is simplified just to find the best values of the following hyperparameters:

1. *C* The weight of an error cost. The regularization⁴ hyperparameters have to be strictly positive. The example from the Fig. 7 has l1 penalty applied (the used modeling library has the squared epsilon-insensitive loss with l2 penalty applied). The strength of the

regularization is inversely proportional to *C*. The larger the value of *C*, the more variance is introduced into the model. A module’s execution time can depend on variables that we do not consider and would introduce noise to the final result. The machine learning algorithm should be able to generalize the result to eliminate the noise impact. It can be done by well-selected regularization parameters.

2. *Epsilon* It specifies the epsilon-tube where no penalty is associated with the training loss function. In other words, the penalty is equal to zero for points predicted within a distance ϵ from the actual value of prediction. As it is shown in Fig. 7 the green data points do not provide any penalty to the loss function because they are within the allowed epsilon range around the approximation⁵.
3. *Gamma* The *gamma* hyperparameter can be seen as the inverse of the radius of influence of samples selected by the model as support vectors. Increasing the value of *gamma* hyperparameter causes the variance increase what is shown in Fig. 8⁶.

To implement the algorithm we have used the scikit-learn library. Its documentation [25] presents more details on implementing the *SVR*-class algorithms. We refer to the documentation which provides detailed information about the *C* and *gamma* hyperparameters [26].

3.4 K-nearest neighbors regression

In the *KNN* algorithm, the target is predicted by local interpolation of the targets associated with the nearest neighbors in the training set [27]. In other words, the target assigned to a query point is computed based on the mean of the targets of its nearest neighbors.

To use the algorithm, we have to define the values of the following hyperparameters:

1. *k* Number of the nearest neighbors,
2. *Weights* Weight function used in prediction. The basic nearest neighbors regression uses uniform weights: each point in the local neighborhood contributes uniformly to the classification of a query point. Under some circumstances, it can be advantageous to weight points such that nearby points contribute more to the regression than faraway points. The weights can be calculated from the distances using any function, a linear one, for example.
3. *Procedure* The algorithm to calculate k-nearest neighbors for the query point. It does not directly impact the

⁴ Regularization is a way to give a penalty to certain models (usually overly complex ones).

⁵ The figure is based on some example data and it only shows the hyperparameter influence on a model.

⁶ See footnote 5.

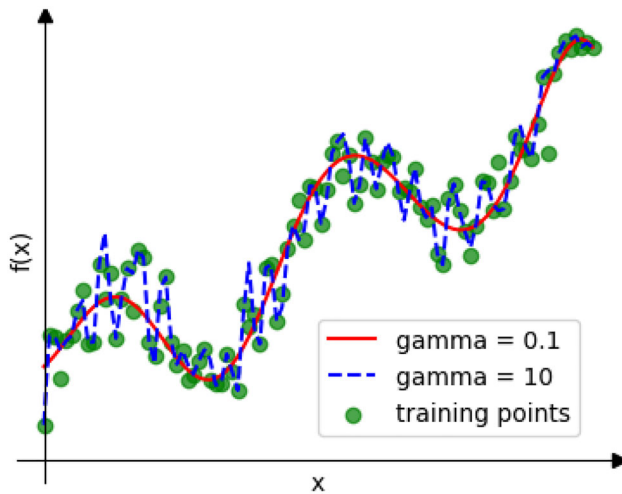


Fig. 8 Influence of the SVR *gamma* hyperparameter on the model variance

final regression result, but the algorithm's parameters do. For example, using the BallTree algorithm, we have to choose the *metric* parameter that will be used to calculate the distance between data points. For example, this could be the Minkowski [28] metric with the l2 (standard Euclidean [29]) distance metric or any function that will calculate the distance between two points.

4 Training and validation

4.1 Training

The training pipeline for each module contains the following steps:

1. Having the 160 data points (see the description of data acquisition in Sect. 2), we divide them into training and test data sets with the 120:40 proportion. We set aside the test data to use it only for the validation of the final models.
2. Standardization of a data set is a common requirement for many machine learning estimators: they might behave badly if the individual explanatory variables do not approximately look like standard normally distributed data (e.g. Gaussian with 0 mean and unit variance) [30]. We scale each column (explanatory variable) of the data set using the following formula:

$$\vec{x}_f^j = (\vec{x}_f - \mu_f) / \sigma_f, \quad (5)$$

where \vec{x}_f^j vector with values from the column f , μ_f mean of the column f , σ_f standard deviation of the column f .

3. Each algorithm has several hyperparameters that should be chosen wisely to achieve satisfactory results. It is hard to predict the best value of a continuous parameter. What we did is an exhaustive search over specified parameter values. For each combination of the parameter values, we validate the model using fivefold cross-validation on the training data set. This procedure is called the *grid search* [31]. The hyperparameter grids for the algorithms are included in Table 3. The ranges were chosen based on several test runs of the learning pipeline. Then, we have established a few values from the given ranges ensuring proper distribution of the values.
4. Finally, we have re-trained our model using the entire training data set and the hyperparameters that have been found in the previous step.

In the *grid search* procedure the determination coefficient R^2 has been used as the performance metric of the prediction.

We assume that each BalticLSC module will be executed many times. This assumption entails many training data points for each model. However, there will be diversities between the number of executions per module. Moreover, the execution time of new modules cannot be predicted due to the lack of training sets. To investigate how the number of training samples affects the relative error of the regression, we have calculated the learning curves. The results are shown in Fig. 9. In line with what could be expected, the relative error decreases (with some hesitations) as the amount of training data increases for each module.

4.2 Validation

Since our models were trained only on the training data sets, we had the possibility to use the test data set as completely new data to validate our models. The separated test data guarantees that the validation results will not be distorted. Training with cross-validation secures the model from overfitting thereby increasing the overall accuracy. In the BalticLSC system, we would like to conduct the prediction process even with data from a small number of historical executions. The formula of relative error is the average from the errors of models trained with a different number of samples. Such metrics indicate that models should perform well even for a small number of training samples. The relative error is described by the following equation:

Fig. 9 Learning curve per computational module

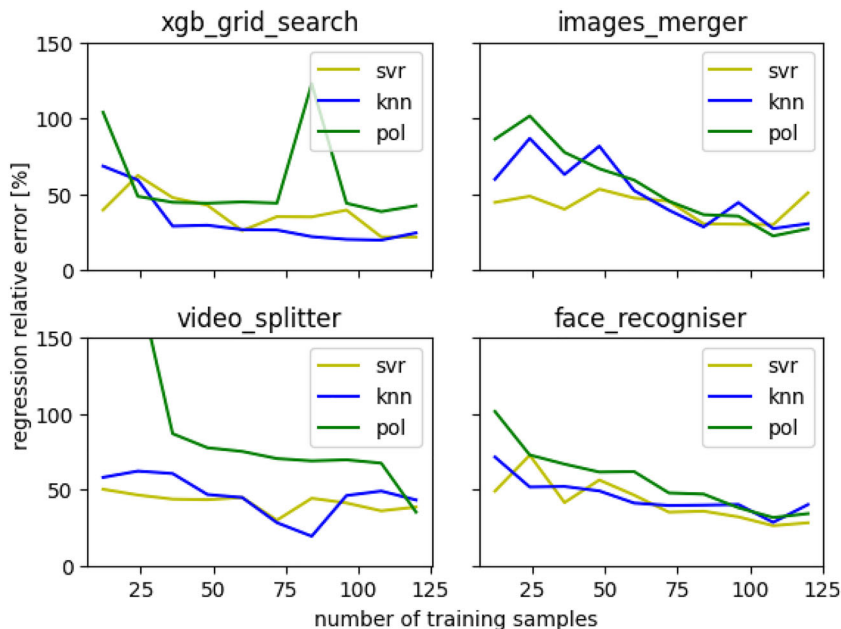


Table 3 Hyperparameter grid for each algorithm

| Algorithm | Hyperparameter | Set of values |
|-----------|---------------------|--|
| SVR | Γ | $\{2^n e^{-04} \in [0, 7] \wedge n \in I\}$ |
| | ϵ | $\{2^n e^{-06} \in [0, 10] \wedge n \in I\}$ |
| | C | $\{2^n e^{03} \in [0, 11] \wedge n \in I\}$ |
| KNN | $n_neighbors$ | $\{n \in [1, 11] \wedge n \in I\}$ |
| | $Weights$ | {‘uniform’, ‘distance’} |
| | p | {1, 2} |
| POL | $Degree$ | $\{n \in [1, 5] \wedge n \in I\}$ |
| | $Interaction_only$ | {‘False’, ‘True’} |
| | $Include_bias$ | {‘False’, ‘True’} |

$$Error = \frac{1}{F} \sum_f \frac{1}{N_f} \sum_i \frac{\|y_i - \bar{y}_i\|}{y_i}, \tag{6}$$

where F number of test data fractions = 10,
 f fraction identifier,
 N_f number of training data points for the fraction f ,
 where $f \in \{q * 12, q \in [1, 10] \wedge q \in I\}$
 y - original execution time of a module.,
 \bar{y}_i -predicted execution time.

Table 4 shows the final results of the estimation error for each module and algorithm. The so-called *reference* model is a model trained only on the two basic explanatory variables: CPUS and OVER. We did research on such reduced models to check if the additional features of input data had any positive influence on the final models.

One could ask how adding the additional features will affect the training time of a model. When it comes to using *POL* or *KNN* algorithm, training is quite fast due to the low complexity of the algorithms. The training was conducted on an average machine with eight cores. It takes less than a second for every model when using full training data (120 samples). However, training a model based on the *SVR* is more computationally demanding. On average, for a single model, using the full training data, training times are around 137 s and 594 s for the *reference* and the *original* method, respectively.

Additionally, we have conducted the Kolmogorov–Smirnov test to check if the differences between the results of models are statistically significant. In the Subject. 5.1, one can find the conclusions from the comparison between *reference* model and the *original* one.

Figures 10 and 11 show the comparisons of regression surfaces⁷ for chosen modules and algorithms. One can see that the *KNN* algorithm created a more irregular surface than the *SVR*. The mentioned irregularity also applies to other modules, the surfaces of which are not attached in the article. This indicates better generalization when using the *SVR* algorithm. Besides the surfaces, the figures show training and test data points as green and red dots respectively. The input data set was split randomly for each module in the same way for each algorithm. One can see that the data points for the *face_recogniser* module have a more uniform distribution for the overall size variable. It leads to a more smooth regression surface and smaller

⁷ To illustrate the surface on 3D plot, only the CPUS and OVER features are plotted. The z-axis (t label) represents the execution time.

Table 4 Comparison of relative errors between *original* and *reference* models

| method: | original | | | reference | | | | | |
|-----------|--------------------|-------------|-------------|-------------|-------------|-------------|-------------|------|------|
| algorithm | SVR | KNN | POL | SVR | KNN | POL | SVR | KNN | POL |
| CM ID | relative error [%] | | | | | | p-value [%] | | |
| 1 | 41.8 | 45.7 | 91.4 | 37.2 | 33.0 | 44.0 | 70.0 | 18.0 | 0.0 |
| 2 | 42.2 | 45.3 | 56.3 | 51.0 | 48.9 | 53.7 | 0.1 | 21.1 | 36.7 |
| 3 | 42.0 | 51.3 | 55.8 | 50.7 | 51.7 | 45.8 | 90.7 | 1.0 | 36.7 |
| 4 | 37.1 | 32.5 | 57.7 | 41.4 | 45.9 | 50.2 | 0.0 | 0.0 | 1.3 |
| average | 40.8 | 43.7 | 65.3 | 45.1 | 44.9 | 48.4 | 40.2 | 10.0 | 18.7 |

relative error (see the final results in Table 4, model *original*).

5 Conclusion

5.1 Observations

As the *PCA* analysis shows (see Fig. 5 and its description), the input data for our models have approximately three independent components that contain all variations of the data. This means that by using these components, we will reduce the model to only a three-dimensional space of explanatory variables. We did not decide to use the reduced space, but one can use the reduction to more complex cases because each set of explanatory variables of a model can be extended with additional features for the specific module. For example, there could be some additional parameters required to run a module, and the parameters have an impact on the execution time. These parameters should be included in the set of input data for the model of the module.

Each module has its own model. Every module execution in the BalticLSC system will be the next training data point (observation) for a model. If a module has many properties that have an impact on the execution time then it will be harder to train the model because of the high-dimensional space of the explanatory variables. Moreover, without executions of a module, there is no training data at all, which can be solved by introducing testing executions. However, one should have in mind, that these testing executions use resources that generate costs. In this context, it is desirable to establish the minimum number of testing executions. Our work shows that several dozen executions per module guarantee an estimation error of an acceptable value (see the learning curves in Fig. 9).

Additionally, we have researched models with a reduced set of explanatory variables. We call the model as *reference* since it represents the model with the most basic set of features required to conduct the experiment. Such a

reduced set consists of only two variables: CPUS (the fraction of a physical CPU used to carry out the execution) and OVER (the total size of a CM input data in bytes). When it comes to analyzing the results of the POL algorithm (Table 4), one can see that the *reference* model has significantly better results. The reason for this is probably the large number of polynomial features generated along with the creation of the *original* model. Since the *reference* model has fewer explanatory variables, it has fewer polynomial features as well. More features, along with a small number of training samples, can lead to a decrease in the overall performance of a model. Both SVR and KNN algorithms have slightly better results when the extended set of features is used. However, since the *p-values* of Kolmogorov–Smirnov tests are high, one should deduce that the differences cannot be described as statistically significant.

Using the POL algorithm leads to significantly worse results when compared to KNN or SVR. One can find the reason for that in the previous paragraph of the current subsection. Both KNN and SVR algorithms give relatively similar results with a bit of advantage for the SVR (average, the relative error of 40.8%, comparing to 43.7% for the KNN). The more critical characteristic which makes the SVR algorithm more promising in use is the possibility to estimate the execution time in a more general way for the inputs that are out of the training range (see Fig. 6 and its description in Sect. 3).

In the context of a real-life environment, a single model can initially have a small number of training samples according to the small number of module executions. We have measured relative error for different sizes of training sets and computed an average according to Eq. 6. Though, it should be noted that this error is a few points lower when we only consider the model trained on the full training set (120 samples). For example, in the case of the SVR algorithm, we end up with an error equaling 40.8% on average. For a full training set, the error equals 34.8%.

The error could be potentially reduced by using well-chosen and representative training samples. This way, we

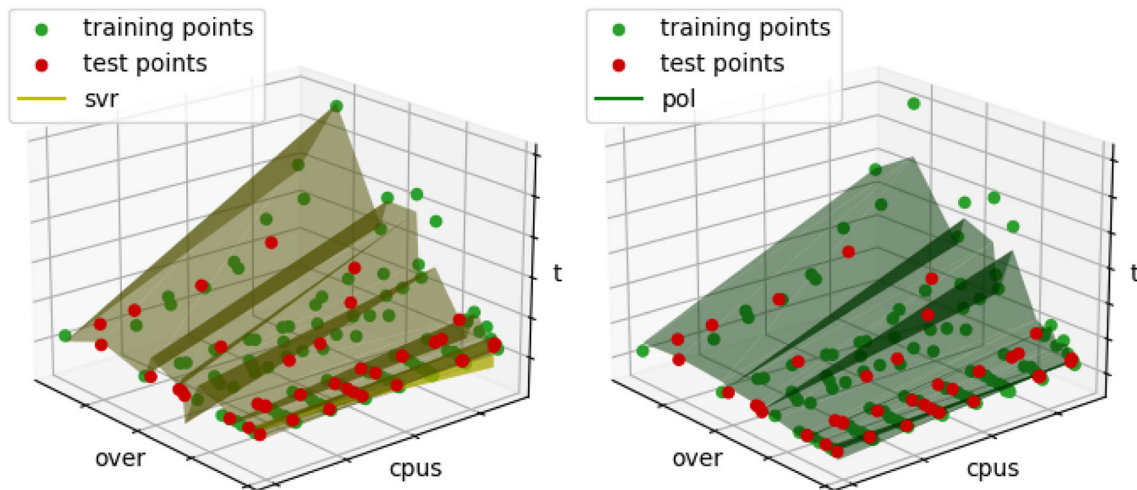


Fig. 10 Regression surfaces for *images_merger* module

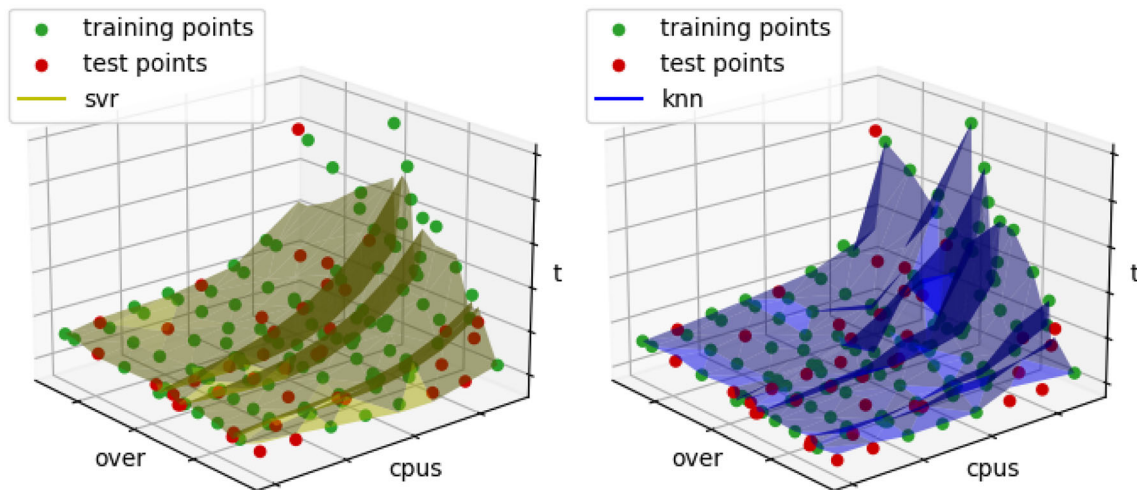


Fig. 11 Regression surfaces for *face_recogniser* module

could reduce the probability of missing information about subspaces where the variability of a mapping function is significant. However, in a real-life context, the training samples are created during the actual runs of the modules. Thus, to be more tied with the real-life environment, we did not strongly focus on providing representative samples. By providing random rather than representative samples, we ended up with slightly worse (but more adequate for real-life use) results.

When using the additional features (the *original* method), there is an evident increase in the training time of the model based on the *SVR* algorithm (594 s as compared to 137 s using the *reference* model). Despite the increase, the training time for models using the *original* method is still acceptable. However, when significantly increasing the number of features or/and the number of training samples,

attention should be paid to the training time of models based on the *SVR* algorithm.

5.2 Future work

Considering the obtained accuracy of the proposed method we can see that there is still significant room for improvement. Thus, our future work will concentrate on further improvement of the presented models. Explanatory variables should explain the input object as much as possible in the context of estimating the response variable. One could determine a different set of explanatory variables for each input data type or even for each computation module. Currently, we use the same set of explanatory variables for every module to simplify the process of verifying the general idea. Future work can extend this

through researching more specific explanatory variables for particular modules or module types.

We are going to build one model for all the modules, based on the control flow graphs embedded in vectors. In such a case, we will leave the assumption of treating the source code of a module as a black box.

Moreover, it should be noticed that our method works for single modules only. Still, we would expect to be able to predict execution times for applications consisting of several connected modules (see, e.g. Fig. 1). In this case, we need to predict input data properties for consecutive modules. This is not an easy problem because we do not know the exact characteristics of data passed between modules (produced on output by one module and treated as input to another module). One idea for estimating the execution time of whole applications is to create predictive models. The input data for the models should contain the metadata of the input data for the entire application and the encoded application structure. The output of the models will provide full metadata of consecutive input data sets for each execution of a module within the execution of the application. This forms an interesting research agenda that we plan to undertake as future work.

Funding This work is partially funded from the European Regional Development Fund, Interreg Baltic Sea Region programme, Project BalticLSC #R075.

Data Availability Enquiries about data availability should be directed to the authors.

Declarations

Competing interests The authors have not disclosed any competing interests.

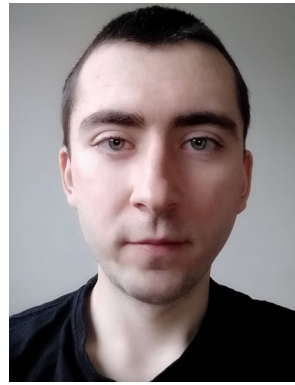
Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

- Marek, K., Śmiałek, M., Rybiński, K., Roszczyk, R., Wdowiak, M.: BalticLSC: low-code software development platform for large scale computations. *Comput. Inform.* **40**(4), 734–753 (2021)
- Roszczyk, R., Wdowiak, M., Śmiałek, M., Rybiński, K., Marek, K.: BalticLSC: a low-code HPC platform for small and medium research teams. In: 2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), pp. 1–4. IEEE (2021)
- Nasar, A.A.: The history of algorithmic complexity. *Math. Enthus.* **13**(3), 217–242 (2016)
- Kozyrev, V.: Estimation of the execution time in real-time systems. *Program. Comput. Softw.* **42**(1), 41–48 (2016)
- Phinjaroenphan, P., Bevinakoppa, S., Zeepongsekul, P.: A method for estimating the execution time of a parallel task on a grid node. In: European Grid Conference, pp. 226–236 (2005). Springer
- Taunk, K., De, S., Verma, S., Swetapadma, A.: A brief review of nearest neighbor algorithm for learning and classification. In: 2019 International Conference on Intelligent Computing and Control Systems (ICCS), pp. 1255–1260 (2019). <https://doi.org/10.1109/ICCS45141.2019.9065747>
- Iverson, M.A., Ozguner, F., Potter, L.C.: Statistical prediction of task execution times through analytic benchmarking for scheduling in a heterogeneous environment. In: Proceedings. Eighth Heterogeneous Computing Workshop (HCW'99), pp. 99–111. IEEE (1999)
- Cortes, C., Vapnik, V.: Support-vector networks. *Mach. Learn.* **20**(3), 273–297 (1995)
- Ermedahl, A., Engblom, J.: Execution time analysis for embedded real-time systems. Citeseer (2007). <https://doi.org/10.1201/9781420011746.ch35>
- Haugli, F.B.: Using online worst-case execution time analysis and alternative tasks in real time systems. Master's Thesis, Institutt for teknisk kybernetikk (2014)
- Shah, S.A.B., Rashid, M., Arif, M.: A prediction model for measurement-based timing analysis. In: Proceedings of the 6th International Conference on Software and Computer Applications, pp. 9–14 (2017)
- Gem5 website. <https://www.gem5.org/>, last visited 12.07.2021 (2021)
- Rashid, M., Shah, S.A.B., Arif, M., Kashif, M.: Determination of worst-case data using an adaptive surrogate model for real-time system. *J. Circuits Syst. Comput.* **29**(01), 2050005 (2020)
- Meng, F., Su, X., Qu, Z.: Nonlinear approach for estimating WCET during programming phase. *Clust. Comput.* **19**(3), 1449–1459 (2016)
- Nisbet, R., Miner, G., Yale, K.: Chapter 7—basic algorithms for data mining: a brief overview. In: Nisbet, R., Miner, G., Yale, K. (eds.) *Handbook of Statistical Analysis and Data Mining Applications*, 2nd edn., pp. 121–147. Academic Press, Boston (2018). <https://doi.org/10.1016/B978-0-12-416632-5.00007-4>
- Jenkins, D.G., Quintana-Ascencio, P.F.: A solution to minimum sample size for regressions. *PLoS ONE* **15**(2), 0229345 (2020)
- Huang, L., Jia, J., Yu, B., Chun, B.-G., Maniatis, P., Naik, M.: Predicting execution time of computer programs using sparse polynomial regression, **23**, 883–891 (2010). <https://proceedings.neurips.cc/paper/2010/file/995665640dc319973d3173a74a03860c-Paper.pdf>
- Chen, T., Guestrin, C.: XGBoost: a scalable tree boosting system. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 785–794 (2016)
- Bielecki, J.: [dataset] Details of the tasks executions. Mendeley Data (2022). <https://doi.org/10.17632/sbfmwwv77d.2>
- Schober, P., Boer, C., Schwarte, L.A.: Correlation coefficients: appropriate use and interpretation. *Anesth. Analg.* **126**(5), 1763–1768 (2018)

21. Jolliffe, I.T., Cadima, J.: Principal component analysis: a review and recent developments. *Philos. Trans. R. Soc. A* **374**(2065), 20150202 (2016)
22. Awad, M., Khanna, R.: Support vector regression. In: *Efficient Learning Machines: Theories, Concepts, and Applications for Engineers and System Designers*, pp. 67–80. Apress, Berkeley (2015). https://doi.org/10.1007/978-1-4302-5990-9_4
23. Vural, E., Guillemot, C.: A study of the classification of low-dimensional data with supervised manifold learning. *J. Mach. Learn. Res.* **18**(1), 5741–5795 (2017)
24. Kouki, R., Griffiths, B.J.: Introducing Taylor series and local approximations using a historical and semiotic approach. *Int. Electron. J. Math. Educ.* (2020). <https://doi.org/10.29333/iejme/6293>
25. Cournapeau, D., et al.: Support Vector Regression. Scikit-learn version 0.24.2 (visited 10.04.2021). <https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html>
26. Cournapeau, D., et al.: RBF SVM parameters. Scikit-learn version 0.24.2 (visited 10.04.2021). https://scikit-learn.org/stable/auto_examples/svm/plot_rbf_parameters.html
27. Cournapeau, D., et al.: KNN Regression. Scikit-learn version 0.24.2 (visited 10.04.2021). <https://scikit-learn.org/stable/modules/neighbors.html#regression>
28. Li, Z., Ding, Q., Zhang, W.: A comparative study of different distances for similarity estimation. In: *International Conference on Intelligent Computing and Information Science*, pp. 483–488. Springer (2011).
29. Clapham, C., Nicholson, J.: *The Concise Oxford Dictionary of Mathematics*. Oxford University Press, Oxford (2009). <https://doi.org/10.1093/acref/9780199235940.001.0001>
30. Cournapeau, D., et al.: Standard scaler. Scikit-learn version 0.24.2 (visited 18.04.2021). <https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html>
31. Cournapeau, D., et al.: Grid search. Scikit-learn version 0.24.2 (visited 18.04.2021). https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Jan Bielecki is currently a PhD student at Faculty of Electrical Engineering, Warsaw University of Technology. His current research interests include data processing and high-performance computations. He love data and the fully open data concept.



Michał Śmiałek is a professor of Software Engineering at the Warsaw University of Technology. He obtained a habilitation (higher doctorate) degree in informatics from the Warsaw Military University and has graduated from the Warsaw University of Technology (MSc and PhD) and the University of Sheffield (MSc). Since 1991 he has worked in the industry as a software developer, project manager and professional coach. His current research interests include model-driven software development, requirements engineering, software reuse, software language engineering and large-scale computing. He published and edited several books and over 100 papers in various journals and conference proceedings. He coordinated two European-level research projects, chaired several international conferences and reviewed for major computer science journals.